# Porting and Evaluating Return-Oriented Programming Defenses Implemented by the OpenBSD Operating System

Jenna Esposito
*Dept. of Computer Science and Software Engineering*
*Monmouth University*
West Long Branch, NJ, USA
0009-0007-1455-8135

Aaila Arif
*Dept. of Computer Science and Software Engineering*
*Monmouth University*
West Long Branch, NJ, USA
0009-0002-4152-9165

Raul Cortinas
*Dept. of Computer Science and Software Engineering*
*Monmouth University*
West Long Branch, NJ, USA
0009-0001-8180-4400

Brian Robert Callahan
*Dept. of Computer Science and Software Engineering*
*Monmouth University*
West Long Branch, NJ, USA
0000-0002-1797-8633

***Abstract*—Return-oriented programming (ROP) continues to be a serious attack vector nearly a decade and a half after first being disclosed. Techniques to circumvent this style of attack include both hardware- and software-based solutions. CPUs that lack the hardware capabilities must rely solely on software-based solutions.**

**In 2019, the OpenBSD project introduced a number of software-based anti-ROP mitigations in their copy of the LLVM compiler suite. They claimed significant reductions in unique gadgets in the OpenBSD kernel and libc with zero to minimal drawbacks. Such solutions have not been ported elsewhere nor integrated into upstream LLVM.**

**In this paper, we port two of OpenBSD's software-based anti-ROP mitigations for x86_64 CPUs, alternative register selection and compile-time instruction rewriting, to the FreeBSD operating system. We then measure their effects on reduction of unique gadgets, binary size, and runtime performance.**

**We were able to corroborate some the claims of minimal size and runtime penalties; gadget reduction was substantially less than claimed and less effective than claimed. Our investigation demonstrates the potential difficulties of seemingly obvious compiler-based anti-ROP techniques.**

***Index Terms*—*return-oriented programming, compilers, mitigation, operating system security***

This is a preprint of the accepted version; the version of record can be found at https://ieeexplore.ieee.org/document/11458911.

Full citation:
J. Esposito, A. Arif, R. Cortinas, and B. R. Callahan, "Porting and evaluating return-oriented programming defenses implemented by the OpenBSD operating system," in *Proc. 14th Int. Symp. Digit. Forensics Secur. (ISDFS)*, Boston, MA, USA, 2026, pp. 1-7, doi: https://doi.org/10.1109/ISDFS69419.2026.11458911.

## I. Introduction

Return-oriented programming (ROP) has proven a powerful binary exploitation technique since its initial discovery nearly fifteen years ago [1]. Enabled by the creative re-use of already existing code found in legitimate binaries and libraries on systems, the attack continues to be not only viable but practical in the modern day [2]. It is an attack that spans operating systems and CPU architectures.

With the recent phenomenon of skyrocketing prices of computing components, we may well see a rise in machines being used longer than perhaps initially intended when originally purchased. While there have been significant strides in defeating ROP attacks such as control flow integrity [3], at least some of those advances depend on hardware functionality that may not exist in older CPUs.

In such an event, software-focused solutions may prove invaluable to protect users against ROP attacks. Such software-based mitigations have been proven for other kinds of attacks, such as canaries for memory corruption attacks [4]. Some software-based mitigations proved so effective they became hardware-based, such as the W ^ X technique to defeat shellcode [5], finding its way into x86 hardware in the form of the NX bit.

In this article, we evaluate a pair of potential compiler-based, purely software mitigations designed to prevent ROP attacks. These mitigations, originally developed and implemented by the OpenBSD operating system, claim significant reduction in ROP gadgets while having neglibile costs to runtime performance and binary sizes [6].

From OpenBSD, we port two of these mitigations to the FreeBSD operating system: a reorganization of register allocation order and an additional optimization pass for the LLVM compiler toolchain to rewrite potential ROP instructions at compile time. Focusing solely on the x86_64 CPU architecture, we port these mitigations both individually and as a set. For each combination, we evaluate runtime performance, binary size differences, and changes in the number of unique gadgets across ten real-world open source applications. We also evaluate binary size differences and changes in the number of unique gadgets for the FreeBSD kernel itself and the FreeBSD C standard library.

Our contributions to the literature include a systematic evaluation of these software-based ROP attack mitigations and discussion as to the relative ease and difficulty to add

additional layers of defense for a still active attack vector. It is our hope that this article spurs continued investigation into software- and compiler-based mitigations for ROP and other attack vectors.

## II. Background

ROP attacks are an evolution of shellcoding attacks that rely not on the attacker creating their own set of machine instructions, but on the creative reuse of instructions already present in legitimate binaries and libraries found on the target system.

ROP attacks focus on using what are called gadgets, tiny snippets of already existing code. Gadgets usually terminate with a return instruction, coded as `ret` in assembly code and compiled to a `c3` byte in object code. These instructions could be aligned, meaning that the `c3` byte represents an actual return instruction in the code, or they can be unaligned or polymorphic, in which a `c3` byte is part of the encoding of another instruction. Because x86_64 CPUs have both variable-length instructions and unaligned access [7], a CPU can be told to simply begin execution at an arbitrary location in memory. In such a case, the CPU will reinterpret subsequent instructions. To take one example:

```
48 c7 c0 4d aa 00 00    movq $0xaa4d, %rax
48 01 c3                addq %rax, %rbx
```

If execution begins at the third byte in the sequence, the entire sequence will be reinterpreted as:

```
c0 4d aa 00     rorb $0x0, -0x56(%rbp)
00 48 01        addb %cl, 0x1(%rax)
c3              retq
```

Thus producing a gadget in already existing, legitimate, code.

In 2019, OpenBSD developer Todd Mortimer developed a series of mitigations designed to make ROP gadget generation more difficult [6]. Leveraging the LLVM compiler suite, the default compiler toolchain for the OpenBSD operating system, Mortimer made a number of observations as to the likelihood of the compiler creating polymorphic gadgets; the two we focus on in this paper are alternative register selection and compile-time instruction rewriting.

### A. Alternative register selection

First, the order in which registers are selected may have a difference in the number of polymorphic gadgets created. To take a common example, an arithmetic instruction such as `addq` will generate a polymorphic gadget when the source register is `%rax` and the destination register is `%rbx`. As `%rax` is perhaps the most commonly used register and `%rbx` is a common nonvolatile, or callee saved, register, this register combination combined with an arithmetic instruction may occur relatively frequently. The end result is a potential polymorphic gadget. As polymorphic gadgets rely on the x86_64 CPU's property of unaligned access of instructions, that means parts of previous instructions can be combined with this particular class of instruction to produce powerful and desirable polymorphic gadgets.

A simple but effective way to handle this situation is to not put the compiler in a position where it would generate such instructions in the first place. For the LLVM compiler suite, this means altering the register selection order. For LLVM 19.1.7, the version of LLVM we used, the register selection order list for 64-bit registers can be found in the LLVM GitHub source code tree at https://github.com/llvm/llvm-project/tree/llvmorg-19.1.7 in file `llvm/lib/Target/X86/X86RegisterInfo.td` starting on line 585:

```
def GR64 : RegisterClass<"X86", [i64], 64,
(add RAX, RCX, RDX, RSI, RDI, R8, R9, R10,
    R11, R16, R17,
R18, R19, R20, R21, R22, R23, R24, R25,
    R26, R27, R28, R29,
R30, R31, RBX, R14, R15, R12, R13, RBP,
    RSP, RIP)>;
```

And for 32-bit registers in the same file starting on line 574:

```
def GR32 : RegisterClass<"X86", [i32], 32,
(add EAX, ECX, EDX, ESI, EDI, EBX, EBP,
    ESP, R8D, R9D,
R10D, R11D, R16D, R17D, R18D, R19D, R20D,
    R21D, R22D,
R23D, R24D, R25D, R26D, R27D, R28D, R29D,
    R30D, R31D,
R14D, R15D, R12D, R13D)>;
```

The OpenBSD solution is to take the `RBX`/`EBX` register and place it at the back of the list. On OpenBSD 7.8, the version of OpenBSD we used, the changes for 64-bit register selection can be found in the OpenBSD GitHub source code tree at https://github.com/openbsd/src/tree/9e0329e9c4709c5bb9b6be969d324e91acca88e5 in file `gnu/llvm/llvm/lib/Target/X86/X86RegisterInfo.td` starting on line 426:

```
def GR64 : RegisterClass<"X86", [i64], 64,
(add RAX, RCX, RDX, RSI, RDI, R8, R9, R10,
    R11,
R14, R15, R12, R13, RBX, RBP, RSP, RIP)>;
```

And for 32-bit registers in the same file starting on line 417:

```
def GR32 : RegisterClass<"X86", [i32], 32,
(add EAX, ECX, EDX, ESI, EDI,
R8D, R9D, R10D, R11D, R14D, R15D, R12D,
    R13D, EBX, EBP, ESP)>;
```

As we can see in these lists, for the 64-bit registers, the `%rbx` register is moved to the back of the list; only `%rbp` and `%rsp` are further behind. As `%rsp` is used as the stack point and `%rbp` is only used as a general purpose register when the `-fomit-frame-pointer` option is used and a stack frame does not need it, this effectively moves `%rbx` to the end of the list. We see the same movement of `%ebx` for

the 32-bit registers. The `%r16` through `%r31` registers are not available to all x86_64 CPUs and are for us not considered in the overall list.

Mortimer [6] claims a 6% reduction in gadgets from the OpenBSD kernel with this technique, with the change being "entirely free" as there was no additional runtime cost or compile-time cost.

### *B. Compile-time instruction rewriting*

The second observation is that we know at compile-time if the compiler is about to output an instruction that could become a polymorphic gadget. For example, the compiler knows when and where it will write a `subq %rax, %rbx` instruction or other instruction that could become a polymorphic gadget. In those situations, we can instruct the compiler to instead output a semantically equivalent set of instructions that are not able to be used as a polymorphic gadget [6].

Taking the `subq %rax, %rbx` example, the LLVM and GCC compilers will both produce this byte sequence:

```
48 29 c3          subq    %rax, %rbx
```

As mentioned before, it is the `c3` byte that enables this instruction's potential use as a polymorphic gadget.

We can teach the LLVM compiler to output the following semantically equivalent set of instructions whenever it is about to create a `subq %rax, %rbx` instruction:

```
xchgq    %rbx, %rax
subq     %rbx, %rax
xchgq    %rbx, %rax
```

That is to say, we can use the `xchgq` instruction to flip the source and destination registers, perform the arithmetic instruction `subq` on the flipped registers, and then use `xchgq` again to flip the registers back how they originally were. This creates the following byte sequence:

```
48 93             xchgq   %rbx, %rax
48 29 d8          subq    %rbx, %rax
48 93             xchgq   %rbx, %rax
```

Now the arithmetic instruction does not produce a `c3` byte; the encoding for the final byte of the `subq` instruction is now `d8`. No `c3` byte, no polymorphic gadget. This does come with two obvious penalties: first, the two additional `xchgq` instructions adds four bytes to the final binary size per instance; and, second, while `xchgq` might be a fast instruction, it is not free in terms of runtime. It is theoretically possible that an exceptionally large number of `xcghq` instructions could result in a significant performance penalty.

Because LLVM is highly modular and its optimization passes are effectively a single list that is iterated over one optimization pass at a time, it is easy to implement one's own optimization passes. This is what OpenBSD did to gain this feature. In the OpenBSD GitHub source tree, they created a new file named `gnu/llvm/llvm/lib/Target/X86/X86FixupGadgets.cpp`. Inside that file is the LLVM optimization pass for x86_64 CPUs that performs this ROP gadget removal rewriting.

Mortimer [6] claims a 5% reduction in unique gadgets in the OpenBSD kernel, with a 0.15% increase in kernel size and "negligible" runtime penalties.

These two mitigations are particularly exciting to study because they are trivial to implement, making them strong candidates for inclusion into upstream LLVM if they deliver the gadget reduction with minimal binary size and runtime effects claimed by Mortimer and the OpenBSD team. This is especially true as other software-based anti-ROP techniques such as shadow stacks have a noticeable overhead [8].

To port the register allocation reordering, all one has to do is modify just those two register allocation order lists: one for the 64-bit registers and one for the 32-bit registers. To port the instruction rewriting optimization pass for LLVM, one only needs to copy the `X86FixupGadgets.cpp` file to the appropriate location, add the new optimization pass to the list of LLVM x86_64 machine-dependent optimization passes, and modify the LLVM build files to include this new optimization pass. Once LLVM is rebuilt, the new compiler will perform these new anti-ROP techniques.

### *C. RETGUARD*

The OpenBSD project did implement one additional strategy for compiler-based anti-ROP mitigations called RETGUARD which serves as a replacement for GCC's stack-smashing protection (SSP) [9] that provides a per-stack frame stack canary and intentional `int3` instructions, which cause the binary to crash at runtime if executed, around every `ret` instruction. Prior to reaching a `ret` instruction, the stack frame's canary is checked and if untampered, jumps to the `ret` instruction; otherwise, control passes into the `int3` instructions, crashing the program [6].

As RETGUARD requires significant modification to all binaries on the system, we did not test RETGUARD in this paper, though other literature suggests that it may be effective against some types of gadgets [9]. Our contribution to the literature in this paper is to examine the register allocation reordering and compile-time instruction rewriting techniques for their efficacy, as trivial to implement with significant impact may yield outsized positive effects on a wide class of machines that are seeing their useful lifetimes expanded due to supply chain shortages.

## III. Methodology

We chose OpenBSD 7.8 and FreeBSD 14.3 as our target operating systems. This is because both were at the time the most recent releases of each operating system and because both operating systems used the same version of the LLVM compiler, version 19.1.7, as their in-base compiler suite. This would permit the easiest possible porting. We used a single machine, an HP t740 with 32 GB of RAM, as our testing machine.

We began by installing a vanilla copy of FreeBSD, installing the ROPgadget automatic ROP chain generation tool [10],

and using it to measure the number of unique gadgets in the FreeBSD kernel and libc. We chose ROPgadget because that is what Mortimer [6] chose for his investigation. We also collected binary data size of both the kernel and libc using the GNU size utility from the GNU binutils.

After that, we built and installed ten open source projects: `algol68g`, the GNU coreutils, GNU gawk, `libgmp`, GNU grep, `libarchive`, `libgcrypt`, OpenSSL, GNU sed, and SQLite3. These projects were selected because they all had easily measurable and comprehensive test suites. We collected the number of unique gadgets with ROPgadget, binary size data with GNU size, and runtime data by running the test suite of each program 50 times and using the GNU time utility to measure real, sys, and user timings. We named these datapoints "vanilla."

This process was then repeated three more times, with the addition of the mitigations. For these next three installations of FreeBSD, as soon as the operating system was installed, we modified the in-base LLVM compiler source code to incorporate the mitigations. We then fully rebuilt FreeBSD twice by following the FreeBSD `release(8)` manual page: running a complete set of `make buildworld && make buildkernel` followed by `make installworld && make installkernel`. We did this build process twice to ensure that all parts of the FreeBSD operating system contained the mitigation. After rebuilding the operating system twice, we then rebuilt and reinstalled the ten additional open source projects we tested.

We named our modified versions as follows:

- BX: Only the alternative register selection mitigation added.
- INSN: Only the compile-time instruction rewriting mitigation added.
- ALL: Both BX and INSN mitigations added.

Data was written out to CSV and plaintext files, and processed using Python scripts to generate raw output and graphs of the data. The scripts grouped the results by configuration (vanilla, BX, INSN, and ALL) and then compared these against the vanilla baseline. The absolute and percentage differences were computed, and paired statistical tests were applied to matched binaries. The scripts then generated bar charts, heatmaps, and summary plots to visualize the information gathered.

A multi-gigabyte open science repository containing all our raw data to encourage reproducability can be found at https://osf.io/dn73j/overview.

## IV. Results

We found that we were unable to achieve the same results claimed by OpenBSD. However, just because these mitigations did not deliver the same effects, it does not mean they should not be pursued by those looking to reduce their ROP gadget footprint.

### A. FreeBSD kernel and libc binary sizes and unique gadgets

Since the kernel runs with full system privileges and has responsibilities including memory management and hardware interaction, it is vital to reduce gadgets in this area. A successful kernel exploit would allow an attacker complete control of the system. The C standard library is a critical target as well due to the gadgets being present in libc being widely available across processes. If gadgets are reduced in libc, the security of the entire system would be strengthened.

For the FreeBSD kernel, there were approximately 875,000 unique gadgets for the vanilla build. Using the BX configuration, they were reduced by about 0.3%, and the kernel binary size increased by 0.5%.

The INSN configuration made a larger difference, reducing the unique gadget count by 3.6% (around 31,000 gadgets). There was a binary size increase of 1.8%.

When using the ALL configuration, the results of the kernel were not improved. The ALL configuration had a gadget reduction of 3.5%, or slightly less than the INSN configuration, combined with a 2.2% increase in binary size. This suggests that there is overlap between the two techniques.

The results were different in the case of libc. We found that the ALL configuration had the largest reduction, lowering the gadget count by around 2% and with a binary size increase of 1.7%. The INSN configuration had a reduction of 1.9% with a binary size increase of 1.3%, showing that instruction rewriting seemingly accounts for most of the difference observed. The BX mitigation alone applied to FreeBSD libc resulted in a reduction of only 0.6% with a binary size increase of 0.4%.

Fig. 1 shows the total number of gadgets in the FreeBSD kernel for each mitigation. Fig. 2 shows the percent reduction. Figs. 3 and 4 do the same, but for FreeBSD libc. For binary size differences, Fig. 5 provides a graph mapping gadget count reduction on the Y-axis and size of the text section of binaries on the X-axis.

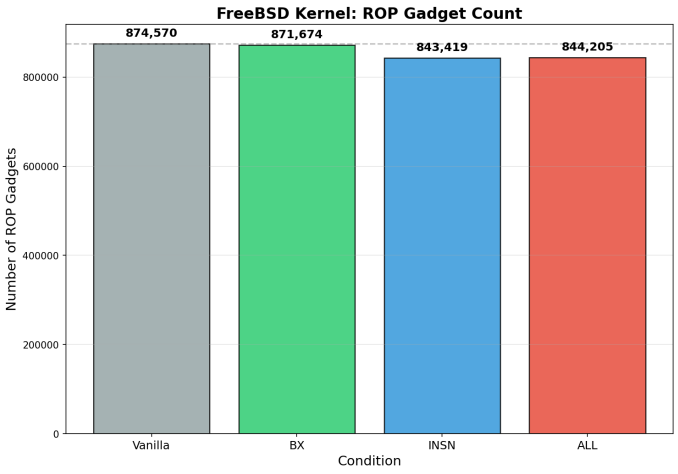


Fig. 1. Total number of unique gadgets in the FreeBSD kernel.

That is to say, we were unable to replicate the claims made by OpenBSD [6]: the BX mitigation did not reach 6%, the

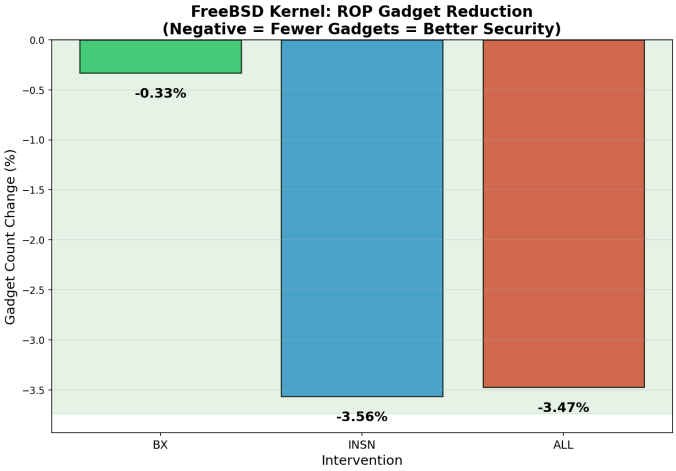


Fig. 2. Percent reduction in unique gadgets in the FreeBSD kernel.

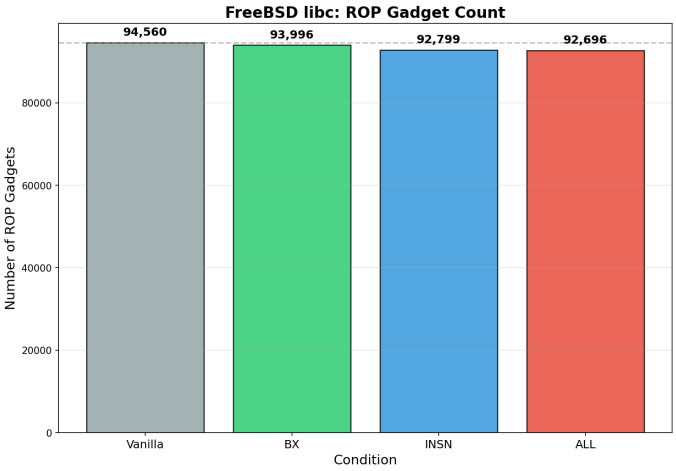


Fig. 3. Total number of unique gadgets in FreeBSD libc.

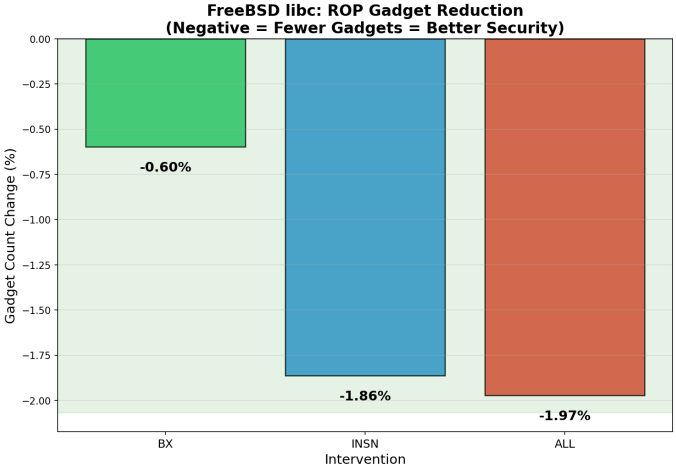


Fig. 4. Percent reduction in unique gadgets in FreeBSD libc.

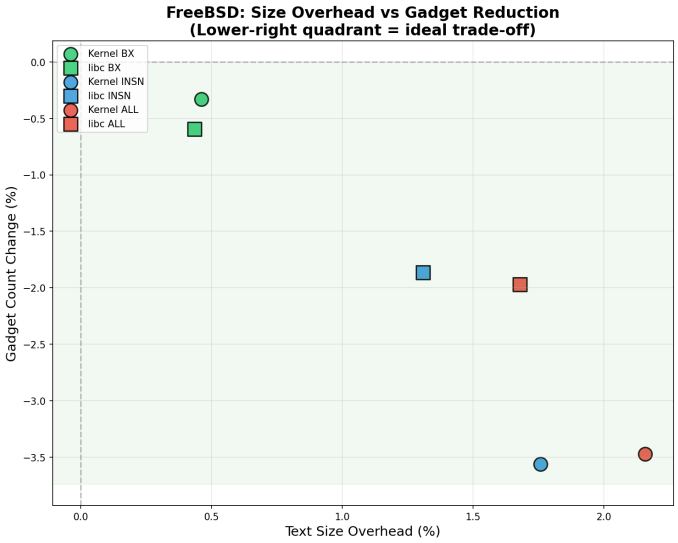


Fig. 5. Size overhead vs. gadget reduction tradeoff map.

INSN mitigation did not reach 5%. In fact, even with both mitigations together, we do not see the kinds of reductions that OpenBSD claims even one of these mitigations should provide. Perhaps paradoxically, for the FreeBSD kernel, combining both mitigations results in a less effective unique gadget reduction strategy compared to just the INSN mitigation by itself: 3.56% reduction with just INSN compared to 3.47% reduction with both enabled.

But that is not to say the techniques are without merit; it is more likely the case that these mitigations need to be combined with additional anti-ROP techniques to be maximally effective.

Unfortunately, ROPgadget had no difficulty generating ROP chains with the kernel and libc no matter what combination of mitigations were turned on. That is to say, we failed to validate any practical exploit resistance. These two mitigations alone, whether deployed separately or together, do not bring any advantage over not having these mitigations at all. This may be because the mitigations are very modest in reduction of gadget quantity and may be even more modest in reduction of gadget quality. This means that practically, they may only be useful when paired with other anti-ROP techniques.

### *B. Userland runtime performance, binary sizes, and unique gadgets*

For runtime testing, a Python script was created to perform the standard deviation calculations, paired t-tests, and plots. Each project generated a CSV file containing runtime test data from 50 runs of that project's test suite using a shell script and the GNU time utility. The Python script then calculated the standard deviation from the data in each CSV file using Pandas, then performed the calculations for the paired t-tests, which compares to vanilla. The Python script then used these results to create spaghetti plots, box plots, and standard error of the mean graphs with real, sys, and user timings.

For most of the open source test suites, runtime tests were in a small range of each other, suggesting a minimal runtime penalty. Confusingly, there were some tests that were significantly faster with mitigations turned on compared to the vanilla runtime values, such as OpenSSL. This suggests there may be something else affecting runtime speeds, and not something that the mitigations would have introduced.

Fig. 6 provides two heat maps detailing absolute and percentage differences for each mitigation compared to vanilla.

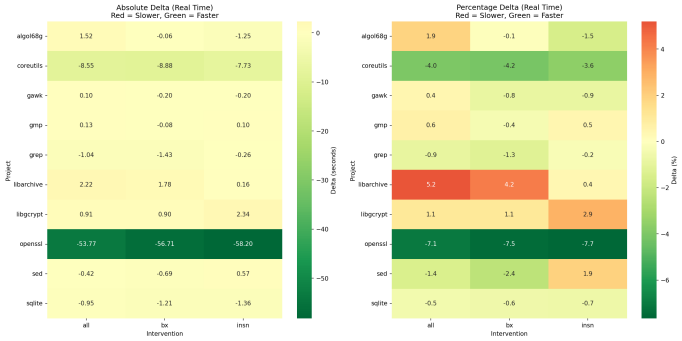


Fig. 6. Absolute and percentage real time differences.

Across the ten userland applications, binary size increases were consistent. The BX mitigation increased the binary size by approximately 0.4%, INSN by 1.5%, and ALL by 1.8%. The trends across userland match those seen in kernel and libc.

The results show that these mitigations are mostly inexpensive, but not free. The increase in binary size is typically only a few percent. The BX configuration has a small reduction of gadgets with barely any cost, while the INSN configuration gives a larger reduction with a larger cost and overhead.

Paradoxically, there were some instances where unique gadget numbers increased. Fig. 7 shows a heat map of binary size and unique gadget changes for the open source projects. Two projects, the GNU coreutils and `libgcrypt`, both saw an increase in unique gadgets as reported by ROPgadget; this may require additional investigation to determine why this happens.

Previous work by Mortimer [6] did not analyze the quality of eliminated gadgets. Quality of gadgets is a better metric than quantity of gadgets; it takes only a very small handful, fewer than 10, gadgets to successfully launch an attack. Furthermore, many gadgets are extremely difficult if not impossible to use in a successful attack. One can imagine the removal of the `nop; ret` gadget: while this will contribute to quantity of removed gadgets, it does not contribute to the quality of removed gadgets.

Because Mortimer did not study the reduction of gadget quality, only gadget quantity, we have no method for comparison. The closest comparison from Mortimer was what he called "gadget density," or number of unique gadgets per kilobyte. As this metric also does not focus on quality, we did not find it useful for comparison. Future work will examine the reduction of gadget quality.

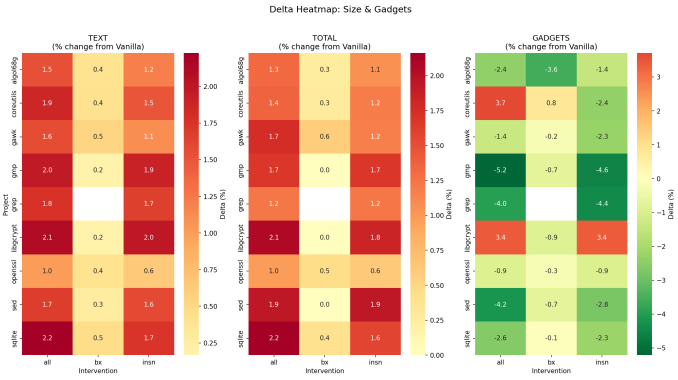


Fig. 7. Heatmap of binary size and unique gadget changes.

## V. Conclusion

This exploration demonstrates that these two software-based anti-ROP mitigations provided by the OpenBSD project provide a modest reduction in ROP gadgets. We were unable to replicate the claimed effectiveness by the OpenBSD team, but we were able to corroborate the idea that the size overhead penalty is at best quite modest: at most a little over 2%.

However, the goal of these mitigations is not necessarily to reduce the number of unique gadgets to absolute zero; it is sufficient to increase the difficulty of successfully exploiting ROP gadgets to the point where an attacker gives up and move on.

As these mitigations do not appear to meaningfully reduce gadget quality, demonstrated by the fact that the ROPgadget tool had no difficulty in producing ROP chains for the FreeBSD kernel and FreeBSD libc, one must keep in mind that these mitigations may provide a modest benefit only when deployed as part of a much larger gadget reduction system.

We also discovered that these two mitigations have some amount of overlap or interplay in the LLVM pipeline. Unlike the original paper by Mortimer that implied these two mitigations were additive, we found that they were never perfectly additive. Indeed, we even found an example of a situation where turning on both mitigations was *worse* than only turning on one of the mitigations. This points to these mitigations needing to be judged on a case-by-case basis and not used as a blanket always-on approach.

Compiler-based anti-ROP techniques that initially appear obvious may in fact be "too good to be true." Indeed, even compiler-based mitigation techniques focused on software diversity which removed between 90% and 99% of unique gadgets still found that the remaining gadgets were sufficient to launch a successful attack [11].

While our independent analysis of these OpenBSD anti-ROP mitigations did not fully corroborate OpenBSD's claims, we still hope our analysis inspires others to continue research into compiler-based and software-based anti-ROP techniques. As there continue to be computers in service that do not have the latest hardware mitigations, software mitigations remain an important line of defense.

### A. Future work

We would like to replicate this work with the GNU Compiler Collection (GCC). As GCC is the primary compiler for the Linux kernel and many of the GNU projects such as glibc, bringing these mitigations to GCC in addition to measuring quality of gadgets reduced would bring a more complete understanding of these particular anti-ROP techniques and their limitations.

## REFERENCES


[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012, https://doi.org/10.1145/2133375.2133377.

[2] T. Zhang, M. Cai, D. Zhang, and H. Huang, "Sebrop: blind rop attacks without returns," *Frontiers of Computer Science*, vol. 16, no. 4, p. 164818, 2022, https://doi.org/10.1007/s11704-021-0342-8.

[3] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, Apr. 2017, https://doi.org/10.1145/3054924.

[4] H. Etoh and K. Yoda, "Propolice: Protecting from stack-smashing attacks," *Technical Report, IBM Research Division, Tokyo Research Laboratory*, 2000, https://dominoweb.draco.res.ibm.com/reports/rt0371.pdf.

[5] J. A. Marpaung, M. Sain, and H.-J. Lee, "Survey on malware evasion techniques: State of the art and challenges," in *2012 14th International Conference on Advanced Communication Technology (ICACT)*, 2012, pp. 744–749, https://ieeexplore.ieee.org/document/6174775.

[6] T. Mortimer, "Removing rop gadgets from openbsd," in *Proceedings of AsiaBSDCon 2019*, 2019, pp. 13–21, https://doi.asiabsdcon.org/10.25263/asiabsdcon2019/p01b.

[7] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, 2007, pp. 23–34, https://doi.org/10.1109/ISPASS.2007.363733.

[8] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 555–566, https://doi.org/10.1145/2714576.2714635.

[9] G.-A. Jaloyan, "A semantic approach to machine-level software security," Theses, Université Paris sciences et lettres, Sep. 2021, https://theses.hal.science/tel-04023324.

[10] J. Salwan, "Jonathansalwan/ropgadget: This tool lets you search your gadgets on your binaries to facilitate your rop exploitation. ropgadget supports elf, pe and mach-o format on x86, x64, arm, arm64, powerpc, sparc, mips, risc-v 64, and risc-v compressed architectures." [Online]. Available: https://github.com/JonathanSalwan/ROPgadget.

[11] J. Coffman, D. M. Kelly, C. C. Wellons, and A. S. Gearhart, "Rop gadget prevalence and survival under compiler-based binary diversification schemes," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. SPRO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 15–26, https://doi.org/10.1145/2995306.2995309.