PIBE: Practical Kernel Control-flow Hardening with Profile-guided Indirect Branch Elimination (Extended Abstract)

Victor Duta, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida

Vrije Universiteit Amsterdam

1. Motivation

Control flow hijacking attacks have ranked among the most dangerous forms of compromise for decades, but now that the attackers have also made the jump to the transient execution domain, they have become particularly worrying.

In this paper, we focus on making control-flow hijacking defenses practical for real-world deployment in OS kernels. Our methodology applies equally to non-transient and transient hijacking defenses, but we deliberately limit ourselves to the latter, for two reasons. First, transient execution attacks [12, 6, 7] are potentially more versatile than traditional control-flow attacks as they can be triggered across address space and privilege boundaries, leading to sensitive data leaks and thus loss of confidentiality even in the absence of software vulnerabilities. This makes them particularly dangerous for kernels running in cloud environments, which commonly co-locate programs of mutually non-trusting users on the same physical machine. Second, full mitigation of this flavor of attacks is currently not practical as it entails applying complex control-flow alterations and expensive serialization instructions (e.g., `lfence`) to all indirect branches in the kernel.

Moreover, by focusing on the kernel, we cover some of the largest, most security-critical, most privileged and most complex code bases on modern systems typically written in unsafe languages such as C and C++. Kernels are attractive targets to attackers because of their direct interaction with less privileged code, while their security is critical for the security of the whole system. Similarly, the performance degradation introduced by transient defenses in the kernel also affects the whole system, making system administrators reluctant to deploy them. Finally, given that our approach is based on the availability of a profiling workload, kernels are ideal targets, since companies such as Google already maintain representative profiling workloads to enable profile-guided optimizations for their production kernels [9]. We emphasize, however, that our approach applies equally to other code: hypervisors, SGX(-like) enclaves, and user programs.

2. Limitations of the State of the Art

Unfortunately, there is no easy fix to control-flow hijacking in hardware-based solutions, as current hardware fixes (a) do not defend against all common flavors of transient hijacking attacks, (b) have more dire performance implications than state-of-the-art software solutions (e.g., 25-53% overhead as discussed in [1]), and (c) are not available in all vulnerable CPUs already used in production.

State-of-the-art software-based solutions against common transient control-flow hijacking attacks harden vulnerable indirect branches with instrumentation [12, 6, 7] and comprehensive protection is currently prohibitively expensive as it requires applying a costly combination of state-of-the-art defenses [11, 2, 5] for every indirect branch. Our experiments show that a combination of retpolines [11], return retpolines [5], and LVI-related control-flow hardening [2] incurs a slowdown of 62.0% in userspace on SPEC CPU2006 userspace applications, and is even more impactful on kernels, with a 149% slowdown on LMBench. Related efforts to make transient execution defenses practical, such as JumpSwitches [1] do not address all transient hijacking attacks and only defend against Spectre v2. JumpSwitches are adaptive to runtime workload changes, but at the cost of extra CPU time to monitor target frequencies, as well as live-patching the code which, in our experiments, is prone to synchronization overhead due to RCU stalls.

3. Key Insights

In this paper, we show that the high overhead incurred by state-of-the-art mitigations against transient control-flow hijacking attacks is mostly due to the effect of hardening hot branches (**main insight**). Therefore, instead of applying the defenses pervasively to all indirect branches we can apply them to colder paths and devise alternative solutions to guarantee protection on hot code paths. Based on this insight we devised a profile-driven approach that avoids much of the overhead incurred by the heavy-weight defenses, dramatically lowering the performance impact of state-of-the-art mitigations and making them practical for real-world deployment.

We propose PIBE, which offers comprehensive protection against control-flow hijacking at a fraction of the cost of existing solutions, by revisiting design choices in compiler-based code optimization. Our approach decides, for every indirect branch whether to harden it with instrumentation code or elide it altogether using code transformations. By removing specifically the heavy hitters among the indirect branches through tailored profile-guided optimization, PIBE aggressively reduces the number of vulnerable branches to allow the simultaneous application of multiple state-of-the-art defenses on the remaining branches with practical overhead.

The distinction between hot and cold code is essential for achieving optimal performance guarantees but, in most cases, depends on the training workload. In this paper we show that, even though the kernel is complex and has multiple subsystems, the performance critical code paths are common across multiple workloads (**secondary insight**). It is thus easier to infer a robust training data set which guarantees that our approach can provide good (if not optimal) performance guarantees even if the workload changes. This makes our approach more generic and allows it to be applied even by vendors of end-user binary software distributions, using a predetermined profile that is not necessarily identical to the end user's usage.

4. Main Artifacts

The key artifact presented in this paper is a solution that enables practical hardening of indirect branches in the kernel against transient control-flow hijacks at a fraction of the cost of existing solutions. More specifically, we treat the transformations needed for hardening solutions as a cost-benefit optimization game where, for each instruction to be instrumented, the compiler decides to either add instrumentation code (i.e., the transient defense hardening) or to transform the code so as to remove the instruction reducing the number of instrumentation points overall. In particular, to remove instructions with no security side effects, we perform profile-guided indirect branch elimination. The key idea is to revisit entrenched notions in profile-guided optimizations such as indirect call promotion and inlining, through novel algorithms that favor aggressively reducing the number of indirect branches in hot code paths over other traditional optimization objectives.

Our main artifact is implemented as a subset of compiler passes applied to the kernel code base via tools from the LLVM compiler framework. We evaluate the effectivness of our design in delivering strong security guarantees at acceptable overheads using the LMBench kernel intensive benchmarking suite. Our evaluation is both fair and rigorous. We compare against a baseline that reflects how Linux is tipically deployed in real-life scenarios but also against a PGO baseline tuned to give the best possible performance on the LMBench test suite. Furthermore, we evaluate our design using both representative and non-representative training data sets for LMBench.

5. Key Results and Contributions

The most important empirical results of our approach are as follows. First, we show that our design preserves the security guarantees of a costly yet comprehensive combination of transient control-flow hijacking defenses while also lowering the (geometric mean) overhead from 149% to a moderate 10.6% (on LMBench). This empirically proves that our design can deliver strong security guarantees at a fraction of the cost of current solutions, making defenses practical for deployment. Second, we show that even if our algorithms are trained with a different workload our design can still achieve good (if not optimal) performance guarantees (22.5% overhead on LMBench while optimizing with an Apache workload). This experimentally proves that our design is robust and provides speedups even when the workload changes. The key contributions we make in this paper are:

- A comprehensive security analysis of all common transient control-flow hijacking defenses and their performance implication in userspace and the kernel;
- PIBE, a design to offer full, yet practical, mitigation of (transient) control-flow hijacking through profile-guided indirect branch elimination;
- An evaluation that shows that PIBE reduces the overhead of comprehensive protection for transient control flow hijacking by an order of magnitude, from 149% (!) to 10.6%.

Comparison with State of the Art:

Our design builds on state-of-the-art mitigations against speculative control-flow hijacking (i.e., retpolines [11], return retpolines [5], and LVI [2]) and reduces their overhead by eliminating the most performance critical forward and backward edges and thus also the heavy instrumentation that needs to be applied. We thus overcome the main limitation of these mitigations: prohibitive performance overhead. We improve each of the individual defenses performance over the state of the art, well below 5% for each of them. As opposed to similar defense optimizations (i.e., JumpSwitches) we do not execute code at runtime, avoid synchronization overhead, and improve cache locality as the instrumentation is placed close to the indirect call and does not require jumps across the address space (unlike JumpSwitches). Our solution performs better, is simpler, and does not require kernel code modifications. Moreover, our approach is broader, additionally mitigating ret2spec [7] and LVI [12].

6. Why ASPLOS

This multidisciplinary systems work covers all three of the main ASPLOS topics: computer architecture and hardware (transient execution attacks), programming languages and compilers (use of compiler instrumentation and optimization), and operating systems and networking (protecting the OS kernel). ASPLOS has in recent years included papers on control flow hijacking defenses [3, 4] transient execution defenses [10], and kernel hardening against information leaks [8]. Each of these topics is also covered in this paper. As such, ASPLOS is a perfect fit for this work.

7. Citation for Most Influential Paper Award

PIBE is the first approach that makes comprehensive defenses against transient control flow hijacking attacks on the kernel practical for real-world deployment. We show that PIBE brings down overhead for a comprehensive protection (with state-of-the-art defenses) against all known such attacks from 149% to just 10.6%.

References

[1] Nadav Amit, Fred Jacobs, and Michael Wei. Jumpswitches: restoring the performance of indirect branches in the era of spectre. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 285–300, 2019.

[2] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html. Accessed: 2020-07-26.

[3] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. Hurdle: Securing jump instructions against code reuse attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 653–666, 2020.

[4] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices*, 52(4):585–598, 2017.

[5] Intel. Deep dive: Managed runtime speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations. Accessed: 2020-08-07.

[6] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[7] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[8] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kmvx: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2019.

[9] Tolvanen Sami, Bill Wendling, and Nick Desaulniers. Lto, pgo, and autofdo in the kernel. In *Linux Plumbers Conference*, 2020.

[10] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–410, 2019.

[11] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. Accessed: 2020-07-26.

[12] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, pages 1399–1417, 2020.