Do we still need canaries in the coal mine? Measuring shadow stack effectiveness in countering stack smashing

Hugo Depuydt[1], Merve Gülmez[2], Thomas Nyman[3], and Jan Tobias Mühlberg[4]

[1] ENS Rennes, France `hugo.depuydt@ens-rennes.fr`
[2] Ericsson Security Research, Sweden `merve.gulmez@ericsson.com`
[3] Ericsson Product Security, Sweden `thomas.nyman@ericsson.com`
[4] Université Libre de Bruxelles, Belgium `jan.tobias.muehlberg@ulb.be`

Abstract Stack canaries and shadow stacks are widely deployed mitigations to memory-safety vulnerabilities. While stack canaries are introduced by the compiler and rely on sentinel values placed between variables and control data, shadow stack implementations protect return addresses explicitly and rely on hardware features available in modern processor designs for efficiency. In this paper we investigate whether stack canaries and shadow stacks provide similar levels of protections against sequential stack-based overflows. Based on the Juliet test suite, we evaluate whether 64-bit x86 (x86-64) systems benefit from enabling stack canaries in addition to the x86-64 shadow stack enforcement. We observe divergence in overflow detection rates between the GCC and Clang compilers and across optimization levels, which we attribute to differences in stack layouts generated by the compilers. We also find that x86-64 shadow stack implementations are more effective and outperform stack canaries when combined with a stack-protector-like stack layout. We implement and evaluate an enhancement to the Clang x86-64 shadow stack instrumentation that improves the shadow stack detection accuracy based on this observation.

1 Introduction

The urgency of mitigating memory-safety vulnerabilities in C and C++ software has grown under increasing regulatory scrutiny [25]. Memory-safety issues are one of the oldest problems in computer security and remain a persistent challenge despite decades of advances in both offensive and defensive techniques [29]. Stack canaries [12] stand out as one of the earliest systematic mitigations to achieve widespread adoption. In this paper, we reassess stack canaries in light of modern hardware-assisted mitigations, particularly shadow stacks [8], now operational in commodity x86-64 systems [21,22].

Stack canaries—a reference to the historic practice of bringing canary birds into coal mines to alert miners of toxic gases—are sentinel values placed between local variables and control data on the stack to detect buffer overflows. Shadow stacks, in contrast, specifically protect function return addresses, preventing exploits such as return-oriented programming (ROP) [27] that hijack a program's control flow. While shadow stacks target a different threat model, both techniques defend against sequential overflows that corrupt the stack canary or return address. We hypothesize that, with modern compiler optimizations omitting other control data from stack frames, stack canaries and shadow stacks offer comparable protection against sequential overflows.

This paper and contributions. This paper investigates whether conventional stack canaries are redundant in applications where hardware-assisted shadow stacks are enabled. We evaluate the effectiveness of both techniques on modern x86-64 systems using the NIST Juliet C/C++ Test Suite [7] which contains a wide range of C/C++ code examples with buffer-overflows among the 118 Common Weakness Enumeration (CWE) categories the suite covers. Our key contributions and findings include:

1. **Systematic evaluation between GCC and Clang.** We evaluate the effectiveness and performance of stack canaries and x86-64 shadow stack in GCC and Clang and show differences in the detection accuracy between the compilers. Across our sample set, Clang demonstrates a better detection rate with stack canaries than GCC, while shadow stacks alone detect significantly fewer buffer overflows compared to stack canaries. We further investigate the reasons for this difference.
2. **Impact of compiler stack layouts.** The stack layout generated by the compiler has a significant impact on detection accuracy for both stack canaries and shadow stack. The stack layout varies between the different compilers, the level of program optimizations used, and between different variants of the stack-canary instrumentation, i.e., the different option variants in the `-fstack-protector` family.
3. **Enhancements to Clang's shadow stack support.** To enhance the protection the x86-64 shadow stack offers against sequential buffer overflows, we propose new Clang compiler options that emulate stack-protector layouts while relying on shadow stack checks. Our evaluation shows these new options improve detection accuracy while allowing stack canary checks to be omitted and incur only a small performance degradation ($\approx 1.6\%$ and $\approx 0.4\%$ on the SPEC CPU 2017 intrate and intspeed test suites respectively) which is lower than that of the corresponding stack canaries ($\approx 3\%$ and $\approx 3.3\%$, when applied to all functions) and comparable to that of conventional x86-64 shadow stacks ($\approx 1.5\%$ and $\approx 0.6\%$).

Our observations have already been shared with security researchers in the GCC and Clang communities, with whom we confirmed that our findings can be disclosed and that the compiler features are working as intended. An extended version of this paper with additional technical details is available as [14].

2 Background

Over half a century since their discovery [2], memory-safety vulnerabilities have become the most prevalent class of software vulnerability [25]. Major software manufacturers, such as Microsoft and Google [16], attribute up to 70% of vulnerabilities discovered in their products to memory-safety issues [24,16]. Examples of vulnerabilities, attacks, and outages attributed to memory-safety issues include the Heartbleed bug in OpenSSL [28], the BLASTPASS exploit chain used to deliver commercial spyware [9], and the CrowdStrike outage of 2024 [13].

2.1 Stack canaries

As exploitation techniques for memory-safety vulnerabilities such as buffers overflows became prevalent, research into countermeasures resulted in several mitigation schemes of which *stack canaries* were eventually integrated into mainstream compilers [31,17]. Stack canaries detect a stack buffer overflow before the execution of malicious code can occur: A function's stack layout is instrumented with canary values between local variables and the return address. A contiguous buffer overflow modifies the stack canary before corrupting the return address. A check inserted by the compiler before returning from a function detects if the canary has been modified and calls an error-handling routine, `__stack_chk_fail()`, that typically terminates the program, rather than returning with a corrupt return address. On x86-64 Linux with the GNU C library (glibc), the stack canary is a 64-bit random value with the final bytes zeroed to make it simultaneously act as a terminator canary. A survey of deployed compiler-based mitigations indicates that stack canaries are enabled in 85% of desktop binaries [32].

Allocation placement. The placement of allocations in the stack frame relative to the stack canary and saved register values in the frame record is significant for the overflow detection efficacy of stack canaries. Both GCC and Clang use the following rules when deciding allocation placement for local variables with stack protector [23]:

- Large arrays and structures containing large arrays are nearest to the canary.
- Small arrays and structures containing small arrays are next nearest to the canary.
- Variables that have had their address taken are third nearest to the canary.
- Other variables whose sizes are known at compile time are further.
- Dynamically-sized variables, such as C99-style arrays, are the furthest.

Jiang et al. [20] observe that GCC and Clang can generate different layouts for allocation placement even when the stack protector feature is enabled and identify one instance in the RecIPE memory error defense benchmark where the differences in the relative placement of an allocations between GCC and Clang result in different outcomes in a benchmark test case.

2.2 Shadow stacks

A shadow stack [8] is a mechanism to protect a function's stored return address while it resides on the call stack. To achieve this, a copy of the return address is stored in a separate, isolated region of memory area that is not accessible to the attacker. Before the function returns, its stored return address is compared against the protected copy on the shadow stack to ensure the original address has not been modified, for example as a result of a buffer overflow. If there is a mismatch between the return address on the call stack and its copy on the shadow stack, program execution is terminated.

By protecting the integrity of return addresses, shadow stacks ensure that returning from function calls leads back to the respective call site, a form of backward-edge Control-Flow Integrity (CFI) [1]. Attacks that violate CFI have been demonstrated at different levels of semantic granularity, across programming languages, and in the presence of defensive mechanisms [6,26,5,19,15,4]. The prevalence of ROP, in particular, have prompted processor manufacturers to incorporate hardware support for shadows stacks into all major processor architectures including x86-64 [10], AArch64 [11], and

RISC-V [30]. On x86-64 hardware shadow-stack support is provided by Intel's Control-flow Enforcement Technology (CET) as well as AMD's Shadow Stack hardware features. At the time of writing, recent releases of commodity Linux distributions, such as Ubuntu 18.04 ship with the necessary software support for x86-64 shadow stacks, but software built with shadow stack support (`-fcf-protection=return` in GCC 8.0.1 and Clang 7.0.0 and later) must explicitly opt-in to shadow stack enforcement.

Comparison of stack canaries and x86-64 shadow stack. Table 1 shows a high-level comparison between stack canaries and the x86-64 shadow stack. The x86-64 shadow stack operates as a mechanism similar to stack canaries to protect the return address. However, due to its placement, the x86-64 shadow stack cannot protect the frame pointer, whereas stack canaries detect the corruption of the frame pointer and the return address. Stack canaries rely on heuristics to determine which functions receive the canary instrumentation based on the option shown in Table 1 (with the exception of `-fstack-protector-all` which applies to all functions). The x86-64 shadow stack applies implicitly to all functions. Stack canaries will detect any contiguous stack buffer overflows that overwrite the canary value. The canary check can be bypassed if the canary value becomes known to an adversary who can overflow the buffer and overwrite the canary with its original value, or if the buffer overflow is not contiguous the adversary can "skip" over the canary without overwriting it. The x86-64 shadow stack, in contrast, can prevent the replacement of the stored return address with arbitrary, or mismatched return addresses regardless of which kind of write primitive is used to manipulate the contents of the call stack.

Table 1: Comparison between stack canaries and the x86-64 shadow stack

Compiler option	Protection of frame record		Characteristics	
	Frame pointer	Return address	Protection coverage	Enforcement model
Stack Canaries				
`-fstack-protector`	✓	✓	Heuristic[1]	Probabilistic
`-fstack-protector-strong`	✓	✓	Heuristic[2]	Probabilistic
`-fstack-protector-all`	✓	✓	All functions	Probabilistic
Shadow stack				
`-fcf-protection=return`	✗	✓	All functions	Deterministic

[1] : `-fstack-protector` applies stack canaries to any function with character arrays that equal or exceed the `ssp-buffer-size` setting set via `--param=ssp-buffer-size` (8 by default).

[2] : `-fstack-protector-strong` applies stack canaries to any function that 1) takes the address of any of its local variables on the right-hand-side of an assignment or as part of a function argument, or 2) allocates a local array, or a struct or union which contains an array, regardless of the type of length of the array, or 3) has explicit local register variables.

3 Methodology

An often overlooked problem in validating compiler-based hardening features is test coverage and assurance of correctness. In normal application development, the codebase is finite and known; developers focus on ensuring that all code paths within their application are tested and function correctly. A compiler, in contrast, is used by countless developers to build a variety of applications. In reality, most security hardening features are tested by just a small number of regression or unit tests [3]. Even widely deployed features, such as stack canaries, can exhibit gaps that affect their effectiveness [18] as applying them to large amounts of code successfully does not necessarily establish their effectiveness; it just demonstrates the feature does not interfere with the normal operation of the code. To evaluate effectiveness, a common approach is to use vulnerable programs, i.e., known Common Vulnerability Enumerations (CVEs). However, CVE-based evaluation is limited both in scope, granularity, and scalability as proof-of-concept exploits are available for relatively few CVEs. A more systematic approach is to use a benchmark suite such as Juliet [7]. Previous work by Jiang et al. [20] focuses on evaluating different defenses across different types of memory locations, rather than over varying stack layouts and omit the x86-64 shadow stack from their evaluation. To the best of a knowledge, no prior work has attempted to perform a quantitative comparison of stack canaries and the x86-64 shadow stack.

3.1 Goal and problem statement

The goal of our evaluation is to answer the following research questions:

RQ1. Are the detection rates of stack canaries and the x86-64 shadow stack consistent across different compilers in large-scale tests.
RQ2. Is the detection rate of contiguous, stack-based buffer overflows comparable between the x86-64 shadow stack stack canaries.
RQ3. Is the impact on software performance in real-world use cases comparable between the x86-64 shadow stack and stack canaries.

Finally, since it is known that the efficacy of stack canaries is affected by the relative placement of a function's allocations on the call stack (see Section 2.1) we evaluate the impact of similar placement heuristics applied for stack canaries on the detection rate of the x86-64 shadow stack:

RQ4. Is the detection rate of the x86-64 shadow stack improved by a `-fstack-protector`-like stack frame layout?

Measuring detection rate. To evaluate RQ1 and RQ2, we use the Juliet test suite [7]. It is a collection of C/C++, C#, and Java programs with known defects organized by the corresponding CWE categories. The latest version released in 2017 covers 64099 C/C++ cases, 28942 C# cases, and 28,881 Java cases. Although the test cases in the Juliet suite are artificial, the defects in it are sourced from real-world applications, including known CVEs. That said, using Juliet for run-time evaluation, rather than the

Table 2: Relevant CWE categories in Juliet C/C++ version 1.3.

CWE Category	**# Test Cases**			
	Total	*Excluded*	*Selected*	*Detectable*
CWE121 Stack-Based Buffer Overflow	4944	96	4848	3562
CWE122 Heap-Based Buffer Overflow	5922	192	5730	1426
CWE124 Buffer Underwrite	2048	96	1952	604
CWE194 Unexpected Sign Extension	1152	384	768	192
CWE195 Signed-to-Unsigned Conversion Error	1152	384	768	288

static analysis it was designed for, comes with a number of challenges. In this work, we focus on evaluating the *difference* of detection rate in RQ1 and RQ2. As such, the results in Section 4.1 should not be taken as indicative of the security of the schemes evaluated, merely as indications of their parity under conditions causing contiguous overflow.

Test case selection. Not all of the 118 CWE categories covered by the Juliet test suite exhibit buffer overflow defects. To keep the compilation and run-time of tests manageable, we had to narrow down the subset of test cases to evaluate those that exhibit contiguous buffer overflow behavior. Through empirical assessment, we narrow our evaluation to the five CWE categories in Table 2 which exhibit relevant defects.

Measuring performance impact. To evaluate RQ3, we use the SPEC CPU 2017 benchmark suite and report the results in Section 4.2, using `-O2 -march=native` for all cases, with 4 copies for rate tests, and 12 threads for speed tests, corresponding to the number of cores without simultaneous multithreading (SMT). SMT and address space layout randomization (ASLR) were disabled for all tests.

Measuring impact of allocation layout. To evaluate RQ4 we implemented a modification to the Clang compiler that applies the stack layout changes implied by the `-fstack-protector` family of options without enabling the stack canary instrumentation and checks. To achieve this, we reuse the analysis passes that the stack canary instrumentation uses, but remove the generation of the failure path, check, and stack canary allocation. These changes result in a `-fstack-layout` -family of options that make local allocations ordered by the rules described in Section 2.1, with large arrays and structures containing large arrays closer to the return address than small arrays and variables. We then evaluate x86-64 shadow stack detection accuracy when combined with the new `-fstack-layout` -family of options and report our results in Section 4.1.

3.2 Experimental setup

We opted to use a source-based Linux distribution, Gentoo Linux, to ensure that the test cases and all dependencies were built with stack canary and x86-64 shadow stack options and the correct compiler. We used GCC 13.3.1_p20240614 p1 and Clang version

18.1.8 along with Gentoo's glibc 2.39-r6, on Gentoo's Linux Kernel version 6.6.51-gentoo-dist-hardened. Experiments executed on an Intel NUC 13 Pro Mini (NUC13ANK) with a Raptor Lake Intel Core™ i7-1360P and 14 GB random access memory (RAM).

4 Evaluation and Results

4.1 Results: Detection of Contiguous Overflows

Figure 1 illustrates the stack canary and shadow stack results for GCC and Clang under the optimization levels `-O2` and `-O0`. All test cases show better detection rates with `-O0` compared to `-O2`, as optimizations under `-O2` can exploit undefined behavior in test cases in ways which may suppress or contain the extent of buffer overflows. For example, in some CWE122 and CWE194 cases the offending buffers are not referenced by the test code after the initialization that overflows them, allowing the compiler to optimize away the entire array access as dead code. Overall, across all our tests, the combination of the x86-64 shadow stack and either `-fstack-protector-strong` or `-fstack-protector-all` using Clang at `-O0` has the highest detection rate, but only reaches $\approx 33\%$ detection of the selected cases in Table 2. Consequently, for comparison, it is more meaningful to compare the detection rates *within* a certain optimization level than the rates *across* optimization levels.

Stack canary detection rates. An overall comparison of the plots in Figure 1 reveals that Clang demonstrates better detection rate with stack canaries than GCC. The `-fstack-protector-all` and `-fstack-protector-strong` options consistently outperform the `-fstack-protector` option, which is expected. The `-fstack-protector -all` option does not perform significantly better than `-fstack-protector-strong`. We attribute the differences in detection results between GCC and Clang as follows:

Differences in stack layout between compilers: In GCC, an array may be placed before another array, while in Clang, the same array may be placed after. This difference in stack layout can result in arrays being positioned closer to the stack canary and return addresses, depending on the compiler. While [20] initially highlighted this variation between compiler stack-layout, our results offer a more quantitative analysis of its impact.

Differences in handling of `alloca()` calls with constant values: Clang treats `alloca` calls with constant values similarly to a local array declaration, optimizing the allocation accordingly. In contrast, GCC employs a dynamic implementation, which may allocate additional space, particularly at the `-O0` optimization level. This behavior can allow a buffer to overflow with a specific length without modifying the stack canary.

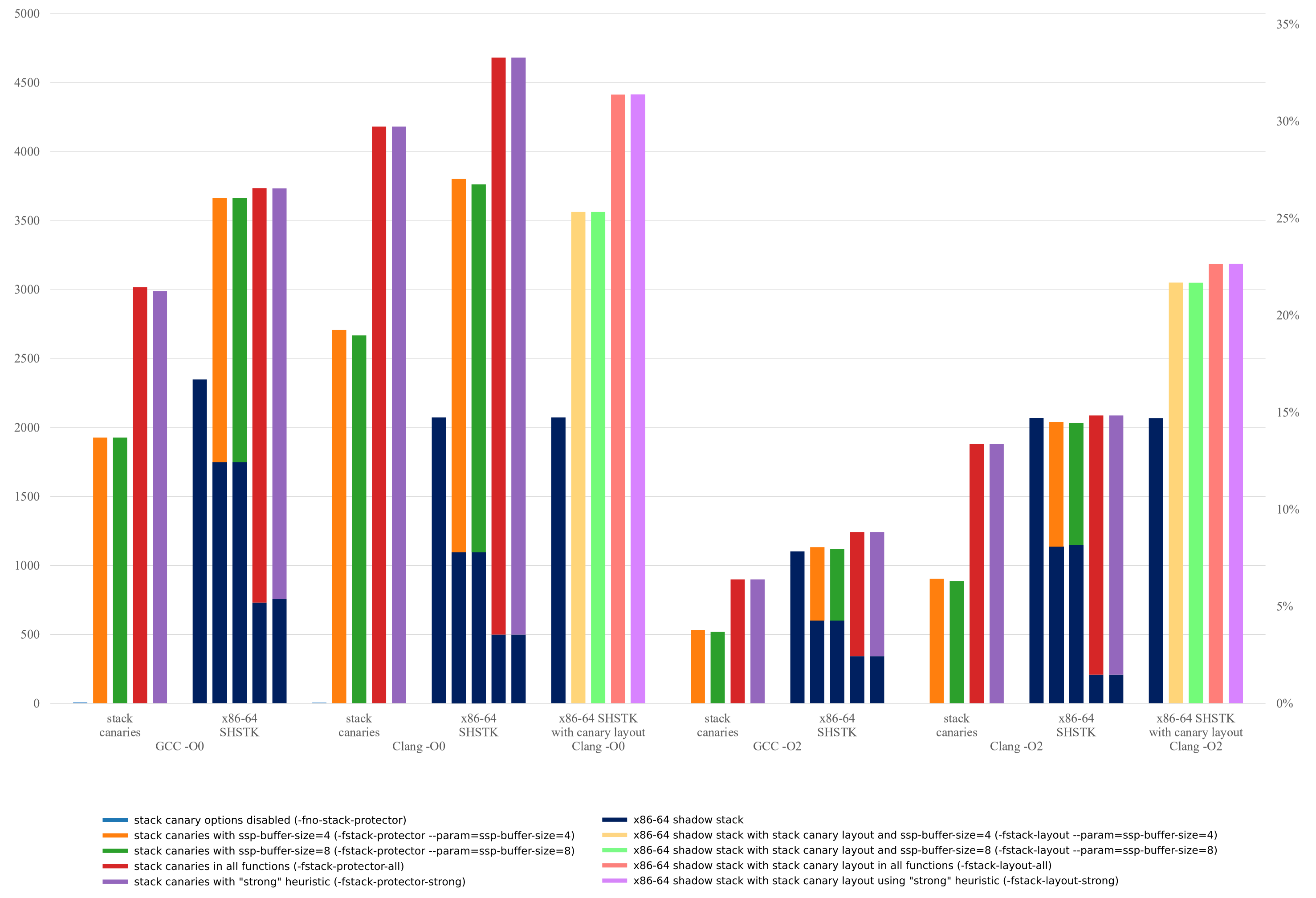

Figure 1: Comparison of Juliet test results by compiler, optimization level, and options. The *stack canaries* bars show the detection rates for different stack canary options (indicated in the legend) with the x86-64 shadow stack disabled. The *x86-64 SHSTK* bars show the detection rate for the x86-64 shadow stack separately, and when combined with different stack canary options (indicated in the legend). The *x86-64 SHSTK with canary layout* bars show the detection rate for the x86-64 shadow stack when combined with the proof-of-concept `-fstack-layout`-family options for Clang `-O0` and `-O2` configurations. The left axis shows the number of test cases with detections, while the right axis shows the percentage of detected cases relative to the *Selected* test cases shown in Table 2.

Table 3: Geometric mean of performance degradation based on SPEC CPU 2017 results.

Protection variant	intrate	intspeed
stack canaries with "strong" heuristic (`-fstack-protector-strong`)	0.55%	0.27%
stack canaries in all functions (`-fstack-protector-all`)	3.05%	3.26%
x86-64 shadow stack	1.48%	0.57%
x86-64 shadow stack with stack canary layout using "strong" heuristic (`-fstack-layout-strong`)	1.55%	0.40%
x86-64 shadow stack with stack canary layout in all functions (`-fstack-layout-all`)	1.60%	0.41%

x86-64 shadow stack detection rates. When the x86-64 shadow stack is enabled without stack canaries present, its detection rates exceeds those of stack canaries in the `-O2` case for both GCC and Clang, but the `-O0` results are reversed with the best performing stack canary options (`-fstack-protector-all` and `-fstack-protector-strong`) detecting more overflows than the x86-64 shadow stack for both GCC and Clang. Considering all compilation options, there are 1217 tests that the x86-64 shadow stack detects successfully that stack canaries do not, and 163 tests that stack canaries detect that the x86-64 shadow stack does not.

x86-64 shadow stack with `-fstack-layout` detection rates. The results for Clang with `-O0` and `-O2` are shown in Figure 1. They show a consistent improvement in x86-64 shadow stack detection accuracy when combined with the new options. We however, identified a specific limitation in our approach of reusing the existing `-fstack-protector` analysis passes: in some cases, such as when a function spills callee-saved registers the stack canary is not placed right next to the return address, but also in such a way to protect any spilled register values. Our `-fstack-layout` options do not alter the placement of spilled registers leaving them unprotected by the x86-64 shadow stack.

4.2 Results: Performance

We evaluate the performance impact of different stack canary implementations and the x86-64 shadow stack using the SPEC CPU 2017 intrate and intspeed benchmarks. For the performance evaluation we focus on the `-fstack-protector-strong` and `-fstack-protector-all` options as these outperformed the other `-fstack-protector` variants in the detection of contiguous overflows experiments (Section 4.1). To improve the consistency of results, we disabled ASLR and SMT. All benchmarks were compiled using Clang compiler with optimization level `-O2` and `-march=native`. We exclude the `548.exchange2_r` benchmark as it is written in Fortran and not supported by Clang.

Table 3 gives an overview of the performance results. Overall we found that the `-fstack-protector-strong` options degraded performance the least (on average $\approx 0.55\%$ on intrate and $\approx 0.27\%$ on intspeed) and `-fstack-protector-all` the most (on average $\approx 3.05\%$ on rate and $\approx 3.26\%$ on speed). The x86-64 shadow stack falls between these stack canary variants by degrading performance on average by $\approx 1.48\%$ on intrate and $\approx 0.57\%$ on intspeed. The x86-64 shadow stack with `-fstack-layout-strong` and `-fstack-layout-all` seems to have comparable performance to that of the conventional x86-64 shadow stack.

4.3 Conclusions from Evaluation

In Section 3 we set out to answer four research questions. Our conclusions regarding these questions and based on the above evaluation of detection and performance is:

RQ1. Are the detection rates of stack canaries and the x86-64 shadow stack consistent across different compilers in large-scale tests.
No ✗: Our results show that different options exhibit different detection rates across compilers.

RQ2. Is the detection rate of contiguous, stack-based buffer overflows comparable between the x86-64 shadow stack stack canaries.
No ✗: The x86-64 shadow stack does not consistently outperform stack canaries in terms of detection rates.

RQ3. Is the impact on software performance in real-world use cases comparable between the x86-64 shadow stack and stack canaries.
No ✗: We measured consistently larger performance impacts for the x86-64 shadow stack compared to `-fstack-protector-strong` in our benchmarks.

RQ4. Is the detection rate of the x86-64 shadow stack improved by a `-fstack-protector`-like stack frame layout?
Yes ✓: We measured a consistent improvement in detection rate for the x86-64 shadow stack with our `-fstack-layout`-family of options.

5 Conclusion

We compared stack canaries and the x86-64 shadow stack for detecting contiguous overflows and return-address corruption across GCC and Clang at optimization levels `-O0` and `-O2`, using the Juliet test suite and SPEC CPU 2017 benchmarks. Clang's stack canaries consistently outperformed GCC's, and both compilers detected more issues at `-O0` than at `-O2`. At `-O2`, the shadow stack caught more overflows than canaries—-with Clang again ahead of GCC—-and by adopting Clang's canary-style stack layout (but without canary checks) we raised the shadow stack's detection rate well above standard canaries, at negligible runtime cost.

These findings suggest that, on supported hardware, a shadow-stack configuration could replace stack canaries, despite its return addresses being more predictable than random canary values. However, because the Juliet suite does not model real-world exploits—-and GCC's lower detection may stem from either a more efficient stack layout or biases in the test suite—-we acknowledge that our evaluation does not enable us to make strong claims regarding the security of the different configurations. We also show that stack-protector implementations influence code generation and stack ordering beyond inserting canaries, and recommend that similar layout effects be evaluated for other mechanisms, such as ARM Pointer Authentication.

Acknowledgements

We want to thank Kristof Beyls, William Huhn, Siddhesh Poyarekar and Niklas Lindskog for reviewing an earlier version of this paper. This research is partially funded by the CyberExcellence programme of the Walloon Region, Belgium (grant 2110186).

References

1. Abadi, M. et al.: Control-flow integrity. In: CCS '05 (2005). https://doi.org/10.1145/1102120.1102165
2. Anderson, J.P.: Computer Security Technology Planning Study Volume 1 - Executive Summary. Tech. Rep. AD-758 206, James P. Anderson and Co. (Oct 1972), https://apps.dtic.mil/sti/citations/AD0758206
3. Beyls, K.: [RFC] BOLT-based binary analysis tool to verify correctness of security hardening. LLVM Discussion Forums (Apr 2024), https://discourse.llvm.org/t/78148
4. Bierbaumer, B. et al.: Smashing the Stack Protector for Fun and Profit. In: Janczewski, L.J., Kutyłowski, M. (eds.) ICT Systems Security and Privacy Protection, SEC '18, vol. 529, pp. 293–306. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-99828-2_21
5. Bittau, A. et al.: Hacking Blind. In: S&P '14 (2014). https://doi.org/10.1109/SP.2014.22
6. Bletsch, T. et al.: Jump-oriented programming: A new class of code-reuse attack. In: ASIACCS '11 (2011). https://doi.org/10.1145/1966913.1966919
7. Boland, T., Black, P.E.: Juliet 1.1 C/C++ and Java Test Suite. Computer **45**(10), 88–90 (Oct 2012). https://doi.org/10.1109/MC.2012.345
8. Burow, N., Zhang, X., Payer, M.: SoK: Shining Light on Shadow Stacks. In: S&P '19 (2019). https://doi.org/10.1109/SP.2019.00076
9. Citizen Lab: BLASTPASS: NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild. Tech. rep., Citizen Lab, University of Toronto (Sep 2023), https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/
10. Corbet, J.: Shadow stacks for user space. LWN.net (Feb 2022), https://lwn.net/Articles/885220/
11. Corbet, J.: Shadow stacks for 64-bit Arm systems. LWN.net (Aug 2023), https://lwn.net/Articles/940403/
12. Cowan, C. et al.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: USENIX Security '98 (1998), https://www.usenix.org/legacy/publications/library/proceedings/sec98/cowan.html
13. CrowdStrike: External Technical Root Cause Analysis — Channel File 291. Tech. rep. (Aug 2024), https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf
14. Depuydt, H. et al.: Do we still need canaries in the coal mine? measuring shadow stack effectiveness in countering stack smashing (extended version) (2024), https://doi.org/10.48550/arXiv.2412.16343
15. Evans, I. et al.: Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In: CCS '15 (2015). https://doi.org/10.1145/2810103.2813646
16. Google: An update on Memory Safety in Chrome. Google Security Blog (Sep 2021), https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html
17. Guelton, S., Poyarekar, S.: Use compiler flags for stack protection in GCC and Clang. Red Hat Developer Blog (May 2022), https://developers.redhat.com/articles/2022/06/02/use-compiler-flags-stack-protection-gcc-and-clang#
18. Hebb, T.: CVE-2023-4039: GCC's -fstack-protector fails to guard dynamic stack allocations on ARM64. Meta Red Team X Blog (Sep 2023), https://rtx.meta.security/mitigation/2023/09/12/CVE-2023-4039.html
19. Hu, H. et al.: Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In: S&P '16 (2016). https://doi.org/10.1109/SP.2016.62
20. Jiang, Y. et al.: RecIPE: Revisiting the Evaluation of Memory Error Defenses. In: ASIACCS '22 (2022). https://doi.org/10.1145/3488932.3524127
21. Larabel, M.: Intel Shadow Stack Finally Merged For Linux 6.6. Phoronix (Aug 2023), https://www.phoronix.com/news/Intel-Shadow-Stack-Linux-6.6
22. Larabel, M.: Glibc Updated For Recent Linux CET Shadow Stack Support. Phoronix (Jan 2024), https://www.phoronix.com/news/Glibc-Intel-CET-Shadow-Stack
23. Magee, J.: [cfe-dev] What do the different stack-protector levels protect in Clang? (Apr 2017), https://lists.llvm.org/pipermail/cfe-dev/2017-April/053662.html
24. MSRC: A proactive approach to more secure code. Microsoft Security Response Center Blog (Jul 2019), https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
25. ONCD: Back to the Building Blocks: A Path Toward Secure and Measurable Software. Whitepaper, United States White House Office of the National Cyber Director (2024), https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
26. Roemer, R. et al.: Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans. Inf. Syst. Secur. **15**(1), 2:1–2:34 (Mar 2012). https://doi.org/10.1145/2133375.2133377
27. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: CCS '07 (2007). https://doi.org/10.1145/1315245.1315313
28. Synopsys: Heartbleed Bug (Apr 2014), https://heartbleed.com/
29. Szekeres, L. et al.: SoK: Eternal War in Memory. In: S&P '13 (2013). https://doi.org/10.1109/SP.2013.13
30. Traynor, B.: The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture, Chapter 35. Control-flow Integrity (CFI) (Nov 2024), https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-ade2bfb-2024-11-28
31. Whitney, T.: /GS (Buffer Security Check). Microsoft Learn (Mar 2021), https://learn.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=msvc-170
32. Yu, R. et al.: Building Embedded Systems Like It's 1996. In: NDSS '22 (2022). https://doi.org/10.14722/ndss.2022.24031