Is Cleaning Costly? Evaluating the `-fret-clean` Anti-Return-Oriented Programming Mitigation from the OpenBSD Operating System

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

Maryam Q. Shaikh
Dept. of Professional Counseling
Monmouth University
West Long Branch, NJ, USA
0009-0004-6970-1179

***Abstract*—Software-based return-oriented programming (ROP) defenses represent one potential aspect of a full suite of anti-ROP mitigations, often considered part of a larger defense-in-depth strategy.**

This paper provides an empirical evaluation of the OpenBSD `-fret-clean` mitigation, which at compile time inserts code that zeros out the callee's return addresses on the stack immediately upon returning to the caller, preventing those addresses from being used to discover useful ROP gadget locations by attackers. This evaluation fills a gap in understanding this particular mitigation, which was placed into production without cost measurements.

We evaluate the mitigation by porting it to FreeBSD and applying it to the FreeBSD kernel, libc, libcrypto, and dynamic linker, matching OpenBSD. We benchmark the system before and after and discover this mitigation produces stable and measurable impacts: ELF `.text` size increases of approximately 8% and runtime performance penalties up to 27.27% in the pathological case, down to small but statistically significant performance penalties depending on workload.

The mitigation is excellent at eliminating stale return addresses from the stack in a synthetic test case. The mitigation does not interact with other libc pointers on the stack, which made up about 80% of total libc pointers on the stack in our synthetic test.

Our evaluation demonstrates that this mitigation has a relatively consistent and modest upper bound for costs, but may not provide more than a marginal security improvement, even when combined with other mitigations. We draw attention to the mismatch between local success and global success.

***Index Terms*—cybersecurity, return-oriented programming, OpenBSD, operating systems, compilers**

Preprint: To appear in *Proc. 6th IEEE Cyber Awareness Res. Symp. (CARS)*, Grand Forks, ND, USA, Oct. 2026.

I. Introduction

Return-oriented programming (ROP) is a serious class of attack. Though first explicated around fifteen years ago [1], we continue to see a proliferation of research on ROP attacks and defenses [2, 3, 4]. As ROP shows no signs of disappearing in attacker usage or academic interest, it is critically important to evaluate proposed mitigations making their way into modern operating systems.

In this paper, we examine an anti-ROP mitigation proposed by the OpenBSD operating system in which they extended their custom version of the LLVM compiler suite with a new `-fret-clean` option flag to zero out stale return addresses on the stack. The primary contribution of this paper is to provide an empirical evaluation of this mitigation technique, as no such evaluation currently exists and the mitigation was put into production in October 2024 starting in OpenBSD 7.6 [5]. Our motivation comes from previous work in the literature that discovered other OpenBSD-developed anti-ROP mitigations did not perform as claimed [6, 7].

We trace the origins of the `-fret-clean` mitigation, noting both related-but-different prior art and an explicit case of exact prior art predating the OpenBSD mitigation by two decades. We then explicate what the mitigation technique does to code it compiles.

Next, we empirically test the mitigation for both size and performance tradeoffs. First, we port the mitigation to the FreeBSD operating system, applying the mitigation to the kernel, libc, libcrypto, and ld.so objects, matching OpenBSD's use of the mitigation. We measure performance tradeoffs through the use of self-developed microbenchmarks designed to exercise worst-case performance scenarios. We then run a series of tests from the Phoronix Test Suite to gather metrics from real-world programs. Finally, we perform a FreeBSD kernel build to provide a real-world workload. In all runtime test cases, we collect runtime data before applying the mitigation and again after applying the mitigation. Finally, we measure size increases by collecting ELF `.text` size information from the `size(1)` utility for the FreeBSD kernel, libc, libcrypto, and ld.so objects before and after applying the mitigation.

We find that this mitigation is surgically precise, virtually eliminating its target of stale return addresses on the stack. However, that surgical precision represents only a minority, a little under 20% of libc addresses found on the stack in our test program; an attacker needs only one libc address for

their purposes. Additionally, while the mitigation did reduce the number of libc memory pages discovered by about 40%, that still left dozens of pages discoverable, likely enough to construct a decent mapping of libc.

Our results present a similar story to previous independent evaluations of other OpenBSD-developed compiler-based anti-ROP mitigations in the literature [6, 7]. These tradeoffs, combined with minimal actual utility in preventing attacks, even when used as part of a larger defense-in-depth strategy, suggests that this mitigation may not be worth its costs.

Finally, we ground our interpretation of the results with previous literature in psychology, management accounting, and organizational behavior through the phenomenon of surrogation [8], the idea that proxy metrics get confused for what you really want to measure, and the potential negative downstream effects of surrogation when applied to mitigation development.

It is our hope that this paper continues to inspire further exploration into and independent evaluation of software- and compiler-based anti-ROP mitigation techniques.

II. Background

In May 2024, the OpenBSD project presented a new mitigation to defend against ROP attacks. Though the mitigation was originally proposed as a demo or proposal [9], by October 2024 it was shipped in production as part of OpenBSD 7.6 [5], and continues to ship in all versions of OpenBSD to the present day.

This mitigation, solely for x86_64 CPUs, teaches their custom version of the LLVM compiler suite to insert `movq $0, -8(%rsp)` instructions immediately following `call` instructions [9]; this instruction yields a 9-byte object code sequence when compiled with LLVM 19.1.7, the version of LLVM that ships with both FreeBSD 15.0-RELEASE and OpenBSD 7.8: `48 c7 44 24 f8 00 00 00 00`.

The mitigation is selectable by a default-off flag for the OpenBSD custom clang compiler: `-fret-clean` [9].

By passing this flag to the clang compiler, subroutines will remove stale return addresses from the stack once they are no longer required. It does so by inserting code to have the caller immediately zero out the callee's return address location on the stack upon returning to the caller. This mitigation therefore removes trivially recognizable targets for finding gadgets; OpenBSD applies this technique specifically to high-value objects such as the kernel and C standard library as those objects have a wide variety of gadgets [10], more than enough to launch a successful attack [1, 11].

The technique of OpenBSD teaching anti-ROP mitigations to their compiler is a signature; previous work in the literature shows that OpenBSD has implemented a number of anti-ROP mitigation techniques to their custom LLVM compiler [12]. However, other authors have demonstrated that these techniques fall short of OpenBSD's original claims, showing that OpenBSD's anti-ROP mitigations of custom register allocation, specifically moving the RBX register to the end of the LLVM register allocator list, and compile-time instruction rewriting to remove potential polymorphic gadget instructions both failed to remove the number of gadgets originally claimed and had binary size increases and performance penalties that originally went unreported [6, 7]. Other work shows that RETGUARD, OpenBSD's improved stack cookie system [12], is only moderately successful at preventing attacks [13].

Similar techniques have been proposed in the literature and been put into production: we note a paper that proposes and evaluates tracking and cleaning used scratch registers immediately before return instructions [14]. Follow-up work took the core idea and moved the dynamic runtime method in the original paper into an ahead-of-time binary rewriter [15]. This idea behind the technique saw adoption as the `-fzero-call-used-regs` family of flags implemented in GCC [16] and LLVM [17]. This was later incorporated as the `CONFIG_ZERO_CALL_USED_REGS` option in the Linux kernel [18]; independent testing found a nuanced runtime performance penalty of sometimes none, sometimes 2-3%, and up to 11% for some workloads [19], greater than the about 1% originally claimed [18]. Furthermore, the maintainers of the Linux hardened kernel recommend against the `CONFIG_ZERO_CALL_USED_REGS` option [20], citing concerns the option has considerable performance impacts [21].

As far as we can tell, OpenBSD does not use any of the `-fzero-call-used-regs` flags to build their operating system.

Interestingly, the idea of zeroing out a callee's return address on the stack immediately upon returning to the caller was not entirely novel at the time OpenBSD introduced it; OpenBSD appears to have been at least the third independent invention of this mechanism.

A 2018 paper about modifying the LLVM compiler to control side effects lists stack and register erasure as one of the techniques to control side effects. While this paper does not reference "ROP" or "return-oriented programming" at all, it nonetheless offers a mechanism not dissimilar to `-fret-clean`. Their proposed `__zero_on_return` keyword zeroes the stack and registers used upon return in order to control side effects. This technique zeroes out the entirety of the 128-byte "red zone" specified in the x86-64 System V ABI, which includes the return address [22].

A patent filed by IBM in February 2000 and ultimately granted in April 2003 describes the exact mechanism that OpenBSD would implement with its `-fret-clean` flag; in the patent's procedures, it states: "scan code for 'call' operations, and for each 'call' operation, inject code following the 'call' operation which re-initializes the stack space used to pass arguments to and receive returned arguments from the called routine, and to re-initialized the stack memory used to store the return address used by the called routine to return to the calling routine" [23].

This is a perfect summary of the OpenBSD `-fret-clean` mitigation, and one of the concerns the patent hoped to cover was security. Though it should be noted that while OpenBSD specifically mentioned ROP as its concern [9], the IBM patent mentions the leaking of sensitive data and misleading debug-

ging as its primary concerns and never mentions ROP [23], likely because ROP did not exist as a named concept in 2003.

It is instructive to note that neither the IBM patent nor `__zero_on_return` made their way into upstream compilers.

Taken all together, we see well-reasoned, even "intuitive," anti-ROP mitigations [10, 14, 15, 17, 18] be found to be less than originally advertised after independent evaluation [6, 7, 19, 20, 21], or be proposed only to not make their way into mainstream compilers [22, 23]. We extend this tradition of independent evaluation of anti-ROP techniques by evaluating `-fret-clean`.

III. Methodology

We provide a detailed methodology, as we believe one of the important contributions of this paper is a replicable testing method for others to test this mitigation for themselves, as well as to provide a model for others to develop their own testing methodologies for other mitigations that have gone undertested.

Our workstation is an HP t740 with an AMD Ryzen V1756B CPU, 32 GB of DDR4-3200 RAM, and a 1 TB NVMe solid-state drive. We began by installing FreeBSD 15.0-RELEASE onto the machine; this was the most recent version of the FreeBSD operating system at the time.

Prior to running any tests, we used the FreeBSD `size(1)` utility to get the binary sizes of the kernel, libc, libcrypto, and ld.so objects. From there, we extracted the `.text` size specifically, as because the `-fret-clean` mitigation adds instructions into the binaries, this lets us make a true cross-object comparison.

Once binary sizes were collected, we built a series of microbenchmarks to understand the worst-case scenario for performance impacts: two of our microbenchmarks test syscalls, `getpid(2)` and `clock_gettime(2)`, with the other two testing libc functions, `strlen(3)` and `printf(3)`. Our benchmark runs a tight loop of 100,000,000 calls per function. For completeness, we also created microbenchmarks that made small adjustments: partially unrolling the loop, nesting the libc function or syscall in a wrapper function, and both together. For all microbenchmarks, we collected both wall-clock time with the GNU time utility and instructions retired with the `pmcstat(8)` utility. Each microbenchmark was run fifty times, then averaged to produce our results.

Next, we installed the Phoronix Test Suite program from the FreeBSD package repository with the command `sudo pkg install phoronix-test-suite-php83`. After installation, we set the environment variable `FORCE_TIMES_TO_RUN` to fifty with the command `export FORCE_TIMES_TO_RUN=50`; this environment variable overrides the default settings for the number of iterations of each test to run. We found the default iterations were often unacceptably low, usually between three and five iterations, not nearly enough to provide statistically significant results. We averaged the results across the fifty runs to produce our results.

We then ran the tests for `compress-gzip` and `unpack-firefox` by running the command `phoronix-test-suite benchmark <testname>`. We chose all the default options for each test when prompted. When the test suite asked us to name the results, we named them `<testname>-pre` to signify that these tests were performed before applying the `-fret-clean` mitigation; after the mitigation was applied, we named our results `<testname>-post` to differentiate and make clear which results were which.

After finishing the Phoronix Test Suite, we built the FreeBSD kernel by installing the FreeBSD source tree, becoming the superuser, and running the command `cd /usr/src && make -j1 buildkernel`. The `make buildkernel` command provides timing details for the build upon its conclusion.

We then ran one last test: a test that intentionally dirties a stack and then scans forwards and backwards from the stack pointer to find libc pointers. The program determines at runtime libc start and ending addresses as well as the start and end addresses of the `.text` section of libc. The program then calls `snprintf`, `strlen`, `clock_gettime`, and `getpid` 100,000 times in a tight loop. After returning from the subroutine containing the loop, the program scans forwards and backwards from the current stack pointer up to 64 KB of total stack space; using heuristics, it collects pointer information about what appears to be a return address location, what appears to be a libc pointer into `.text` but not a return address location, and libc pointers into other sections such as `.data`, `.rodata` and `.bss`. Finally, the test collected unique libc pages found while scanning the stack. We ran this test 50 times to match the other tests.

Next, in order to successfully add the mitigation, we applied only the `llvm.patch` file from our open science repository. This patch adds the `-fret-clean` flags and logic to LLVM but does not apply it to the kernel or any libraries. After applying the `llvm.patch` file we rebuilt all of FreeBSD with the command `make -j8 buildworld buildkernel` and then installed the new binaries with `make installworld installkernel`. This gave us a copy of FreeBSD with only LLVM modified. We then rebooted the system.

Now we can apply the `makefiles.patch` patch from our open science repository. This patch applies the `-fret-clean` flag to the FreeBSD kernel, libc, libcrypto, and ld.so `Makefile`s. We then rebuilt and reinstalled all of FreeBSD; this ensures that all target objects were rebuilt with the mitigation applied.

Finally, we re-ran all the binary size collections, runtime tests, and the stack artifacts test with the exact same commands.

Our testing apparatuses and results can be found at our open science repository at https://osf.io/k6v2j/overview.

IV. Results

We used RStudio for all data processing. Measurements are rounded to two decimal places, unless less than 0.01 after

rounding; in such cases measurements are rounded to three decimal places. All time measurements are in seconds. For all tables, ± reports standard deviation.

In all cases, we observe a substantial increase in `.text` sizes, measured in bytes. Table I lists these increases, with Fig. 1 representing them visually.

TABLE I
ELF `.text` SIZE INCREASES

Object	Before	After	Change
Kernel	20497658	22271706	8.65%
ld.so	113317	123397	8.90%
libc	1992345	2146313	7.73%
libcrypto	5790012	6261908	8.15%

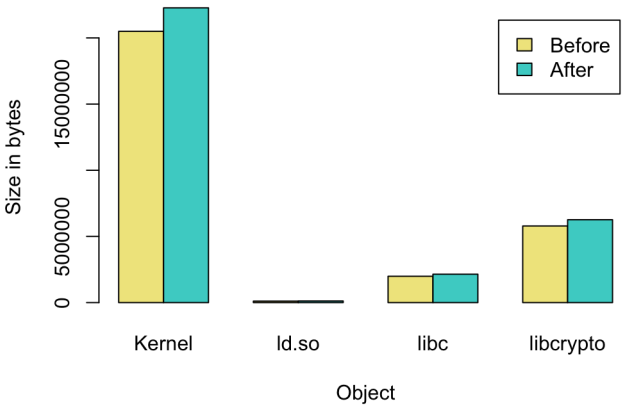

Fig. 1. Difference in binary sizes before and after applying the `-fret-clean` mitigation.

We also observe a runtime performance penalty up to 27.27% in the pathological case.

Table II lists the runtime performance penalties for the microbenchmarks. The symbol (U) represents the partially unrolled variant, (N) the nested variant, and (UN) the combined unrolled and nested variant.

Table III presents the same tests, but with instructions retired as collected by `pmcstat(8)`. Numbers were divided by 100,000,000 to provide approximate instructions retired per iteration.

Table IV lists runtime penalties for the `compress-gzip` test suite from Phoronix.

Table V lists runtime penalties for the `unpack-firefox` test suite from Phoronix.

Fig. 2 visualizes the differences as a boxplot.

As a sanity check, we ran a Mann-Whitney U test on the `unpack-firefox` test, obtaining a p-value well below $p < 0.0001$, with an effect size estimate of 0.860 and large magnitude reported by the `wilcox_effsize` function from the `rstatix` package. This statistical test was chosen

TABLE II
MICROBENCHMARK PERFORMANCE (WALL-CLOCK TIME)

Test	Before	After	Change
clock_gettime	4.01 ± 0.02	3.98 ± 0.01	-0.75%
clock_gettime (U)	3.85 ± 0.01	3.83 ± 0.09	-0.52%
clock_gettime (N)	4.88 ± 0.03	4.82 ± 0.01	-1.23%
clock_gettime (UN)	4.84 ± 0.01	4.79 ± 0.01	-1.03%
getpid	6.93 ± 0.01	7.64 ± 0.39	10.25%
getpid (U)	7.11 ± 0.28	7.61 ± 0.39	7.03%
getpid (N)	7.24 ± 0.35	7.62 ± 0.39	5.25%
getpid (UN)	7.22 ± 0.24	7.63 ± 0.40	5.68%
printf	5.72 ± 0.06	5.83 ± 0.03	1.92%
printf (U)	5.73 ± 0.10	5.84 ± 0.03	1.92%
printf (N)	5.86 ± 0.19	5.91 ± 0.10	0.85%
printf (UN)	5.78 ± 0.04	5.91 ± 0.03	2.25%
strlen	0.22 ± 0.004	0.28 ± 0.002	27.27%
strlen (U)	0.21 ± 0.002	0.25 ± 0.01	19.05%
strlen (N)	0.27 ± 0.002	0.34 ± 0.002	25.93%
strlen (UN)	0.24 ± 0.002	0.29 ± 0.002	20.83%

TABLE III
MICROBENCHMARK PERFORMANCE (INSTRUCTIONS RETIRED)

Test	Before	After	Change
clock_gettime	180.06 ± 0.000	184.06 ± 0.000	2.22%
clock_gettime (U)	178.44 ± 0.000	182.44 ± 0.001	2.24%
clock_gettime (N)	204.07 ± 0.000	208.07 ± 0.000	1.96%
clock_gettime (UN)	202.82 ± 0.000	206.82 ± 0.000	1.97%
getpid	263.09 ± 0.002	267.10 ± 0.004	1.52%
getpid (U)	261.46 ± 0.003	265.47 ± 0.004	1.53%
getpid (N)	268.09 ± 0.003	272.10 ± 0.004	1.50%
getpid (UN)	266.47 ± 0.002	270.47 ± 0.004	1.50%
printf	431.08 ± 0.001	436.08 ± 0.000	1.16%
printf (U)	429.45 ± 0.001	434.45 ± 0.000	1.16%
printf (N)	436.08 ± 0.002	441.08 ± 0.001	1.15%
printf (UN)	434.45 ± 0.000	439.46 ± 0.000	1.15%
strlen	19.022 ± 0.000	19.023 ± 0.000	0.01%
strlen (U)	17.397 ± 0.000	17.398 ± 0.000	0.01%
strlen (N)	23.023 ± 0.000	23.024 ± 0.000	0.004%
strlen (UN)	21.397 ± 0.000	21.398 ± 0.000	0.01%

because there is no inherent pairing between the 50 pre-mitigation runs and the 50 post-mitigation runs, thus making it a comparison of a pre-mitigation control group against a post-mitigation intervention group, testing if the runtime in the intervention group is stochastically larger than the control group. The Mann-Whitney U test was designed for such comparisons [24].

A Welch two sample t-test corroborates the findings with a p-value also well below $p < 0.0001$.

Table VI shows the runtime penalties for a FreeBSD kernel build by running the `make -j1 buildkernel` command.

TABLE IV
COMPRESS-GZIP RUNTIME PERFORMANCE PENALTIES

Before	After	Change
45.20 ± 0.75	45.35 ± 0.76	0.33%

TABLE V
UNPACK-FIREFOX RUNTIME PERFORMANCE PENALTIES

Before	After	Change
26.68 ± 0.10	27.21 ± 0.14	1.99%

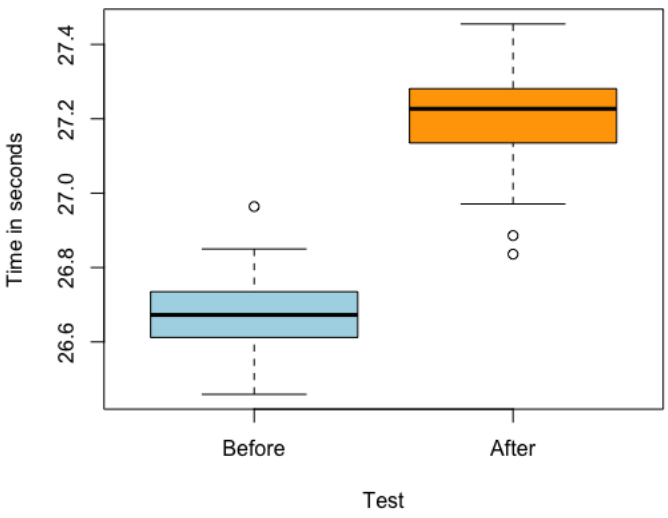

Fig. 2. Boxplot showing the differences in pre- and post-mitigation runtime performance for the Phoronix `unpack-firefox` test.

TABLE VI
FREEBSD KERNEL BUILD TIMES

Before	After	Change
3041	3051	0.33%

Finally, Table VII presents a benchmark where a stack was created, dirtied, and then scanned for libc pointers; the program organizes the libc pointers it finds into three categories: those that appear to be in return address slots on the stack, those that do not appear to be in return address slots but do point into the libc `.text` section, and those that point into other sections of libc, such as `.data`, `.rodata`, and `.bss`. The top row of Table VII is the test before the `-fret-clean` mitigation was applied; the bottom row is the test after application.

TABLE VII
LIBC ADDRESSES ON THE STACK BEFORE AND AFTER MITIGATION

Total	`.text`	Return address	Other `.text`	`.(ro)data/.bss`
110	32	20	12	78
92	14	2	12	78

Fig. 3 presents this information visually. We notice that only the "Return address" group changes; "other `.text`" and "in `.data/.rodata/.bss`" remain the same.

Furthermore, this benchmark found 40 unique libc pages before the mitigation was applied and 24 unique libc pages after mitigation. We visualize the difference in Fig. 4.

We note that while we did run this test 50 times both before and after applying the mitigation, we got the exact same numbers each time. That is to say, all 50 before tests produced 110/32/20/12/78 and all 50 after tests produced 92/14/2/12/78; this test is entirely deterministic.

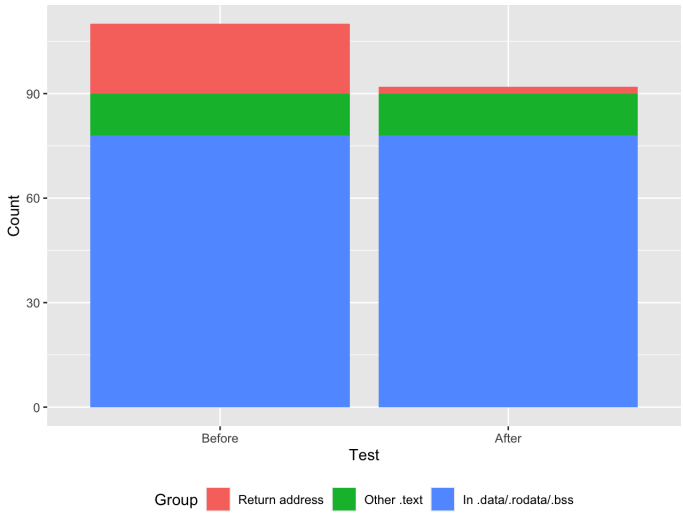

Fig. 3. Stacked barplot showing only return addresses are affected by the `-fret-clean` mitigation.

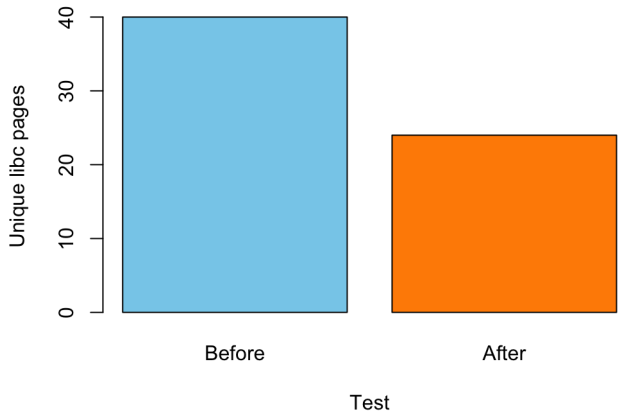

Fig. 4. Difference in unique libc pages found by the stack scanning tool before and after applying the `-fret-clean` mitigation.

V. DISCUSSION

Even though the original OpenBSD introduction of the mitigation did not make any direct mention of tradeoffs, a cursory reading of the original patch uncovers an approximately 4.8% increase in the OpenBSD installer's `FSSIZE`, from 1,359,872 to 1,425,408 [9], implying the size tradeoff was already known at the time the mitigation was originally shared but not forefronted. Our own empirical analysis suggests that this size increase was not a one-off but a core tradeoff of the overall mitigation; additionally, the size increase is remarkably stable as indicated by Table I, suggesting that the number of calls is similar across large and disparate code bases. Indeed,

our results demonstrate a consistent `.text` size increase of around 8%.

Runtime penalties exhibit a much more nuanced story: as the original OpenBSD introduction made no mention of performance penalties, we believe this represents the first evaluation of the `-fret-clean` mitigation as to its performance penalties. In our real-world benchmarks, we see changes from negligible in `compress-gzip` and kernel rebuilding to about 2% in `unpack-firefox`. Additionally, we found statistical significance in the `unpack-firefox` test, implying that we should expect modest performance impacts in other workloads.

These performance impacts make sense, as you are only adding one additional instruction per call in instrumented objects only. The tests themselves were not compiled with `-fret-clean`, only the kernel, libc, libcrypto, and ld.so were, matching OpenBSD. If you never cross those boundaries, you will never see performance impacts.

However, our synthetic microbenchmarks demonstrate that there is potential for significant workload slowdowns: up to 27.27% in the pathological case, the baseline `strlen` case. Syscalls such as `getpid` and `clock_gettime` and libc functions like `strlen` and `printf` are called frequently in real software; any slowdown in such functions will accumulate—and quickly.

There is another connection worth mentioning: slowdowns might be explained by the added instructions causing instruction cache misses or other interactions with microarchitectural realities; that is to say, the 8% ELF `.text` section increases may produce unpredictable runtime performance swings: this would explain how `strlen`, a function written in assembly on FreeBSD and thus unaffected by the mitigation, performed so much worse despite not having any real increase in instructions retired. It would also help explain the slight improvement in `clock_gettime`.

If this is the case, these layout changes might be an even worse cost than the increased instruction count: instruction count increases can be well-understood ahead of time and budgeted for; layout changes are entirely unpredictable and may be dependent on particular microarchitectures, and cannot be reasoned about prior to deployment. Future work should explore those relationships more thoroughly.

With that said, in a real sense, our runtime testing vindicates OpenBSD's decision to apply the mitigation only to high-value targets, and we commend OpenBSD for its skill in design taste.

However, we believe the mitigation itself is of extremely limited utility. So much so, that we consider recommending against other operating systems and upstream LLVM adopting this mitigation. Table VII shows unequivocally that the mitigation does exactly what it sets out to do: it removes return addresses on the stack. We see a 90% reduction, from 20 to 2. We note that the mitigation does not apply to functions marked with the `returns_twice` attribute; for our purposes we grant that the mitigation is perfect.

Unfortunately, this excellence in its small task amounts to only about 16% of all libc pointers on the stack in our benchmark. Even when discounting the libc pointers found that point into the `.data`, `.rodata`, and `.bss` sections, that is still only about a 56% reduction in libc pointers. While a 56% reduction in the `.text` section might sound significant, we must remember that an attacker often needs only one libc pointer to find gadgets, and it often does not matter if that pointer comes from the `.text` section or any other section.

Similarly, while unique page discovery did fall about 40%, from 40 unique pages to 24 unique pages, that is still likely sufficient for an attacker to reconstruct a workable map of libc for gadget chaining. OpenBSD does have a mitigation, first appearing in OpenBSD 6.0 released in 2016, where at boot time libc is relinked in a random order [25]. Future work should evaluate how much protection this mitigation provides in the event of unique page discovery.

In OpenBSD's defense, the original introduction explicitly states that it does not target other address locations [9]. However, all that is needed is a single pointer leak for an attacker to be successful [26]; this remains true even with fine-grained proposed mitigations such as instruction manipulation [27]. Without additional clearing of all other libc pointers, likely impossible because some number of pointers will need to be live at any given point in program execution, there may be no tradeoff—binary size or performance—that justifies a result of no improved security in practice that is likely for the `-fret-clean` mitigation. Indeed, it is exactly these concerns that led to the recommendation that Linux's `CONFIG_ZERO_CALL_USED_REGS` option not be used for hardened kernels [20, 21].

We feel compelled to agree with that recommendation for `-fret-clean` and note that this recommendation follows other work in the literature that specifically evaluated other compiler-based OpenBSD anti-ROP mitigations and discovered that while they are precise in eliminating their small niche of potential issues, they all had greater binary size and performance penalties than originally claimed, and did not provide any meaningful additional security due to a mismatch between what the mitigation did and what attackers need to be successful [6, 7].

It is this mismatch we want to highlight: for anti-ROP mitigations, OpenBSD appears to continually conflate local success for global success, and as a result design and implement mitigations that do not map to attacker capabilities. We believe this leads to a false sense of security, and may be worse than no security at all.

Indeed, we are far from the first to note that "obvious" or "intuitive" mitigations for ROP may not map to attacker capabilities. For example, previous work repeatedly points out how ROP gadget reduction does not result in greater security, as gadget reduction may paradoxically make security worse by introducing gadgets that are more useful for attackers [28] and even with 90-99% gadget reduction there is still enough gadget variety for attackers to be successful [29].

We emphasize, paraphrasing from previous work in the literature [6, 7], that "obvious" or "intuitive" mitigations are the ones that require the most empirical evaluation before

deployment; we believe the `-fret-clean` mitigation is little more than an instructive example of this necessity.

VI. Conclusion

We conclude by recapping the primary motivations and results of this paper: a first empirical evaluation of `-fret-clean`, finding about 8% growth in ELF `.text` sizes, up to a pathological 27.27% runtime penalty with modest but statistically significant runtime penalties for real-world workloads, and a 90% reduction in return addresses found on the stack which translates only to about 16% of all libc pointers. Put together, the mitigation is locally successful but may not be globally successful.

To make this point, we step back and comment on the pattern we have noticed across now two previous papers [6, 7] and this one: we have labeled it the difference between local success and global success, and confusing the former for the latter. This is an instance of Goodhart's Law [30], popularly quoted via Strathern's distillation that "when a measure becomes a target, it ceases to be a good measure" [31].

From management accounting, we have the concept of surrogation [8], the idea that people mistake a proxy metric for what they really want to measure. We see that idea over the course of our work: compiler-based anti-ROP mitigations measure gadget reductions or pointer reductions as if they were measurements for whether or not attackers are meaningfully constrained. The idea of surrogation builds on the psychological concept of "attribute substitution," the idea that when people are faced with a difficult question they tend to answer a simpler question instead, and are unaware of the substitution from difficult question to simple question they are making [32].

Pairing with attribute substitution, organizational behavior argues that we see time and again the development of reward systems that prioritize one behavior, even though another behavior is what is wanted. One such example is in the school system: while the behavior we wish to occur is knowledge transfer from teacher to student, earning high grades becomes the behavior that is rewarded—leading to organized cheating groups [33].

We map surrogation onto `-fret-clean` by way of following our constructed argument from the literature. Starting with Goodhart's Law, we can surface the following idea: reduction of libc pointers began as a proxy measure for defense, and silently became a target. The original email hints at further work on stack sanitization as if it is itself the goal [9]: removal of libc pointers silently transmogrifies into the target, rather than remain a metric that helps us understand the larger question of attacker constraint.

The analogue is in ROP gadget removal: even massive amounts of gadgets removed may not meaningfully constrain an attacker [29]; using gadget reduction as the target elides its usefulness as a potential measure towards attacker constraint. There is no reason to believe, save for empirical evidence proving otherwise, that the same elision is not also true for pointer reduction.

Following on to attribute substitution, it is clear the difficult question we want to answer is "will `-fret-clean` meaningfully constrain attackers trying to use ROP against a system?" Because that question is genuinely difficult to measure and answer, the question "does `-fret-clean` eliminate return address pointers into libc?" stands in as a simpler question that as a bonus is easy to quantify: it is clear from our results that the mitigation is essentially perfect at eliminating that specific class of pointers into libc. But it is far less clear whether or not `-fret-clean` meaningfully constrains attackers—in isolation, as we measured it in this paper, the answer may well be no. As we mentioned in our discussion, in combination with other mitigations such as internal libc relinking might yield a different answer. The behavior we want to reward is developing mitigations that meaningfully constrain attackers; the behavior that has been rewarded is intuitive-sounding mitigations that plausibly reduce a number on a chart, regardless of whether or not such reductions meaningfully constrain attackers.

Our primary concern is that these proxy metrics will come to serve as stand-ins for overall defensive strategy, a concern that is found in management [34]; this surrogation may well prove catastrophic in providing meaningful defenses. Coupled with the intuitiveness of these compiler-based anti-ROP mitigations, we highlight the need to keep the real question of attacker constraint at the forefront of mitigation development, and reaffirm the need for a genuine battery of empirical measurements of any such mitigation aimed at answering the real question to ensure that the strategy stays true and we do not accidentally produce a false sense of security.

A. Future Work

For completeness, this mitigation should be ported to GCC, the other mainstream open source compiler suite, and reevaluated. Differences in mitigation impacts between LLVM and GCC found in previous work [7] suggest that size increases and runtime penalties may be different enough between the two compiler suites to warrant a more nuanced consideration of the `-fret-clean` mitigation based on workload and compiler suite used.

Acknowledgments

No Generative AI was used in the preparation of this manuscript.

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, pp. 1-34, 2012, doi: https://doi.org/10.1145/2133375.2133377.

[2] J. Wang, P. Xie, Y. Wang, and Z. Rong, "A survey of return-oriented programming attack, defense and its benign use," in *2018 13th Asia Joint Conf. Inf. Secur. (AsiaJCIS)*, Guilin, China, 2018, pp. 83-88, doi: https://doi.org/10.1109/AsiaJCIS.2018.00022.

[3] A. V. Vishnyakov and A. R. Nurmukhametov, "Survey of methods for automated code-reuse exploit generation," *Program. Comput. Soft.*, vol. 47, pp. 271–297, 2021, doi: https://doi.org/10.1134/S0361768821040071.

[4] Y. Ruan, S. Kalyanasundaram, X. Zou, "Survey of return-oriented programming defense mechanisms," *Secur. Comm. Netw.*, vol. 9, pp. 1247–1265, 2016, doi: https://doi.org/10.1002/sec.1406.

[5] T. de Raadt, "OpenBSD 7.6," OpenBSD Project, Oct. 8, 2024. [Online]. Available: https://www.openbsd.org/76.html

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

[7] B. R. Callahan, J. Esposito, A. Arif, and R. Cortinas, "A final return for OpenBSD anti-return-oriented programming mitigations," in *Proc. 21st Annu. Symp. Inf. Assurance (ASIA '26)*, Albany, NY, USA, Jun. 2026, pp. 98-110, doi: https://doi.org/10.2139/ssrn.6869668.

[8] J. Choi, G. W. Hecht, and W. B. Tayler, "Lost in translation: the effects of incentive compensation on strategy surrogation," *Accounting Rev.*, vol. 87, no. 4, pp. 1135-1163, 2012, doi: https://doi.org/10.2308/accr-10273.

[9] T. de Raadt, "clang -fret-clean: cleaning return addresses off stack," OpenBSD tech mailing list, May 25, 2024. [Online]. Available: https://marc.info/?l=openbsd-tech&m=171661784618821

[10] OpenBSD Project, "Enable -fret-clean on amd64, for libc libcrypto ld.so kernel, and all the ssh tools. The dynamic objects are entirely ret-clean, static binaries will contain a blend of cleaning and non-cleaning callers.," OpenBSD CVS Repository, commit 1457ca8725ed4b2243ae5e62536b5691f061f6e3, Jun. 4, 2024. [Online]. Available: https://github.com/openbsd/src/commit/1457ca8725ed4b2243ae5e62536b5691f061f6e3

[11] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur. (CCS)*, Alexandria, VA, USA, 2007, pp. 552–561, doi: https://doi.org/10.1145/1315245.1315313.

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

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

[14] Z. Rong, P. Xie, J. Wang, S. Xu, and Y. Wang, "Clean the scratch registers: a way to mitigate return-oriented programming attacks," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Milan, Italy, 2018, pp. 1-8, doi: https://doi.org/10.1109/ASAP.2018.8445132.

[15] S. Xu and Y. Wang, "Defending against return-oriented programming attacks based on return instruction using static analysis and binary patch techniques," *Science of Computer Programming*, vol. 217, pp. 1-15, 2022, Art. no. 102768, doi: https://doi.org/10.1016/j.scico.2022.102768.

[16] J. Corbet, "Two security improvements for GCC," LWN.net, Sep. 24, 2021. [Online]. Available: https://lwn.net/Articles/870045/

[17] LLVM Project, "[X86] Implement -fzero-call-used-regs option," Feb. 8, 2022. [Online]. Available: https://github.com/llvm/llvm-project/commit/deaf22bc0e306bc44c70d2503e9364b5ed312c49

[18] K. Cook, "hardening: introduce CONFIG_ZERO_CALL_USED_REGS," Jul. 20, 2021. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a82adfd5c7cb

[19] M. Larabel, "Benchmarking the performance impact of Linux 5.15's newest protection around side channel attacks," Phoronix, Sep. 3, 2021. [Online]. Available: https://www.phoronix.com/review/zero-used-regs

[20] "Consider removing/not recommending CONFIG_ZERO_CALL_USED_REGS #82," GitHub a13xp0p0v/kernel-hardening-checker. [Online]. Available: https://github.com/a13xp0p0v/kernel-hardening-checker/issues/82

[21] J. Voisin, "Paper notes - clean the scratch registers: a way to mitigate return-oriented programming attacks," Artificial Truth, Oct. 5, 2021. [Online]. Available: https://dustri.org/b/paper-notes-clean-the-scratch-registers-a-way-to-mitigate-return-oriented-programming-attacks.html

[22] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you C: Controlling side effects in mainstream C compilers," in *2018 IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, London, UK, 2018, pp. 1-15, doi: https://doi.org/10.1109/EuroSP.2018.00009.

[23] A. C. Wynn, "Stack clearing device and method," U.S. Patent 6 550 058 B1, Apr. 15, 2003.

[24] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 1947.

[25] T. de Raadt, "OpenBSD 6.0," OpenBSD Project, Sep. 1, 2016. [Online]. Available: https://www.openbsd.org/60.html

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. -R. Sadeghi, "Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, 2013, pp. 574-588, doi: 10.1109/SP.2013.45.

[27] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: stopping address space leakage for code reuse attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Denver, CO, USA, Oct. 2015, pp. 280-291, doi: https://doi.org/10.1145/2810103.2813694.

[28] M. D. Brown and S. Pande. "Is less really more? towards better metrics for measuring security improvements realized through software debloating," in *12th USENIX Workshop Cyber Secur. Experimentation Test (CSET 19)*, Santa Clara, CA, USA, 2019, pp. 1-9.

[29] J. Coffman, D. M. Kelly, C. C. Wellons, and A. S. Gearhart, "ROP gadget prevalence and survival under compiler-based binary diversification schemes," in *Proc. 2016 ACM Workshop Softw. PROtection (SPRO '16)*, Vienna, Austria, 2016, pp. 15-26, doi: https://doi.org/10.1145/2995306.2995309.

[30] C. A. E. Goodhart, "Problems of monetary management: the U.K. experience," in *Monetary Theory and Practice: The UK Experience*, London UK: Macmillan Education UK, 1984, ch. 3, pp. 91-121.

[31] M. Strathern, "'Improving ratings': audit in the British University system," *Eur. Rev.*, vol. 5, no. 3, pp. 305-321, Jul. 1997, doi: https://doi.org/10.1002/(SICI)1234-981X(199707)5:3<305::AID-EURO184>3.0.CO;2-4.

[32] D. Kahneman and S. Frederick, "Representativeness revisited: attribute substitution in intuitive judgment," in *Heuristics and Biases: The Psychology of Intuitive Judgment*, T. Gilovich, D. Griffin, and D. Kahneman, Eds., Cambridge, UK: Cambridge University Press, 2002, ch. 2, pp. 49-81, doi: https://doi.org/10.1017/CBO9780511808098.004.

[33] S. Kerr, "On the folly of rewarding A, while hoping for B," *Acad. Manage. J.*, vol. 18, no. 4, pp. 769-783, Dec. 1975.

[34] P. W. Black, T. O. Meservy, W. B. Tayler, and J. O. Williams, "Surrogation fundamentals: measurement and cognition," *J. Manage. Accounting Res.*, vol. 34, no. 1, pp. 9-29, 2022, doi: https://doi.org/10.2308/JMAR-2020-071.