Improving Speculative Execution Attack Mitigations via Mitigation-Aware Compilation

Alon Berkenstadt
Technion
Haifa, Israel
alon.b@campus.technion.ac.il

Yakir Vizel
Technion
Haifa, Israel
yvizel@cs.technion.ac.il

Mark Silberstein
Technion
Haifa, Israel
mark@ee.technion.ac.il

ABSTRACT

Trusted execution environments (TEEs) rely on trusted CPUs. However, recent disclosures reveal data leakage in transient execution attacks, which leave TEEs vulnerable.

Software-based techniques that harden the code to prevent undesirable misspeculation side-effects are the only way to protect against such attacks until hardware gets patched. Such software mitigations are implemented now in compilers and special post-compilation tools. Unfortunately, the mitigations are not properly integrated into compilers and are typically added as an afterthought, creating security and performance issues.

We report several such issues. First, we find that some optimizations turn non-vulnerable code into vulnerable. Thus, the order in which mitigations and optimizations are applied affects the security and requires mitigation pass on an optimized version of the code. At the same time, post-optimization patches are not accounted for by popular optimization techniques, leading to incorrect decisions by optimizer heuristics and significant performance loss.

To overcome the first problem, we suggest an efficient and secure integration of Spectre V1 and Spectre V2 mitigation techniques. For the second, we present a few changes to optimization heuristics that gain up to X5.97 speedup for LVI mitigated code.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Software and its engineering → Compilers**.

KEYWORDS

Transient execution, code generation, software analysis

1 INTRODUCTION

Transient execution attacks such as Spectre, Microarchitectural Data Sampling (MDS), and Load Value Injection (LVI) [10, 21, 26, 29, 30], are an Achilles heel of modern processors. More crucially, however, they effectively break the fundamental confidentiality and integrity guarantees of Trusted Execution Environments (TEEs) offered by most recent processors. As a result, mitigating these attacks is a prerequisite to securely executing bug-free programs in TEEs.

By their very nature, the attacks that exploit hardware vulnerabilities cannot be efficiently mitigated without changing CPU hardware itself. The hardware development and deployment cycle, however, are inherently slow, so hardware mitigations cannot usually be completed in the time window of the publication embargo between the vulnerability discovery to its public disclosure. For example, the time frame between the discovery of LVI to its full hardware mitigation has been two years ([15]). Moreover, replacing the whole fleet of vulnerable CPUs with new ones is often unrealistic.

Therefore, software mitigation is often the only practical means to achieve security. Existing solutions [8, 12, 14] are based on code patches that either prevent potentially vulnerable speculative execution entirely or strive to stop the collateral effects of misspeculation that can be exploited. Recent compilers [8] offer the convenience of applying these patches automatically as part of the compilation toolchain.

Software mitigations, however, come with high performance costs. For example, LLVM's LVI mitigation instruments each memory access with `LFENCE`, leading to dramatic slowdowns [23]. Several tools have been developed to reduce the performance penalty, such as SpecFuzz [24] and PIBE [17], which leverage dynamic runtime information to eliminate the patches that are deemed unnecessary.

Our main claim is that *the rapid evolution of software mitigations has led to a fractured and incoherent ecosystem of tools, patches, techniques, and compiler hacks.* Consequently, the resulting software has lower performance and is less secure than one would hope.

In this paper, we demonstrate several instances of this problem. First, we observe that optimization techniques in modern compilers might introduce new vulnerabilities. For example, Indirect Call Promotion (ICP) transforms an indirect call into a direct call preceded by a conditional branch, which effectively makes it vulnerable to Spectre V1 as reported by SpecFuzz [24]. Notably, recent works apply ICP-based techniques to reduce the overheads of Spectre V2 mitigation in the Linux kernel [2, 17], which in turn makes the code vulnerable to Spectre V1.

Thus, invoking dynamic optimizations *after* the Spectre mitigation pass would be insecure. On the other hand, manually updating all potentially vulnerable optimizations is tedious. Therefore, we show that invoking SpecFuzz as the last stage of the compilation pipeline improves performance significantly.

Second, we observe that many optimizations are *oblivious* to the code modifications introduced by the post-compilation mitigations. Indeed, the heuristics used by the compiler to determine whether or not to perform the code transformations do not account for the mitigation code, thus resulting in potentially incorrect decisions, i.e., adding an ineffective optimization or missing a useful one. To this end, we show two examples. First, we demonstrate a substantial performance improvement in LVI-hardened code due to updating the cost function of the loop unrolling optimizer to consider `LFENCE`s inserted by LLVM to mitigate LVI. Similarly, we show that the Indirect Call Promotion (ICP) heuristic should be modified to account for the cost of Spectre V1 and V2 mitigations.

Interestingly, the updated heuristic affects the performance of the code without mitigations.

In summary, our work highlights the need for a more consistent and holistic approach to integrating software mitigations of speculative execution attacks into compilers and development toolchains.

2 BACKGROUND AND RELATED WORK

2.1 Spectre V1

This vulnerability is attributed to the speculative execution of mispredicted conditional branches. For example, a bounds check bypass (BCB) vulnerability allows an attacker to leak memory by controlling an index to an array accessed by a program. A memory read might directly lead to confidential data leakage. However, memory writes are also of concern because they might enable transient control-flow hijacking similar to memory corruption through a buffer overflow.

Software mitigations. **Branch-free code** is a trivial solution for Spectre V1 and is used in constant-time algorithms. For instance, cryptographic algorithms use it to mitigate timing-attacks that might be speculative but not necessarily [6, 9].

Intel suggested using an **LFENCE** instruction to prevent speculative execution by waiting until the branch condition is properly resolved [16].

Speculative Load-Hardening (SLH)[12] relies on strict hardware enforcement of data dependencies upon load operations or their results. It effectively delays load instructions by creating fake dependencies of preceding conditional jumps.

Optimizations. Several efforts have been focused on **identifying Spectre V1 gadgets** in software that might become vulnerable, assuming the hardware speculation behavior [24, 25, 31, 32]. They analyze the code (statically or dynamically at runtime) and do not apply the mitigation patches to branches deemed secure.

For example, **SpecFuzz**[24] enables dynamic testing for speculative execution vulnerabilities. It performs speculation exposure, i.e., it simulates the mispredicted control flow and identifies Spectre V1 vulnerabilities by fuzzing the mispredicted branches. It is implemented as a post-optimization LLVM pass that instruments code with simulation and analysis logic. Dynamic testing produces a report of vulnerabilities that applies to the input code of the instrumentation pass. Then, the analyzed code combined with the report can be served as an input to a selective mitigation pass and complete the compilation process. As with other dynamic analysis methods, the chosen fuzzing driver affects the precision of the analysis, whereas not covered branches cannot be considered benign.

2.2 Spectre V2

This vulnerability is attributed to the speculative execution of mispredicted indirect branches. Indirect branches include indirect calls, indirect jumps, and return instructions. A mispredicted speculatively executed control flow may lead to information leakage.

Software mitigations. **Retpoline** is a recommended mitigation for Spectre V2. It transforms indirect calls into another software construct with the same functionality but a more constrained speculative behavior. Retpolines are often not used despite the recommendations because they have an inherent performance penalty as execution serializers [11, 13].

Optimizations. PIBE [17] and JumpSwitches[2] focus on increasing the performance of the Linux kernel, which is configured to mitigate speculative control flow hijacking. They suggested using ICP to reduce the Retpoline overheads. In addition, PIBE suggests using inlining to eliminate return instructions and their respective hardening overhead. We observe that ICP is vulnerable to Spectre V1, the vulnerability inherited by these optimizations.

2.3 Load Value Injection (LVI)

Explicit or implicit load instructions might use dummy values or other data from shared microarchitectural buffers during faults or assisted instructions. An attacker can poison the shared buffers with her data or use a dummy value to produce the desired behavior in the victim's execution. This attack was shown to particularly affect SGX enclaves [29].

Software mitigations. LVI mitigation requires serialization of load instructions so that their possibly malicious loaded value cannot be used during transient execution. It is implemented by inserting an `LFENCE` before each memory access and before speculative branches and by avoiding return instruction as it involves both operations of loading a return address from the stack and an indirect jump to that loaded address. This mitigation is deployed in LLVM [8].

In contrast to the Spectre vulnerabilities, which are not expected to be fully eradicated, LVI is considered a bug and is expected to be fixed in future processor generations. In any case, however, running programs on already manufactured processors requires software mitigation to protect against it [15].

2.4 Mitigations in Commodity Compilers

Commodity compilers have already deployed mitigations for the vulnerabilities above. For instance, LLVM applies LFENCE/SLH patches against Spectre V1, Retpolines against Spectre V2, LVI-CFI to mitigate LVI vulnerabilities related to control-flow hijacking.

Speculative Execution Side Effect Suppression (SESES) is a more robust approach to mitigate multiple vulnerabilities at once. It inserts an `LFENCE` instruction before any memory access. In addition, it breaks indirect branches with address in memory (e.g., `RET` instruction) into load instruction followed by an indirect jump with address in a register. Then it inserts an `LFENCE` as a serialization barrier before speculative control-flow branch instructions[8].

3 THREAT MODEL AND EXPERIMENTAL SETUP

We use a standard threat model where an attacker has full control over transient mechanisms. The attacker then abuses that capability to initiate an incorrect transient execution that eventually leaks data. The described settings fit previously known attacks, including cryptographic key exfiltration[5, 7, 18–20, 22, 27, 28, 33, 34].

We use Intel Xeon CPU E5-2620 v2 2.10GHz, Ubuntu 18.04.3 64-bit, Linux kernel v4.15.0-70. Each experiment is executed 10 times, and the mean is reported. The standard deviation is below 5%.

	Only without optimization	Only with optimization
Xerces-C++	6	25
Xalan-C++	18	25
JM19.0	45	2
OpenSSL	2009	2007

Table 1: Unique Spectre V1 vulnerabilities reported by SpecFuzz for programs compiled with different compiler configurations.

4 DEPENDENCY BETWEEN CODE OPTIMIZATIONS AND SPECTRE MITIGATION PASSES

Code optimization passes and the speculative execution attack mitigation pass are not independent. However, today these passes are not aware of each other.

On the one hand, optimizations are usually applied to a final code version. Yet, some optimizations may introduce *new vulnerabilities*. For instance, Indirect Call Promotion (ICP) inserts conditional branches, making it vulnerable to Spectre V1. Similarly, conditional move instructions sometimes get transformed into conditional branches too. Additionally, various optimizations such as inlining, global value numbering (GVN), and other loop optimizations[3], might cause excessive register spilling because of increased register pressure, thereby introducing new load instructions which must be hardened to prevent LVI.

Moreover, some optimizations may improve *both* security and efficiency, e.g., by eliminating load and branch instructions. For example, function inlining eliminates Spectre V2-vulnerable return instructions. Similarly, constant propagation might eliminate a conditional branch if it is redundant because its condition is always satisfied. Thus, if the mitigation pass is applied *after* the optimization pass, it will end up instrumenting fewer instructions and result in lower overall overhead.

To illustrate that optimizations may both introduce new vulnerabilities and eliminate existing ones, we consider four large production programs (see Table 1). We compile them with and without profile-guided/link-time optimizations and run SpecFuzz on the results [1]. We enable compilation with Retpolines to protect against Spectre V2 in both cases, but no Spectre V1 mitigations are introduced.

Table 1 shows the number of unique vulnerabilities reported by SpecFuzz in each configuration. For example, it finds 2009 vulnerabilities in OpenSSL compiled without optimization, which were not present in the optimized version, and 2007 vulnerabilities vice versa. The shared vulnerabilities would be mitigated similarly in both cases.

Modifying the optimizations to include the hardening natively as part of the optimization transformation might be a viable solution, but doing so might lead to unnecessary overheads (since it does not account for dynamic information whereas SpecFuzz does) and breaks the modularity of the compiler toolchain. Accordingly, we claim that the mitigation pass must be performed *on the final code version* after applying both static and dynamic optimizations.

We now show the concrete example of a security issue with the Indirect Call Promotion (ICP) optimization.

[1]LLVM performs link-time optimizations on the IR, hence SpecFuzz can be applied after them

5 SPECTRE V1 VULNERABILITY IN ICP

Indirect Call Promotion (ICP) is an optimization technique that can improve performance by reducing the indirect branch misprediction penalty [4]. By applying the optimization, an indirect branch is transformed into a software construct, including direct calls to most-frequently-used targets of the indirect branch. The possible targets are usually collected via profiling.

Listing 1 shows an example where fun_ptr is a function pointer implemented with an indirect call. A profiler discovers that the common target addresses are of the functions foo() and bar(). The code is modified to call these targets directly, whereas conditional branches find the correct target at runtime. In case the target is not one of the statically instantiated ones, an indirect call is invoked as a fallback.

```
//  Function pointer
fun_ptr(x);
//  ICP
if (fun_ptr == foo)
    foo(x);
else if (fun_ptr == bar)
    bar(x);
else
    fun_ptr(x);
```

Listing 1: Indirect Call Promotion

This construct, however, is vulnerable to Spectre V2. The mitigation replaces the indirect branch with a Retpoline.

On the other hand, the introduced conditional branches make the code vulnerable to Spectre V1. As a result, an attacker may affect the control flow integrity and guide the execution into any of the branches, in which there could be access to secret data, which it then can attempt to leak.

We note that ICP with Retpoline has been introduced by PIBE[17] and Jumpswitches[2] to reduce the Retpoline performance overheads. However, both these proposals suffer from the Spectre V1 vulnerability above.

5.1 Eliminating Vulnerabilities with SLH

Whole program SLH is the immediate solution for mitigating Spectre V1 vulnerabilities caused by vulnerable dynamic optimizations such as ICP. However, adding SLH to the optimized code might result in higher overheads, potentially nullifying the benefits of the optimization and having an adverse effect on the performance.

To see whether the dynamic optimization is still worthwhile even with Spectre V1 mitigation, we choose four C/C++ workloads, including OpenSSL, JM19.0, Xerces-C++, and Xalan-C++, characterized by frequent indirect calls (i.e., many opportunities for the ICP optimization). For OpenSSL, we use its internal measurement of signing rate with the standardized NIST P-224 algorithm. The rest use workloads from the Alberta Workloads[1], where, for JM19.0,

	Speedup
Xerces-C++	1.33
Xalan-C++	1.84
JM19.0	1.05
OpenSSL	1.38

Table 2: Speedups of ICP optimization in mitigated workloads

	1	2	3
Xerces-C++	0.96	0.99	0.99
Xalan-C++	0.96	0.98	0.99
JM19.0	0.38	0.86	0.87
OpenSSL	0.50	0.87	0.93

Table 3: Cumulative distribution of call targets per call site

we specifically use the decoder ldecod.exe, and the sample application DOMCount for Xerces-C++.

Table 2 shows speedups achieved by using ICP on an SLH-hardened build with -O3 flag, and Retpolines. In all these benchmarks, ICP is profitable even with SLH. Thus we conclude that using a Spectre-V1 - protected version of ICP is still beneficial.

5.2 Eliminating Vulnerabilities with the Post-Optimization SpecFuzz

The previous approach used static SLH hardening. Now we consider an alternative method to mitigate Spectre V1 by using SpecFuzz.

We apply SpecFuzz on a whole program *after running all dynamic optimizations* to protect all potentially vulnerable branches added by them.

The baseline is a compilation with -O3 flag and Retpoline (referred to as *native* in the graphs). We then add *dynamic* optimization, including the Profile Guided Optimizations (PGO) and Link Time Optimizations (LTO) in LLVM.

Each compilation configuration uses different types of Spectre V1 mitigation: unprotected, with static whole-program SLH, or SpecFuzz. The speedups are normalized to the native, unprotected version (first bar for each application).

The results are presented in Figure 1. We observe that in some cases, SpecFuzz offers substantial benefits over SLH, particularly in combination with dynamic optimizations. For example, JM19.0 shows 30% higher performance with SpecFuzz over SLH when used in conjunction with the dynamic optimizations but no benefit in the unoptimized version.

6 MITIGATION-AWARE OPTIMIZATION HEURISTICS

In this section, we show two examples in which mitigation-aware optimization heuristics improve the generated code performance.

6.1 Revisiting ICP

The ICP heuristic considers the overheads related to promoting call targets. Specifically, the heuristic determines which call targets need to be transformed into direct calls among all the discovered targets. The higher the frequency of the call, the higher the chances it will be transformed into a direct call.

However, ICP does *not* consider the extra overhead of hardening the conditional jumps associated with each direct call target as part of the ICP construct. This overhead biases the promotion to higher call frequency by making less frequent targets non-profitable.

We evaluate several frequency values and report the best one discovered. Our heuristic promotes the most likely target if the probability of it being called exceeds 50% and the second most likely if it exceeds 35%. Otherwise, it spares the extra overhead in the ICP construct. For comparison, in a worst-case scenario, the default heuristic would promote the most likely target for any probability above 30% and the second most likely for a probability of at least 21%.

Results are presented in Figure 2. The baseline is the performance of the default heuristic (first bar in each bar pair). We observe that the new heuristic performs worse than the default one for the vulnerable code and improves the performance of the mitigated code.

JM19.0 stands out in that it enjoys an exceptionally high speedup of X1.31 for the mitigated version. Table 3 sheds light on a difference between the workloads that explains these results. The statistics of indirect call executions derived from the dynamic analysis show that many call sites are candidates for promoting no more than a single target with 96% in Xerces-C++ and Xalan-C++. The distribution in the JM19.0 workload gives more weight to multiple targets, which guides the heuristic to promote multiple targets for call sites more often.

6.2 Integrating SESES Mitigations With Unrolling Optimization

`LFENCE`s are used in LVI mitigations for hardening memory accesses and speculative branches. The overhead of such mitigation is substantial because the hardening affects frequently executed instructions.

We claim that the cost model that LLVM applies to determine whether to perform optimizations or not should be modified to accommodate the introduction of these serialization instructions. To achieve this result, we modify the heuristic to minimize the number of executed memory accesses and branches.

While LLVM natively supports many related passes such as vectorizer, inliner, loop unroller, register allocator, elimination of redundant branches and loads, we showcase the implications of our observation on the loop unrolling alone. A naive solution would be to increase the default threshold of the loop unroller, thus, merging more loop iterations and thereby reducing the number of executed conditional jumps in the loop control logic.

The problem, however, is that the unroller optimization is applied in the middle of the optimization pipeline, and the change causes later passes to perform poorly if the naive solution above is applied. For instance, merging two iterations might make the register allocator heuristic spill more than twice the number of registers, causing many more memory accesses. Thus, an LFENCE instruction removed with a conditional branch due to unrolling

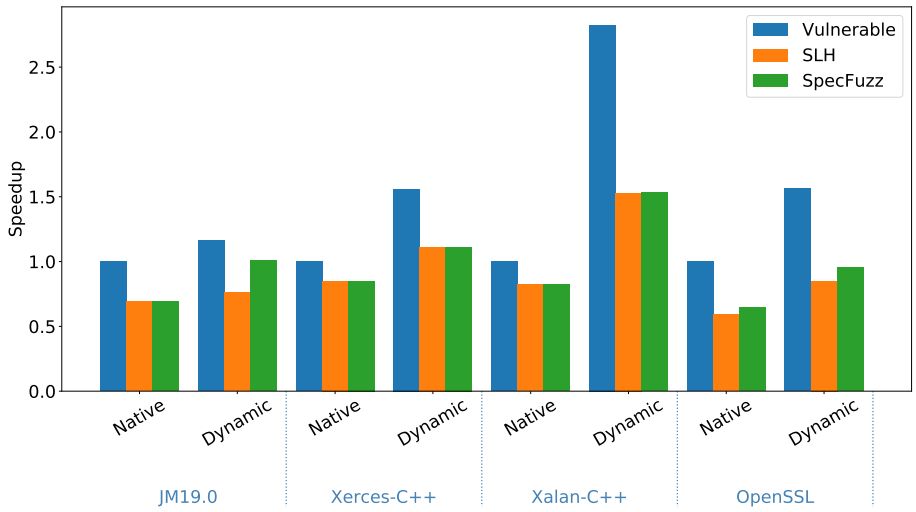

Figure 1: Speedups in different builds with varying Spectre V1 mitigations

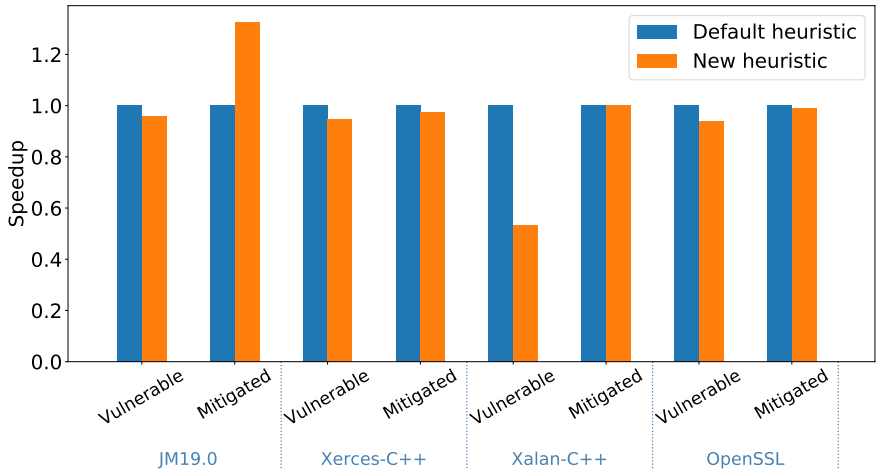

Figure 2: Speedups in Spectre V1 vulnerable and mitigated builds using different ICP heuristics

grows into multiple such instructions associated with the additional memory accesses due to the excessive register spillage.

To solve this problem, we leave the native unroller unmodified and create an *additional* loop unroller at a later pipeline stage after the register allocation. This way, we can guarantee no interference with the following pipeline passes. The passes would have better optimization opportunities if we did modify the native unroller. However, it would require modifying many optimization and analysis passes to take advantage of the new opportunities and avoid performance decrease.

The unrolling is done by duplicating loop body instructions and patching the necessary branch targets. As a result, we skip loop control $F - 1$ times for unrolling factor F every iteration of the unrolled loop. Avoiding excessive register spillage becomes trivial since register allocation was already done.

Some loops have a dynamic trip count, which is unknown at the compile time. In this scenario, the loop can still be unrolled by some factor F. The remainder of the division of the trip count by F would be executed in another loop, called a remainder, which is a duplication of the original loop. The increased code complexity of

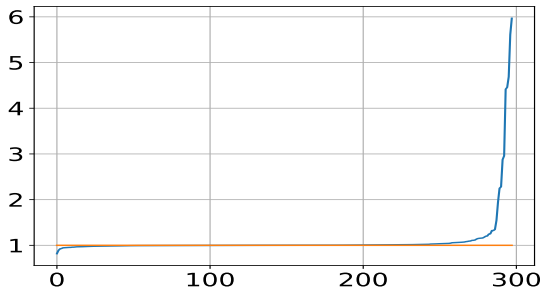

Figure 3: Speedups thanks to the mitigation-aware loop unroller over the native SESES build, for ~300 standard benchmarks in the LLVM Test Suite

adding a remainder loop might make the runtime loop unrolling not worthwhile if the unrolled loop is not being entered enough times. Thus, we use profiling to decide the unrolling factor - the factor will not be greater than the average trip count.

The native loop unroller uses a heuristic with a parametric threshold to determine the unrolling budget, meaning the loop is being unrolled only by a factor that would keep the cost within budget. We follow the same heuristic but increase the threshold.

Evaluation. To evaluate the performance of the mitigation-aware loop unroller, we test it with the LLVM Test Suite. We enable the SESES mitigation and Profile Guided Optimizations (PGO) for the baseline. Figure 3 presents the speedups due to the new heuristic for each of the ~300 benchmarks. Results show that the new unroller *distinctively* improves performance, and often by a significant factor, with speedups of up to X5.97.

7 CONCLUSION

We foresee a new toolchain that achieves mitigation-aware compilation, as can be seen in Figure 4. Ultimately, the unified Spectre V1 and Spectre V2 mitigation method can be generalized into a code generation pipeline that should include mitigation-aware optimizations and properly ordered mitigations. The compilation process leverages program semantics and speculative execution mitigation knowledge to guide optimization and mitigation choices. Static analysis and dynamic optimizations can help achieve better performance while reducing the overheads of software mitigations.

Our work argues that security should be integrated holistically into the compiler, considering inter-dependencies and changes in the cost-benefit model for building efficient and secure programs.

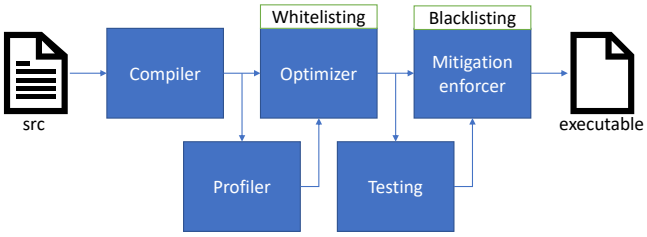

Figure 4: The envisioned mitigation-aware compilation pipeline

REFERENCES

[1] José Nelson Amaral, Edson Borin, Dylan R Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. 2018. The alberta workloads for the spec cpu 2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 159–168.

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

[3] Ivan Baev and Qualcomm Innovation Center. 2014. Controlling Virtual Register Pressure in LLVM Middle-End. In *LLVM Developers Meeting, Oct.*

[4] Ivan Baev and Qualcomm Innovation Center. 2015. Profile-based indirect call promotion. In *LLVM Developers Meeting, Oct.*

[5] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1046–1060.

[6] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).

[7] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.

[8] Zola Bridges. 2020. [x86][seses] Introduce SESES pass for LVI. https://reviews.llvm.org/D75939.

[9] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 769–784.

[11] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. 2020. Evolution of defenses against transient-execution attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 169–174.

[12] Chandler Carruth. 2018. Speculative Load Hardening: A Spectre Variant 1 Mitigation Technique. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0.

[13] Jonathan Corbet. 2018. Relief for retpoline pain. https://lwn.net/Articles/774743/.

[14] Intel Corporation. 2018. *Speculative Execution Side Channel Mitigations.*

[15] Intel Corporation. 2020. Processors Affected: Load Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/processors-affected-load-value-injection.html.

[16] Intel Corporation. 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*

[17] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2021. PIBE: practical kernel control-flow hardening with profile-guided indirect branch elimination. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 743–757.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[19] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1871–1885.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

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

[22] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.

[23] Michael Larabel. 2020. Google Engineer Shows "SESES" For Mitigating LVI + Side-Channel Attacks - Code Runs 7% Original Speed. https://www.phoronix.com/scan.php?page=news_item&px=LLVM-SESES-Merged.

[24] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[25] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets. (2021).

[26] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[27] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 279–299.

[28] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.

[29] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 54–72.

[30] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[31] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.

[32] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).

[33] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. (2018).

[34] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring branch predictors for constructing transient execution trojans. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 667–682.