PIBE: Practical Kernel Control-Flow Hardening with Profile-Guided Indirect Branch Elimination

Victor Duta
Vrije Universiteit
Amsterdam
Netherlands
v.m.duta@vu.nl

Cristiano Giuffrida
Vrije Universiteit
Amsterdam
Netherlands
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit
Amsterdam
Netherlands
herbertb@cs.vu.nl

Erik van der Kouwe
Vrije Universiteit
Amsterdam
Netherlands
vdkouwe@cs.vu.nl

ABSTRACT

Control-flow hijacking, which allows an attacker to execute arbitrary code, remains a dangerous software vulnerability. Control-flow hijacking in speculated or transient execution is particularly insidious as it allows attackers to leak data from operating system kernels and other targets on commodity hardware, even in the absence of software bugs. Having made the jump from regular to transient execution in recent attacks, control-flow hijacking has become a top priority for developers. While powerful defenses against control-flow hijacking in regular execution are now sufficiently low-overhead to see wide-spread adoption, this is not the case for defenses in transient execution. Unfortunately, current techniques for mitigating attacks in transient execution exhibit high overheads—requiring a costly combination of defenses for every indirect branch.

We show that the high overhead incurred by state-of-the-art mitigations is mostly due to the effect of hardening frequently executed branches. 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 the compiler's optimization passes. For every indirect branch, it decides whether to harden it with instrumentation code or elide it altogether using code transformations. By specifically removing 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. Demonstrating our solution on the Linux kernel, one of the largest, most complex and most security-critical code bases on modern systems, we show that PIBE reduces the overhead of comprehensive defenses against transient control flow hijacking by an order of magnitude, from 149% to 10.6% on microbenchmarks and from ~40% to around 6% on several application benchmarks.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446740

KEYWORDS

transient execution, control-flow hijacking, profile-guided optimizations, operating systems

ACM Reference Format:
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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3445814.3446740

1 INTRODUCTION

Control-flow hijacking attacks have ranked among the most dangerous forms of compromise for decades. Now that attackers have transitioned to control-flow hijacking in the transient execution domain, they have become particularly worrying. While powerful and efficient defenses to mitigate such attacks exist for regular execution (e.g., in the form of fine-grained control-flow integrity or stack-protector-strong), this is not the case for control-flow hijacking during transient execution. Modern high-performance CPUs rely on sophisticated mechanisms that predict control and data flow, to speculatively execute instructions (possibly even out-of-order) and keep their pipelines full. If predictions turn out to be correct, instructions are retired to make their outcomes architecturally visible. Otherwise, the CPU flushes the pipeline to roll back architectural effects of pending instructions and guarantee functional correctness, but *non-architectural side effects sometimes remain.* The execution life cycle of instructions prior to their retirement is often referred to as transient execution [9]. Transient execution vulnerabilities [8, 20, 24, 28, 31–33] allow attackers to leak sensitive data (e.g., from the OS kernel), even in the absence of software bugs. Control-flow hijacking attacks during transient execution are noteworthy because defenses such as retpolines [30], return retpolines [18], and Load Value Injection (LVI) related control-flow hardening [10] are not cheap even by themselves and full mitigation in software requires protecting every indirect branch with all of them simultaneously. As a consequence, few administrators apply such combined defenses, even if it leaves their systems vulnerable.

Are the high overheads inherent to the defenses, or artifacts of the way we generate code? We show that we can avoid much of the overhead by reconsidering past design choices in code generation to defend against all forms of control-flow hijacking with low overhead. We view a hardening pass that adds instrumentation to a program as a cost-benefit optimization game where, for each instruction to be instrumented, the compiler selects one of two possible moves: it either instruments the code or attempts to transform

it to remove the target instruction altogether. Each move has its own cost or benefit in terms of performance. For instance, a code transformation that elides a target instruction may remove all the overhead of the instrumentation itself, but if it increases code size, it may still reduce overall performance through less efficient cache usage. In that case, it may be better to apply the transformation sparingly, limiting it to the code's hot paths.

Since control-flow hijacking relies on the manipulation of targets of indirect branch instructions (indirect jumps, calls, and returns), we focus exclusively on hardening such branches. 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 due to 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 defenses in the kernel also affects the whole system, making administrators reluctant to deploy them. Finally, given that our approach requires a profiling workload, kernels are ideal targets, as companies such as Google already maintain representative profiling workloads to enable profile-guided optimizations for their production kernels [26]. We emphasize, however, that our approach applies equally to other code: hypervisors, SGX(-like) enclaves, and user programs.

We present PIBE, a solution to harden indirect branches at a fraction of the cost of existing solutions by (a) aggressively reducing the *number* of vulnerable indirect branches in frequently executed code through *profile-guided indirect branch elimination*, and (b) applying a range of state-of-the-art defenses to the remaining ones. The key idea behind PIBE 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.

Unlike existing approaches, PIBE makes comprehensive defenses against control-flow hijacking practical in performance for both the forward edge (indirect calls and jumps) and backward edge (return instructions), while requiring no complex run-time modifications of kernel code. Moreover, we show that the performance is robust with regard to varying workloads, allowing it to be applied even by vendors of end-user binary software distributions, using a predetermined profile that is not necessarily identical to that of the end user.

Contributions. We make the following contributions:

- A comprehensive security analysis of all common transient control-flow hijacking defenses and their performance implication in userspace and the kernel;
- 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% on microbenchmarks and from ~40% to around 6% on several application benchmarks.

PIBE's source code is available at https://github.com/vusec/pibe.

2 BACKGROUND

2.1 Control Flow Hijacking

In control flow hijacking, attackers gain control over the program counter to execute code of the attacker's choosing. In addition to traditional control flow hijacking, recent attacks [20, 24, 32] abuse hardware vulnerabilities to allow an attacker to control the program counter during *transient execution* even in programs that themselves contain no software bugs. For performance reasons, whenever the CPU encounters an indirect branch, it predicts its target and continues to execute the code at the expected address speculatively. If the prediction turns out to be incorrect, the CPU reverts the changes that were made transiently, and continues execution at the correct address. However, some microarchitectural state is not reverted and can be used to leak information from the transient execution. For example, caches retain data speculatively loaded from memory. By timing subsequent memory operations, attackers can observe such state indirectly and leak sensitive data. While these vulnerabilities should ideally be fixed in hardware, microcode updates to protect existing hardware incur prohibitive performance overhead (e.g., 25–53% for Spectre V2 mitigation alone [2]). Efficient hardware mitigations require an overhaul of the microarchitecture itself, leaving billions of devices vulnerable[14]. Available software defenses against transient control flow hijacking attacks include retpolines [30], return retpolines [18], and LVI-CFI [10], but these defenses incur high performance overheads and are rarely (comprehensively) deployed. Recently, Canella et al. [7] specifically clamored for more efficient LVI defenses.

2.2 Transient Execution Optimizations

Transient variants of control flow hijacking attacks target several CPU optimizations designed to predict control flow, allowing speculative execution of the most likely path. These variants are described below:

Branch Target Buffer. The BTB is a fixed-size microarchitectural buffer used to predict the targets of indirect or conditional branches. For an indirect branch, the BTB indexes the likely target of the branch using the least significant bits of the branch address. Multiple indirect branches may alias to the same BTB entry and BTB predictions are shared across processes running on the same core (even when they run at different privilege levels). Transient attacks that target the BTB (e.g., Spectre V2) typically poison the BTB with malicious destinations to trick the CPU into speculatively executing the target gadget when it runs a victim indirect branch that aliases to poisoned BTB entries.

Return Stack Buffer. The RSB is a small per-core microarchitectural buffer storing the return addresses of the N most recent *call* instructions (typically $N = 16$). When encountering a *ret* instruction, the CPU pops the last entry from the RSB to predict the return flow. The RSB causes misspeculation when the address in the RSB does not match the return address from the software stack. Transient attacks that target the RSB (e.g., Ret2spec [24], SpectreRSB [21]) aim to desynchronize the RSB and software stack to cause a return to misspeculate to an adversarial gadget. Techniques to achieve this effect include: direct pollution of the RSB (by explicitly overwriting the return address on the software stack);

speculative pollution of the RSB (RSB entries pushed by speculatively executed calls are not reverted on a pipeline flush); RSB reuse across execution contexts (on a context switch the newly scheduled thread reuses the RSB entries left by the previous thread); programming constructs that break call-ret semantics (e.g., setjmp/longjmp); and overfill or underfill of the RSB (relevant to CPUs that speculate through the BTB on an RSB underflow).

Memory Order Buffer. The MOB is equipped with various prediction and resolution circuits used to predict data dependencies between stores and subsequent loads. If a store-to-load dependency is detected, the MOB forwards the stored data to the dependent load. However, if the data dependency is mispredicted, the load may consume either stale data or wrong data from the MOB's internal buffers. Transient control flow hijacking attacks like LVI [32] poison the MOB buffers such that faulting loads caused by *call* and *ret* instructions may pick up attacker-controlled branch targets from the poisoned microarchitectural buffers.

Control flow hijacking attacks that target these microarchitectural optimizations are major threats for which no practical mitigation exists. In Section 6, we provide an in-depth security analysis of the defenses available for these attacks and propose improvements to make practical defenses possible.

2.3 Profile-Guided Optimizations

Optimizing compilers have long focused on generating faster code. Many compiler optimizations are beneficial in any context. A typical example is constant folding, which effectively moves computation from runtime to compile time. Some optimizations, however, can either speed up or slow down the program, depending on the workload. A typical example is inlining, which takes a callsite and replaces it with the function body of the callee. Doing so removes function call overhead and allows for further optimizations that would not be performed interprocedurally but also increases the code size. Increased code size reduces locality and fills up caches and may well *slow down* the program. As such, compilers are usually conservative and avoid inlining for all but the smallest functions, assuming the benefits may not be worth the cost.

Modern compilers support profile-guided optimizations (PGO), where the compiler uses information about execution patterns to perform those optimizations that are most beneficial for a particular workload. For example, when deciding whether to perform inlining, the compiler can use this information for a cost/benefit estimation and make an informed trade-off. Traditionally, profile-guided optimizations are applied in two phases. First, a profiling run collects various execution statistics for a target program. The resulting profile allows the compiler to determine the hot and the cold parts of the program. Afterwards, a second compilation uses the information to make better code-generation decisions.

In addition to inlining, PIBE builds on indirect call promotion, which applies to indirect call sites, where the callee is not known at compile time. Profiling information allows the compiler to determine which functions are common targets for a call site and add a conditional check to call those targets directly rather than through an indirect call. This not only allows the CPU to better predict which function will be called but also provides more opportunities for inlining. We will discuss in Section 5 how we adapt these general approaches to improve performance for defenses against control flow hijacking.

3 THREAT MODEL

PIBE focuses specifically on control-flow hijacking attacks in the *transient* domain, for which current defenses are too expensive. While our solution also improves their performance somewhat (but not much), we deliberately do not evaluate defenses against non-transient attacks due to bugs in the source code, since practical defenses exist already (see Section 6). For the same reason, we do not target Spectre V1, as static analysis already provides a practical solution for the kernel [13]. Finally, we do not consider inline assembly, which is relatively rare, even in the kernel, as it may make assumptions incompatible with the ABI, making automatic instrumentation unsafe.

4 OVERVIEW

PIBE is fully integrated in the LLVM pipeline and operates in two phases. In the first phase, it instruments the program for profiling. In the second phase, it uses highly tailored profile-guided optimization to reduce the number of indirect branches in frequently executed code, while selectively applying a range of state-of-the-art defenses to the remaining indirect branches. Accordingly, PIBE generates two versions of the target program: a profiling and a production binary. A profiling binary runs a representative workload and collects profiling information as input for the code transformations for indirect branch reduction and hardening for the production binary.

In the profiling pass, PIBE instruments each function entry point and call site with monitoring code to track the frequency of direct and indirect calls, maintaining a counter for each edge in the call graph. After execution, PIBE returns these statistics in an LLVM-IR friendly format that allows them to be mapped back to the corresponding kernel code, even in the presence of code changes due to optimization and randomization.

In the hardening pass, PIBE enforces arbitrary combinations of defenses against transient control flow diversion and currently supports the state-of-the-art solutions for all known attacks: retpolines [30], LVI-CFI [10] and return retpolines [18]. After matching the statistics collected by the profiling binary with the call graph, PIBE selectively applies indirect call promotion and/or inlining—using the statistics to eliminate the hottest indirect branches. Finally, it hardens the remaining branches using the requested defenses.

5 MAKING DEFENSES PRACTICAL

Control flow hijacking attacks manipulate the program counter—a register only written to explicitly in branch instructions. Since direct branches with a fixed target allow no control over the value written to the program counter, control flow hijacking critically depends on *indirect* branch instructions. Indirect branches read their targets from a register or memory, so attackers can use a vulnerability to modify them, transiently or nontransiently, to divert control flow. To prevent such attacks, we must protect indirect branches.

Our approach to protect indirect branches at low cost is to reduce the number of indirect branches, using profile-guided indirect branch elimination, and then protect the remaining ones by a combination of state-of-the-art defenses. In this section, we describe

how PIBE revisits profile-guided (PGO) techniques to aggressively reduce the number of indirect branches in hot code. In Section 6, we analyze the security of defenses against control flow hijacking and consider how PIBE improves their performance.

5.1 Indirect Branches

On most CPU architectures, indirect branches come in three flavors: indirect calls, indirect jumps, and returns. Indirect calls correspond to function pointers and virtual functions and lead to control-flow hijacking, for instance, when an attacker uses a vulnerability to alter the target. We eliminate performance-critical indirect calls using indirect call promotion and protect the remainder using state-of-the-art defenses.

Indirect jumps typically result from optimizations rather than language constructs. In particular, compilers may convert indirect calls into indirect jumps for tail calls where the callee returns directly to its caller's caller. This case is covered by the same mechanisms as for indirect calls. The other sources of indirect jumps are jump tables, which compilers introduce as an optimization for multiway jumps (typically, switch statements). Jump tables include a bounds check that forces target addresses to be picked from valid table indices and are not typically target for nontransient attacks. Transient attacks can bypass this check. To protect against jump table hijacking under transient execution, PIBE disables jump table generation in the compiler—the default LLVM behavior when retpolines or LVI defenses are enabled and the approach also adopted by JumpSwitches [2].

Finally, return instructions signal the end of a function. They load the return address from the stack and jump to it. Returns tend to be the most common indirect branches by far, as almost every function ends in a return instruction. An attacker can abuse this instruction by altering either the return address stored on the stack or the stack pointer pointing to the return address. We elide the most performance-critical indirect calls using inlining (eliminating both the call and its corresponding return instruction).

5.2 Inlining

Inlining replaces a call site with the body of the callee, while replacing the formal parameters with their actual values (Listing 1). The main goal of inlining is to create additional opportunities for optimization, such as constant propagation, dead code elimination, or loop vectorization, which are hard to do across functions. Given modern branch predictors, the benefit of removing the call/return pairs themselves tends to be small. Inlining is a trade-off, as it increases the code size, fills up the instruction cache and reduces locality. Therefore, compilers today use *heuristics based on the expected potential for additional optimizations, by inlining only very small functions.*

In contrast, PIBE uses inlining for security. In particular, inlining removes backward edges, which are undesirable to have at runtime, as their prediction can be poisoned by attackers, leading to control flow hijacking. Since return instructions that are not eliminated must be protected against transient attacks using costly defenses, we need a new algorithm to inline functions solely to reduce the number of runtime calls—a different goal from that of traditional compiler inlining. Note that simply inlining all non-recursive calls

```
void callee(
  int value) {
  x = value;
}
void caller(void) {
  callee(42);
}
```

```
callee:
  mov    %edi, x
  retq
caller:
  mov    $42, %edi
  call   callee
  retq
```

```
caller:
  movl   $42, x
  retq
```

Listing 1: Inlining example, assembly code before and after inlining on the right-hand side

is not a viable solution due to the excessive increase in code size, and selecting one call site to inline might disqualify others for this reason.

To remove as many indirect branches as possible from hot paths, we use a greedy approach that optimizes indirect branches from the most frequently executed to the least frequently executed. Doing so also decreases the chance that hot call sites will be blocked from being inlined due to earlier inlining of colder call sites. As mentioned, inlining has diminishing returns as the increase in code size eventually causes a net decrease in performance. Moreover, unlike regular code, a kernel has many entry points and subsystems. The most common entry points are system calls, which are exercised independently by user processes, some far more often than others. The workload imbalance complicates the selection of an optimal threshold beyond which to stop inlining, as no threshold will lead to an uniform decrease in overhead for all kernel paths.

To elide as many call sites as possible without harming performance, PIBE's inlining strategy is governed by three simple rules: (1) inline only the hot call sites, (2) inline only those calls that do not cause excessive complexity in the caller, and (3) inline only those calls for which the callee has an insignificant impact on the caller's complexity budget.

Rule 1: Inline only hot call sites. We determine the hot call sites by setting an optimization budget that represents a percentage of the cumulative execution count. For example, a budget of 99% will attempt to inline all of the hottest code that together represents 99% of the execution counts found while profiling. At the begining, we greedily select all targets that fit in this budget. Then, at each step we attempt to inline the hottest remaining call site. After inlining a function f with execution count ϵ, we also add its callees to f's caller. To include these sites, we heuristically assign them an execution count equal to the one they had in function f, multiplied by the ratio r between ϵ and f's invocation count. This simple heuristic provides good results in practice and helps us keep track of new hot sites obtained through inlining operations.

Rule 2: Avoid excessive complexity in the caller. Merging too many nested calls into the same function may lead to poor stack frame utilization. In particular, a long chain of nested call sites can make it difficult for stack coloring algorithms to merge stack allocations. As such, hot functions may end up allocating a significant portion of the stack with each invocation but using only a small

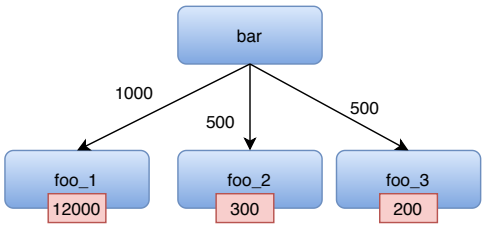

Figure 1: Example function where inlining heuristic 2 fails

fragment on each execution. Therefore, we do not inline a call site if the complexity of the caller exceeds a threshold determined experimentally. We measure the added complexity of inlining a call site using LLVM's built-in *InlineCost* analysis. The analysis computes a numerical cost heuristic for each instruction in the callee, and returns the sum of the instruction costs. Most instructions incur a standard cost, while some have specific costs assigned to them. On *X86* architectures the standard cost of an instruction is 5, which is perhaps used as an approximation for the average binary instruction size. For example, a nested call instruction, is assigned cost $5 + 5 * num\_args$. Intuitively, this is because one needs on average *num_args* extra assembly instructions to set up the arguments of the call plus the call itself.

Rule 3: Inline calls that do not impact the complexity budget. While Rule 2 prevents us from creating inefficient functions due to excessive inlining, it may unintentionally inhibit beneficial inlining. Figure 1 shows an example. The caller `bar` has three callees: `foo_1`, `foo_2` and `foo_3`. The execution counts are annotated on each edge, while the cost of inlining is marked in red bellow each callee. Using our first two rules, the greedy inliner will first select the call to `foo_1` as an inlining candidate. However, after inlining this call, we already depleted `bar`'s complexity budget. If, instead, we inline `foo_2` and `foo_3`, and skip `foo_1`, we would eliminate the same number of execution counts with enough budget left for more inlining in `bar`. For this reason, we avoid inlining a callee if its complexity is above a second (lower) threshold. We found that this combination of heuristics effectively reduces the number of backward edges to be defended, while still producing efficient code without excessively increasing the image size (e.g., 5–30%, depending on the budget).

Selecting the thresholds. We experimentally determined the *Rule 2* threshold by comparing against LLVM's PGO inlining algorithm. In particular, to determine a near-optimal value for the threshold we started with a value of 3,000 (LLVM's inhibitor threshold for hot branches) and increased the parameter (in steps of +3,000) until no noticeable improvement could be observed. We used the following metrics to evaluate improvement: (i) performance improvement over LLVM's PGO algorithm when optimizing an image instrumented with LVI with each algorithm (evaluated on the LMBench micro-benchmarks) and (ii) performance stability with increasing optimization budgets—the main drawback of LLVM's approach is that the inlining order is irrespective of profiling weight, which leads to colder calls inhibiting more beneficial inlining. We observed that LLVM's inliner has fluctuating performance while increasing the optimization budget (because raising the budget enables more cold sites as inlining candidates). Based on these metrics, we arrived at the value of 12,000 for *Rule 2*. We added *Rule 3* after observing signs of cache contention (e.g., some LMBench microbenchmark results showed increased standard deviation over 11 benchmark iterations). This may happen because inlining large functions impacts caching behavior. For Rule 3, we selected the default LLVM threshold of 3,000.

```
void caller(
  void (*callback)
  (void)) {
  callback();
}
```

```
caller:
  callq   *%rdi
  retq
```

```
caller:
  mov   $callee,%rax
  cmp    %rax, %rdi
  je     direct
  callq  *%rdi
  jmp    end
direct:
  callq  callee
end:
  retq
```

Listing 2: Indirect call promotion example (ICP). Assembly code before and after ICP on the right-hand side when profiling shows `callee` to be the most common target.

5.3 Indirect Call Promotion

Inlining is only possible when the target function is known. Indirect calls cannot be inlined. Unfortunately, eliding such calls is especially important, because doing so removes both a forward edge and a backward edge (and their instrumentation). To reduce the number of indirect calls, we use indirect call promotion. Indirect call promotion uses profiling information to determine the most common target(s) for an indirect call site and then adds conditional direct calls to those targets. The indirect call site itself remains as a fallback in case the runtime target has not been promoted. Listing 2 shows an example.

Our algorithm identifies all indirect calls for which profiling information is available. We again set a budget as the percentage of the cumulative execution count and apply a greedy algorithm that promotes the hottest targets first. Unlike existing indirect call promotion algorithms, we do not limit the number of targets to promote from a single indirect call. After all, for costly instrumentation such as LVI-CFI or retpolines, target checks are far less expensive (~2 clock cycles) than a slow path that inhibits prediction (e.g., a retpoline takes ~21 clock cycles). In other words, more checks will not be prohibitive to performance.

6 SECURITY ANALYSIS

We now consider the various classes of vulnerabilities that allow control-flow hijacking attacks against the kernel. We consider the mitigations, why the mitigations are performance bottlenecks and how they can benefit from our code transformations to yield performance that allows for deployment in practice.

Table 1: Overhead of control flow hijacking mitigations in clock ticks per direct (dcall), indirect (icall), and virtual function call (vcall), and geometric mean overhead on SPEC CPU2006. Results marked with an '*' were incomplete as LLVM-CFI failed at runtime for 453.povray and 456.hmmer, while 447.dealII failed to compile with safestack.

	dcall (ticks)	icall (ticks)	vcall (ticks)	cpu2006 % slowdown
uninstrumented	0	0	0	0.0
LLVM-CFI	2	3	1	*-0.4
stackprotector	4	4	4	1.0
safestack	2	1	1	*0.6
LVI-CFI	11	20	23	29.4
retpolines	1	21	21	16.1
retpolines + LVI-CFI	14	53	54	44.3
return retpolines	16	16	16	23.2
all defenses	32	73	71	62.0

Table 1 shows the overhead for state-of-the-art mitigations against (normal and transient) control flow hijacking on an Intel Core i7-8700 CPU running Ubuntu 18.04 with Clang 10.0, with support for return retpolines [18]. In each case, the function called is empty, all memory used has been preloaded into the cache, and except for the direct branch, we made sure the branch target is unpredicable for both the compiler and the CPU. The table shows that all defenses against transient control flow hijacking (LVI-CFI, retpolines, and return retpolines) incur high overheads, making them unattractive for pervasive deployment. In contrast, defenses against nontransient attacks (LLVM-CFI, stackprotector, and safestack) incur minimal overhead, justifying our focus on defenses against transient attacks. Even so, in principle our approach applies equally to other defenses, and would for instance be useful in more precise high-overhead research defenses such as path-sensitive CFI [16]. We discuss transient attacks and defenses in more detail in the remainder of this section.

6.1 Spectre

The Spectre vulnerability [20] concerns a range of issues whereby modern CPUs may leak information over covert channels, triggered by transient execution. To keep their pipelines filled, CPUs speculate past conditional or indirect branches by using various heuristics to predict the target that follows such an instruction. If the prediction is correct, the instructions are retired, and execution proceeds normally. If it is incorrect, the CPU rolls back the transiently executed instructions. However, the CPU does not roll back all microarchitectural state. For instance, data that was loaded into the cache due to transiently executed instructions remain there. By carefully measuring memory access times, attackers can determine which memory addresses were loaded into the cache and leak sensitive information. Moreover, an (unprivilged) attacker can mislead the CPU into consistently mispredicting particular branches, giving them control over what will be leaked. Spectre affects all programs running on a vulnerable CPU, including sensitive software such as kernels and secure enclaves.

```
if (index < size) {
  ptr = data[index];
  value = *ptr;
}
```

Listing 3: Typical Spectre V1 gadget

There are several variants of Spectre. They all use side channels to leak information past mispredicted branches but involve different predictors such as the Pattern History Table (Spectre V1), the Branch Target Buffer (Spectre V2), Return Stack Buffer (Ret2spec), and Store To Load (Spectre V4) [9]. We now discuss mitigation of the different variants.

Spectre V1. Allows attacks against conditional branches [20]. The CPU's Pattern History Table (PHT) keeps track of whether conditional branches were recently taken or not taken, and is used to transiently execute the most likely path following a conditional branch. A Spectre V1 attack poisons the PHT to force a misprediction, which causes the CPU to transiently execute the path not taken. A typical attack targets a bounds check followed by an array access as shown in Listing 3. The code is vulnerable if `index` is under attacker control, as the attacker can poison the PHT to make the CPU predict that the index is in bounds and transiently access the pointer loaded out-of-bounds of the array. Using specially crafted `index` values, an attacker can leak arbitrary memory through cache side channels. While Spectre V1 is a serious threat, few conditional branches are suitable gadgets, and static analysis [13] can identify and protect them efficiently. We conclude that Spectre V1 is not an interesting target for PIBE.

Spectre V2. Targets indirect call and jump instructions [20]. Prediction for these instructions is based on the Branch Target Buffer (BTB), which keeps track of likely target addresses. An attacker can poison the BTB, causing the CPU to mispredict the target and transiently execute the code at the address inserted into the BTB.

To defend against Spectre V2 on vulnerable hardware, retpolines [30] ensure that speculation on the branch target does not lead to arbitary control flow diversion. Listing 4 shows an example. The compiler loads the target address into a register (in this case `%r11`) and replaces the indirect call with a call to the retpoline. The retpoline uses a return instruction rather than an indirect call, which uses the return stack buffer (RSB), a small hardware stack, rather than the BTB for prediction. The retpoline performs a call to place the return address `loop` in the RSB, and immediately replaces the return address on the normal stack with the indirect branch target. Therefore, the CPU speculatively executes an endless loop using the RSB entry, irrespective of any poisoning attempts, until finally the real target is resolved.

As we saw in Table 1, retpolines take much more time to execute than normal indirect calls with significant overhead on SPEC CPU2006 (16.1%) and will incur even more for code with many indirect branches. PIBE reduces the number of retpolines executed by eliminating hot indirect calls through indirect call promotion.

Ret2spec. Targets return instructions which, as we saw, use a separate branch prediction mechanism [24]. In particular, the CPU

```
        call __llvm_retpoline_r11

__llvm_retpoline_r11:
        callq   jump
loop:   pause
        lfence
        jmp     loop
        nopl    0x0(%rax)
jump:   mov     %r11,(%rsp)
        retq
```

Listing 4: Retpoline

assumes that returns match earlier calls, which it tracks in the Return Stack Buffer (RSB). With Ret2spec, the attacker poisons the RSB, causing return instructions to be mispredicted. Like Spectre version 2, Ret2spec allows arbitrary transient code execution.

To mitigate Ret2spec, return instructions must be instrumented to prevent speculation, for which we implemented "return retpolines", as recommended by Intel [18]. The approach is identical to the retpoline example in Listing 4, except that there is no need to leave a return address on the stack, and therefore we also do not need the additional call at the start. Instead, the return retpoline is inlined in the original location of the return instruction. Since the return retpoline places the top of the RSB in a known state, poisoning it is no longer an effective attack. Return retpolines incur 16 clock ticks of overhead on every function return. While the microbenchmark shows less overhead than normal retpolines, the overall overhead is higher, because return instructions are far more common than indirect calls. We measured 23.2% overhead on SPEC CPU2006. PIBE reduces this overhead, because profile-guided inlining eliminates backward edges on the hot paths.

Spectre V4. Does not directly affect branch instructions but rather memory load instructions. In particular, it uses the Store To Load (STL) mechanism to make the memory load transiently return a stale value [15]. In the context of control-flow hijacking, the most practical attack uses this value to bypass a branch. Accordingly, the same defenses discussed for the previous attacks are effective to prevent transient branches[1].

6.2 Load Value Injection

While by themselves Meltdown [23] and similar vulnerabilities (such as Foreshadow [31] and MDS [8, 28, 33]) are not relevant for control-flow hijacking, this is not true for Load Value Injection (LVI) [32]. LVI is a recent attack that allows an attacker to transiently inject any value into any load operation that triggers a fault or microcode assist. This value can be used to transiently control indirect branches or array indices, providing a powerful primitive for attackers to leak information through the cache. The attack involves no branch mispredictions, so retpolines are ineffective.

For control-flow hijacking, LVI requires hardening of memory loads in indirect branches with LFENCE instructions [32]. LLVM offers LVI-CFI [10], which implements such a defense. The defense instruments both indirect calls (Listing 5) and returns (Listing 6). In both cases, the LFENCE forces the load of the target address to complete before the control transfer. By considering direct and indirect calls in Table 1, we can see that LVI-CFI slows down backward edges by 11 clock ticks and forward edges by 9 clock ticks. As a consequence, this defense incurs 29.4% overhead on SPEC CPU2006. PIBE eliminates both forward and backward edges along the hot code paths, reducing the number of memory fences necessary to defend against LVI.

[1]While data-only attacks are theoretically also possible, they are out of scope as they do not involve control-flow hijacking

```
  call __x86_indirect_thunk_r11

__x86_indirect_thunk_r11:
  lfence
  jmpq    *%r11
```

Listing 5: LVI-CFI forward edge instrumentation

```
pop     %rcx
lfence
jmpq    *%rcx
```

Listing 6: LVI-CFI backward edge instrumentation

6.3 Combining Defenses

Both retpolines and LVI-CFI replace indirect branch instructions by a code sequence to inhibit CPU speculation. Retpolines prevent speculation on branch targets, while LVI-CFI prevents speculation on memory contents. As such, both defenses are needed to simultaneously mitigate Spectre V2 and LVI. However, both defenses instrument the same code sequences, and therefore they are incompatible. Moreover, LVI-CFI introduces an indirect jump, which is vulnerable to BTB poisoning. For a combined defense, we added a fenced retpoline sequence to LLVM, as shown in Listing 7. This code is based on the standard retpoline, while also protecting the target against LVI using an alternative, proposed by Van Bulck et al. [32], that uses a return instruction rather than an indirect jump. When using both regular retpolines and return retpolines, the combined defense incurs 32 cycles of overhead on backward edges and 42 on forward edges. This comprehensive defense against transient control flow hijacking slows down SPEC CPU2006 by 62.0%, clearly showing the need for our code transformations. However, we emphasize again that our approach is not limited to these defenses and applies to all defenses that have high overheads.

6.4 The Kernel Transient Threat

Retpolines are the standard Spectre V2 defense for the kernel, given that, on most X86 CPU architectures software mitigation is faster than existing hardware mitigations (e.g., IBRS, STIBP) [22, 29]. In recent hardware (e.g., Intel Cascade Lake) Enhanced IBRS (eIBRS) can be enabled to replace *retpolines*, but the hardware mitigation has limitations and does not prevent attacks that train on kernel execution [19]. Prior research [21] shows that an attacker can leverage

```
        call __llvm_retpoline_r11

__llvm_retpoline_r11:
        callq   jump
loop:   pause
        lfence
        jmp     loop
        nopl    0x0(%rax)
jump:   mov     %r11,(%rsp)
        notq    (%rsp)
        notq    (%rsp)
        lfence
        retq
```

Listing 7: LVI-protected retpoline

user code to pollute the RSB and trigger a Ret2spec-like vulnerability in the kernel. RSB refilling limits the attack surface, defending against known userspace-to-kernel RSB attacks. However, other RSB exploitation scenarios (see Section 2.2) are still possible under RSB refilling. Conversely, *return retpolines* defend against all known RSB poisoning scenarios. Upon a misspeculation from the RSB, *return retpolines* will always force the CPU into an endless loop until the target is resolved. Moreover, RSB refilling was designed with a different purpose in mind (i.e., to defend retpoline returns on hardware that speculates from the BTB on an RSB underflow) and its adoption in the kernel is ad-hoc, with many processor lines (e.g., Intel Xeon, I7 CPUs prior to Skylake) left undefended. Though the LVI attack is viewed as a threat mostly for SGX enclaves, in theory, kernel attacks are also possible, as proved by Van Bulck et al. [32]. The only real obstacle for LVI kernel exploitation is inducing kernel page faults, which is possible on any OS running in a virtualized environment (e.g., the guest forces the VMM to reclaim memory pages [17]). In Section 8.5 we show that, with PIBE's optimizations, these defenses can incur overheads similar to that of an unoptimized kernel with *retpolines* (defense recommended by default in the Linux Kernel for both Intel and AMD CPUs).

7 IMPLEMENTATION

Kernel Profiling. When PIBE runs as a profiler, it adds monitoring instrumentation to the kernel. The instrumentation makes use of hardware profiling features available on modern Intel CPUs (i.e., Last Branch Record) to keep track of execution counts for each edge in the kernel dynamic call-graph, at the binary level. Furthermore, our instrumentation assigns unique identifiers to each edge, each identifier mapping back to the IR call site of the edge. After the profiling run finishes, we aggregate all the profiled edges in a list and leverage the identifiers to *lift* the binary profile to an LLVM-IR intermediate representation that allows us to remap all execution counts to their respective call graph edges during the optimization run. For direct calls we attach the execution count to their IR call site. For indirect sites, which may target multiple functions, we attach *value profile* metadata represented by a list of (target name, execution count) tuples. We recover the function name from the binary address, for each target function, when we *lift* the profile to the intermediate representation.

Prototype. We implemented a prototype of PIBE on the Linux Kernel version 5.1.0 built with the default kernel configuration. We can easily port PIBE to newer kernel versions, but this was the latest kernel on which we could safely patch JumpSwitches [2] to compare against our static retpolines optimization. To compile and link the kernel we use the tools supplied by the LLVM 10 framework. We patched the framework with the official LLVM 11 implementation of LVI's indirect call and return hardening schemes. We made modifications to kernel makefiles and build scripts such that all kernel (.c) files are linked as LLVM bitcode and converted to machine code only after applying passes to the linked code. Our changes seamlessly integrate with the compilation pipeline and can be enabled by supplying a few configuration flags during the kernel build process. PIBE's profiling instrumentation and interprocedural optimizations run on the linked bitcode as LLVM passes. To run the passes we leverage LLVM's *opt* tool.

8 EVALUATION

To evaluate the performance and security of our system, we tested it on a server equipped with an Intel i7-8700K CPU, a SanDisk SSD Plus disk, and 32GB of RAM, running Ubuntu 18.04. We run the LMBench [25] test suite in the default OS configuration, which contains LMBench's standard latency and bandwidth microbenchmarks. We report the results of the latency benchmarks throughout the evaluation. Each measurement was performed at least 11 times, and we report the median. To obtain an *exact* profiling workload for our microbenchmark experiments we run the same LMBench configuration 11 times and collect all edge execution counts observed across all 11 iterations. We chose LMBench as it is specifically designed to focus attention on the basic building blocks of many OS subsystems and is widely used to identify performance bottlenecks in OSes [6] and virtualized environments [5]. Our macrobenchmark experiments (see Section 8.5) empirically prove that this workload is representative for typical application-to-kernel interaction as well.

8.1 Performance Baseline

We first present the baselines to which we compare when evaluating PIBE. Our main baseline is a vanilla kernel, with no profile-guided optimizations and no defenses activated, obtained through PIBE's Link Time Optimization (LTO) pipeline. This baseline represents how Linux is typically deployed in real-life scenarios. However, comparing an optimized kernel with defenses deployed against this baseline masks some of the residual overhead of defending the branches that are not elided by PIBE, due to the added speedup in optimized code (e.g., the promoted targets of an indirect call are faster than when called indirectly). To show that our approach specifically speeds up the defenses, we also compare against the LTO baseline optimized using PIBE's PGO algorithms but without defenses enabled. Our PGO baseline is tuned to give the best possible performance on the LMBench test suite. In Table 2 we show the latencies of both baselines for all LMBench microbenchmarks used in our evaluation. The second column depicts the LTO baseline latency while the third column shows the latency of our optimized

Table 2: The two baselines we compare against throughout the evaluation. We show absolute latencies in microseconds. For the PIBE baseline we also show the overhead relative to the LTO baseline. (+) means slowdown while (-) means speedup.

Test	LTO Baseline	PIBE Baseline	
null	0.14	0.15	**3.4%**
read	0.2	0.18	**-6.7%**
write	0.17	0.16	**-4.5%**
open	0.78	0.64	**-17.7%**
stat	0.4	0.33	**-16.4%**
fstat	0.21	0.21	**2.7%**
af_unix	3.79	3.43	**-9.5%**
fork/exit	64.57	61.19	**-5.2%**
fork/exec	158.59	151.51	**-4.5%**
fork/shell	418.62	402.0	**-4.0%**
pipe	2.28	2.23	**-2.3%**
select_file	4.37	3.95	**-9.6%**
select_tcp	9.38	8.13	**-13.4%**
tcp_conn	8.01	7.4	**-7.5%**
udp	3.81	3.42	**-10.3%**
tcp	4.61	4.13	**-10.5%**
mmap	8.73	8.35	**-4.3%**
page_fault	0.11	0.1	**-3.5%**
sig_install	0.2	0.2	**0.1%**
sig_dispatch	0.67	0.63	**-5.6%**
Geometric Mean	-		**-6.6%**

baseline (for brevity we will refer to our second baseline as the *PIBE* baseline). The geometric mean overhead of our LTO algorithms is -6.6%, which means that our approach speeds up the kernel even if no defenses are enabled.

8.2 Performance Comparison against State-of-the-Art

Although PIBE supports comprehensive protection, the state-of-the-art JumpSwitches [2] supports only retpolines, so we focus on this technique for a performance comparison. Table 3 shows our results for a series of benchmarks that are strongly impacted by *retpolines*. The table shows the latencies of the LTO baseline fully protected with retpolines (column 2), JumpSwitches' runtime indirect call patching methodology (column 3), and our static indirect call promotion (*icp*) algorithm with different optimization budgets (columns 4 and 5). JumpSwitches and our *icp* configurations harden all remaining indirect calls with retpolines. For each test image we show the overhead of each microbenchmark relative to the LTO baseline. PIBE manages to bring the 20.2% overhead incurred by retpolines down to just 1.3%, considerably less than the 5.0% for JumpSwitches. Our performance edge over JumpSwitches may either be related to concurrency issues caused by their complicated runtime patching mechanism, as we ported their implementation on a newer kernel version than the one it was designed for or, more likely, because LMBench exercises many code paths comprising of multi-target indirect calls. For indirect calls with more than one common target, the JumpSwitch mechanism must be periodically put in a learning state, case in which the call is reconverted into a retpoline that relearns targets. This should reduce performance on code paths that access multi-target indirect calls while the calls are in learning mode. In Table 4, we show the distribution of indirect calls based on the number of targets they call, as seen in our workload. From the table we can deduce that a great portion of kernel indirect calls are multi-targeted and when defended with JumpSwitches may be periodically downgraded to learning retpolines.

Table 3: Retpolines overhead compared to LTO baseline.

Test	LTO w/retpolines *(no optimization)*	JumpSwitches *(w/retpolines)*	+icp w/retpolines *(99%)*	*(99.999%)*
null	**3.8%**	**7.9%**	**10.3%**	**9.5%**
read	**12.8%**	**0.1%**	**4.8%**	**1.1%**
write	**14.7%**	**-1.5%**	**5.7%**	**0.8%**
open	**12.3%**	**8.6%**	**-0.5%**	**0.7%**
stat	**11.9%**	**8.4%**	**2.8%**	**0.2%**
fstat	**5.4%**	**9.2%**	**8.1%**	**1.0%**
select_tcp	**146.5%**	**-10.5%**	**4.6%**	**5.8%**
udp	**18.7%**	**7.4%**	**-0.2%**	**0.4%**
tcp	**17.5%**	**13.3%**	**0.3%**	**0.6%**
tcp_conn	**28.5%**	**13.3%**	**12.5%**	**1.8%**
af_unix	**10.6%**	**-0.9%**	**-2.0%**	**-5.6%**
pipe	**4.3%**	**7.1%**	**1.7%**	**0.4%**
Geometric Mean	**20.2%**	**5.0%**	**3.9%**	**1.3%**

Table 4: Number of indirect calls relative to the number of targets they invoke.

Targets	1 targets	2 targets	3 targets	4 targets	5 targets	6 targets	> 6 targets
Indirect Calls	517	109	34	23	6	12	22

Table 5: Overhead with all defenses enabled, with indirect call promotion (icp) and several inlining budgets.

Test	LTO w/all-defenses	+icp *(99.999%)*	+icp +inlining *(99%)*	*(99.9%)*	*(99.9999%)*	*(lax. heuristics)*
null	**48.1%**	**52.7%**	**42.3%**	**42.4%**	**45.6%**	**43.6%**
read	**166.9%**	**139.6%**	**49.1%**	**16.6%**	**22.6%**	**16.8%**
write	**143.8%**	**121.6%**	**32.1%**	**16.9%**	**16.8%**	**16.3%**
open	**253.2%**	**233.0%**	**11.8%**	**9.6%**	**8.3%**	**-5.9%**
stat	**239.3%**	**220.9%**	**41.8%**	**17.8%**	**20.9%**	**-0.8%**
fstat	**93.8%**	**75.0%**	**56.7%**	**24.0%**	**23.1%**	**23.8%**
af_unix	**146.1%**	**131.8%**	**23.9%**	**18.5%**	**13.3%**	**14.1%**
fork/exit	**93.8%**	**97.2%**	**21.7%**	**6.8%**	**4.9%**	**4.5%**
fork/exec	**93.5%**	**91.6%**	**24.4%**	**8.8%**	**8.0%**	**6.8%**
fork/shell	**75.3%**	**74.3%**	**19.2%**	**8.2%**	**3.3%**	**6.8%**
pipe	**126.7%**	**106.3%**	**8.1%**	**7.5%**	**6.3%**	**4.6%**
select_file	**307.6%**	**313.9%**	**-8.6%**	**-8.9%**	**-3.5%**	**-5.3%**
select_tcp	**567.0%**	**359.9%**	**-6.9%**	**-12.1%**	**-7.0%**	**-6.1%**
tcp_conn	**270.2%**	**232.6%**	**139.6%**	**116.5%**	**30.6%**	**43.6%**
udp	**184.5%**	**156.3%**	**15.3%**	**14.2%**	**13.4%**	**15.4%**
tcp	**200.8%**	**165.5%**	**16.3%**	**15.4%**	**15.7%**	**14.3%**
mmap	**94.7%**	**83.3%**	**26.0%**	**11.5%**	**12.7%**	**10.3%**
page_fault	**94.1%**	**92.8%**	**-1.1%**	**0.5%**	**0.6%**	**-0.4%**
sig_install	**57.3%**	**52.4%**	**27.4%**	**33.8%**	**22.3%**	**15.2%**
sig_dispatch	**100.7%**	**103.4%**	**91.1%**	**12.8%**	**8.1%**	**9.6%**
Geometric Mean	**149.1%**	**133.1%**	**28.0%**	**15.9%**	**12.7%**	**10.6%**

8.3 Comprehensive PIBE Performance

Table 5 shows the performance overhead of PIBE with all defenses enabled, offering comprehensive protection from transient control flow hijacking due to LVI, Spectre V2, and Ret2spec. Without our optimizations, the defenses together incur 149.1% overhead, which is clearly far from practical. We included several configurations. The

Table 6: LMBench geometric mean overhead per defense.

Defense	LTO	PIBE
None	0.0%	-6.6%
Retpolines	20.2%	1.3%
Return retpolines	63.4%	3.7%
LVI-CFI	61.9%	1.8%
All	149.1%	10.6%

'lax heuristics' configuration uses a optimization budget of 99.9999%, while disabling heuristics on function size for sites that fit in the 99% budget, which our experiments show to be counterproductive at this budget. This is the optimal configuration, reducing overhead for the comprehensive defense to just 10.6%. Compared to the PIBE baseline with PGO but no defenses, the overhead is 18.4%. This shows that PIBE is effective in specifically reducing overhead of the defenses.

To show which defenses cause the most overhead, Table 6 provides the geometric mean overhead for each individual defense, in each case selecting the optimal configuration. The comprehensive defense incurs more overhead than the sum of the individual defenses, which is due to the fact that the LVI defense sequence must be modified to combine it with retpolines. This is consistent with the microbenchmarks shown in Table 1. Most remaining overhead is due to return retpolines, which suggests PIBE is more effective in removing forward edges (through indirect call promotion) than in removing backward edges (through inlining). Still, in each case, we reduce overhead by more than an order of magnitude, making each defense practical.

8.4 Performance Robustness to Workload Profiles

In the JumpSwitch paper [2], the authors claim that PGO approaches may be unfeasible for real-world scenarios as optimizations are only relevant for the test workload, and may become inefficient when the workload changes. This concern is understandable as optimizations would indeed highly favor the workload for which they are tuned. However, this does not mean that it will not benefit other workloads as well, especially in our case, where optimizations eliminate expensive indirect branch checks that hinder performance even in colder code. Moreover, specifically for the kernel, most applications that access the kernel context will exercise specific kernel paths most frequently (e.g., most of the applications will read/write files). Though workloads may stress kernel facilities differently, most performance relevant pathways will be the same no matter the workload.

To assess the robustness of our LMBench workload, we compare it against a workload generated for Apache. We use ApacheBench [1] on a remote client to send 1 million requests to a server configured with the PIBE profiler. We run the experiment 11 times and collect all branch execution statistics observed during these runs. We select indirect branches based on a reference optimization budget and compute the fraction both workloads have in common. Our experiment shows that, at a 99% budget, the two workloads share 58% of indirect call promotion candidate weight and 67% of inlining candidate weight.

Table 7: Throughput measurements for Nginx, Apache and DBench.

Benchmark	Configuration	*Vanilla*	*no optimization* *throughput (%)*	*PIBE optimizations* *throughput (%)*
Nginx	w/retpolines	**400645.95** *(req/sec)*	**-6.98%**	**+1.37%**
	w/ret-retpolines		**-33.32%**	**+6.05%**
	w/LVI-CFI		**-27.45%**	**+9.21%**
	w/all-defenses		**-51.71%**	**-5.95%**
Apache	w/retpolines	**192019.63** *(req/sec)*	**-3.8%**	**+0.76%**
	w/ret-retpolines		**-22.87%**	**-0.08%**
	w/LVI-CFI		**-23.41%**	**+1.88%**
	w/all-defenses		**-39.26%**	**-7.93%**
DBench	w/retpolines	**13716.2** *(MB/sec)*	**-4.25%**	**-1.78%**
	w/ret-retpolines		**-27.9%**	**-0.84%**
	w/LVI-CFI		**-20.4%**	**1.61%**
	w/all-defenses		**-45.61%**	**-6.68%**

To determine performance robustness to workloads experimentally, we use the ApacheBench workload to optimize the kernel with comprehensive defenses (retpolines, return retpolines, and LVI-CFI) and measure LMBench latencies on the kernel optimized for Apache. Even though the Apache workload is monotonic compared to LMBench, our approach still achieves a geometric mean LMBench overhead of 22.5%, a large improvement over the LTO baseline, where comprehensive defenses incur 149.1% overhead, even though it is not as good as the 10.6% achieved with the correct workload. To show that the speedup is a result of the workload and not our agressive promotion and inlining algorithms, we also measured the results with the default LLVM inliner. The default inliner's bottom-up approach guarantees that it will visit all call sites in the kernel call-graph. However, its inlining decisions are made solely based on size complexity and inline hints. With the default inliner, comprehensive defenses incur 100.2% overhead, far more than our new approach even with a mismatched workload. We conclude that our approach is robust to workload changes but achieves the best performance if our PGO uses a matching profile. This situation is common in data centers, where workloads are often predictable and applications customized to them [12, 26].

8.5 Macrobenchmarks

We next demonstrate how PIBE performs with real-world workloads that do not specifically stress the userspace-kernel transition. We run the following benchmarks: *dbench*, a disk benchmark simulating a file server workload, running on *tmpfs*; Apache Server *2.4.29* configured with the MPM event module; and the Nginx *1.14.0* web server. All macrobenchmark measurements are executed on the Skylake testbed described at the beginning of Section 8. To saturate the web servers, we run two *wrk* HTTP workload generators on a remote *Intel I7 2600* series machine. Each *wrk* instance sends 100 concurrent requests for a 4 bytes static web page for a duration of 20 minutes and reports the average requests/sec served by the web server. Our Skylake testbed is connected via two 1000 Mb/s (direct) links to the remote machine. The *wrk* instances send requests on

separate links and are configured to run on disjoint (logical) CPU subsets such that their worker threads do not race for the same CPU resources.

We evaluated the benchmarks on four kernel configurations: one for each of the three transient mitigations enabled separately and one with all transient mitigations enabled. We benchmarked each kernel configuration with and without PIBE's optimizations (using a LMBench training workload). For the retpolines-only configuration we apply only indirect call promotion. Table 7 reports the throughput degradation for each benchmark. All results are expressed as a percentage relative to the throughput obtained on the LTO baseline for the benchmark in question (baseline throughput is reported in the **Vanilla** column). To obtain the throughput for the web servers we sum up the average requests/sec served by each *wrk* instance.

We can observe that transient mitigations have a significant impact on macrobenchmarks as well, though not as pronounced as on LMBench. In most cases, PIBE's optimizations lower throughput degradation significantly. In some benchmarking scenarios, optimized fully-protected images are even faster than unoptimized retpolines-only images. On optimized kernels defended against LVI or Ret2spec, Nginx's lightweight design significantly improves performance over the vanilla baseline (e.g., 9.2% for the LVI-only configuration). Moreover, we observe that for Nginx, for some heavyweight configurations (e.g., LVI-only configuration) an optimization budget of 99% suffices to alleviate the throughput degradation. Though Apache seems less impacted by transient mitigations in the kernel, its throughput also improves more linearly with the optimization budget. We ran optimizations at a budget of 99.9999% to completely remove the throughput degradation of the LVI-only kernel configuration while Apache's throughput is still -7.93% after aggressive optimizations, while all defenses are enabled.

8.6 Security Evaluation

Our goal with PIBE is to secure the target program as much as possible by eliminating gadgets that can be used for a transient control flow hijacking attack. Indirect call promotion removes indirect calls, while inlining removes return instructions. While remaining gadgets are still protected, they require defenses and therefore incur overhead. Table 8 shows our effectiveness in removing indirect branch gadgets. The weight measurements reflect execution counts, while the sites measurements reflect code locations elided by each optimization. The last line shows absolute values for each measurement. For weight measurements it shows the cumulative branch weight candidate for inlining/promotion, while for location measurements it denotes the total number of (promotion/inlining) candidates. It should be noted that the total weight for inlining varies as indirect call promotion (running before the inlining optimization) creates more inlining candidates with increasing budgets. Moreover, promotion can inhibit beneficial inlining due to the added size complexity (in some callers) resulting from promotion operations. Furthermore, the total number of return sites that are candidates for elision varies, as inlining sometimes creates new gadgets. The results show that our approach is effective in removing indirect calls at higher budgets, while inlining results in diminishing returns. This is mostly due to our heuristics that prevent excessive inlining.

Table 8: Number of indirect branch gadgets eliminated by PIBE.

budget	indirect call						return			
	weight		call sites		call targets		weight		return sites	
99%	1243m	**98.8%**	124	**17.2%**	162	**12.3%**	12906m	**93.9%**	1447	**13.6%**
99.9%	1256m	**99.9%**	237	**32.9%**	326	**24.7%**	13019m	**93.8%**	3198	**29.7%**
99.9999%	1258m	**100.0%**	647	**89.7%**	1130	**85.6%**	13018m	**93.7%**	9969	**86.1%**
total	1258m		721		1320		variable		variable	

Table 9: Weight not elided due to size heuristics or other reasons (e.g., sites from callers with optnone attribute or callees marked as noinline).

budget	Ovr.	Rule 2		Rule 3		other	
99%	13745m	96m	**0.7%**	461m	**3.35%**	265m	**1.93%**
99.9%	13875m	120m	**0.86%**	468m	**3.37%**	264m	**1.91%**
99.9999%	13889m	133m	**0.96%**	473m	**3.41%**	264m	**1.9%**

Additional measurements show that these heuristics are effective at improving performance at budgets higher than 99%, but diminish performance at 99% (see Section 8.3).

Table 9 shows the weight not elided while inlining, due to size heuristics, for a series of budgets. The second column of the table denotes the overall execution count eligible for inlining at each budget. The percentages are relative to overall execution counts. As the table suggests, Rule 3 is a far more effective inlining inhibitor than Rule 2, as it prevents around 4x more execution counts from being elided than Rule 2. Together our size heuristics block only a small ~4% fraction of beneficial inlining, but produce reasonably sized images (see Section 8.7). Rule 2 tends to be quite effective in preventing code bloating on code paths close to kernel entry points (e.g., at the 99% budget nearly 10% of blocked inlining candidates are invoked from the syscall handler and its callees). The execution weight blocked by Rule 3 does not change much across optimization budgets, which suggests that our greedy approach has some stability (i.e., candidates inlined at a lower budget tend to be inlined at a higher budget as well). Rule 2 however has a noticeable increase in blocked weight between the 99% and the 99.9% budgets. This might be a sign of indirect call promotion inhibiting some beneficial inlining (otherwise inlined at 99%) due to the algorithm's impact on size.

Our algorithms may seem quite aggressive, but in reality they target only a small fraction of kernel indirect branches. Table 10 depicts the percentage of initial promotion/inlining candidates (the *"Candidates"* line) that our algorithms attempt to elide at different optimization budgets relative to the overall number of kernel indirect branches (*the "Ind. Branches"* line). In the most aggressive configuration (i.e., the 99.9999% budget) our inlining algorithm attempts to elide ~7.5% of available indirect branches, while for any other configuration both our algorithms attempt optimizations for at most 3% of the available kernel indirect branches.

We analyzed the kernel binaries generated by PIBE and observed that, with the exception of a few backward edges that execute during system boot and are thus not subject of transient attacks past this stage, all kernel return instructions are protected with the appropriate defenses. However, not all forward edges are converted to

Table 10: The percentage of initial promotion/inlining candidates (*"Candidates"* line) relative to the total number of indirect call/return branches (*"Ind. Branches"* line), for different optimization budgets.

Statistic	Algorithm					
	icp			inlining		
	(99%)	*(99.9%)*	*(99.9999%)*	*(99%)*	*(99.9%)*	*(99.9999%)*
Ind. Branches	**20927**	**20927**	**20927**	**133005**	**133169**	**133973**
Candidates	**0.59%**	**1.13%**	**3.09%**	**1.14%**	**2.54%**	**7.5%**

our fenced retpoline sequence (see Listing 7), so some are still vulnerable to Spectre V2 and LVI attacks. We summarize our forward edge analysis results in Table 11. The table shows the total number of protected indirect calls (converted to our retpoline thunk), the total number of vulnerable indirect calls (not converted to thunks), and the total number of vulnerable indirect jumps for a kernel image hardened with all transient mitigations (without optimizations) as well as three images hardened with all transient mitigations and optimized with PIBE at different optimization budgets (budgets included in parentheses).

As the table suggests, most of the kernel indirect calls are converted to our fenced retpoline sequence and are therefore protected against transient attacks. The number of protected indirect calls increases as optimizations are applied more aggressively (i.e., by increasing the optimization budget), as inlining duplicates more indirect calls. Disabling jump table optimizations in the compiler leaves only 5 vulnerable indirect jumps (subject to Spectre V2 attacks). In contrast, a vanilla kernel with jump tables enabled in the compiler will expose 1432 indirect jumps vulnerable to Spectre v2. As noted by the "Vuln. ICalls" line, even without PIBE's optimizations, enforcing all mitigations in the kernel still leaves 41 indirect calls vulnerable to transient attacks. These indirect calls belong to the kernel para-virtualization layer (i.e., guest hypercalls), and are implemented using inline assembly macros. As LLVM does not yet support applying the retpoline mitigation in inline assembly code, one solution to protect these vulnerable indirect calls would be to modify the assembly macros to use retpoline thunks and extend the LLVM retpoline implementation with a specialized thunk to handle memory indirect calls (the calls in question reference the target function via a memory location rather than a register). The number of vulnerable calls increases as inlining optimizations are applied more aggressively, due to code duplication, leading to 170 vulnerable indirect calls at the 99.9999% budget. We verified that on a kernel protected using JumpSwitches, the 5 (vulnerable) indirect jumps and 41 (vulnerable) indirect calls are still present.

8.7 Kernel Size due to Algorithms

Table 12 shows how our optimizations impact the size and memory usage of the kernel. For a series of budgets and configurations we show the following statistics: effective increase in size of the kernel binary, memory occupied by the kernel code at startup, slab usage and dynamic kernel memory usage. Each statistic (except *abs. size*) shows the **(%)** increase or decrease **(-)** relative to a kernel image that applies no optimization (configured as specified in the config column). Slab usage and dynamic memory usage are computed from peak values observed while running a LMBench workload. Additionally, we show the size increase relative to the LTO baseline image (the *abs. size* column).

Table 11: The number of forward edges vulnerable/protected against transient attacks. The "Def. ICalls" line shows the number of protected indirect calls. "Vuln. ICalls" line shows the number of vulnerable indirect calls. "Vuln. IJumps" shows the number of vulnerable indirect jumps.

Statistic	+ all defenses (*no optimization*)	+ all defenses (*99% budget*)	+ all defenses (*99.9% budget*)	+ all defenses (*99.9999% budget*)
Def. ICalls	20927	21638	22588	26066
Vuln. ICalls	41	73	115	170
Vuln. IJumps	5	5	5	5

Table 12: Increase in size and memory usage due to algorithms.

config.	budget	abs. size	img size	mem size	slab size	dyn size
w/all-defenses	99%	**8.1%**	**4.8%**	**0%**	**0.2%**	**0.01%**
	99.9%	**13.8%**	**10.3%**	**12.5%**	**0.3%**	**1.02%**
	99.9999%	**36.8%**	**32.7%**	**25%**	**0.1%**	**-0.21%**
w/retpolines	99.999%	**1.6%**	**0.4%**	**0%**	-	-
w/LVI-CFI	99%	**6.2%**	**4.6%**	**0%**	-	-
	99.9999%	**34.8%**	**32.8%**	**28.6%**	-	-
w/ret-retpolines	99%	**6.4%**	**4.9%**	**0%**	-	-
	99.9999%	**35.2%**	**33.2%**	**25%**	-	-

9 RELATED WORK

We propose profile-guided solutions to eliminate indirect branches, including a new *inlining* algorithm specifically tuned to maximally reduce the number of backward edges executed. Our solutions allow the application of multiple state-of-the-art *transient defenses* on the remaining branches with acceptable overheads. We consider related work in both directions.

Inlining. Profiling-directed inlining was first introduced by Scheifler [27], who noted that the optimization is close to the KNAPSACK problem. Scheifler proposed a greedy heuristic that builds on a constant ratios assumption to infer frequencies of inlined call sites. Later approaches [3, 4, 11] proposed different heuristics but all focus on trading off expected benefit (e.g., number of execution counts eliminated) against expected cost (e.g., size and complexity). Our inliner differs in that it is specifically tailored to security, prioritizing the removal of as many backward edges as possible from hot code paths. Our heuristic is similar to Scheifler's *constant ratios* assumptions, but we use it to keep track of hot backward edges that appear in the call graph due to successive inlining operations. Like [3], we use indirect call promotion to inline indirect calls. Our implementation builds on LLVM's profile-guided inliner, but alters the inlining order to eliminate as many indirect branches as possible.

Software transient execution mitigation. Ever since the discovery of Spectre [20] and Meltdown [23], the security community has scrambled to design software mitigations for systems running on vulnerable hardware. However, these mitigations [20, 24, 32] still

incur high overhead that we greatly reduced by eliminating indirect branches. Most similar to our work is JumpSwitches [2], which also optimizes Spectre V2 mitigations on forward edges. However, our approach is much more generic, mitigating more vulnerabilities (e.g., Ret2spec [24] and LVI [32]) on both backward and forward edges. Unlike our work, JumpSwitches promotes indirect calls at runtime based on execution counts. While their solution may adapt to runtime workload changes, it does so at the cost of additional overhead due to monitoring the frequencies and live-patching the code (which, in our experiments, is prone to synchronization overhead due to RCU stalls). Moreover, we have shown that our approach is robust to workload changes, even though it does not adapt at runtime. We do not execute additional code at runtime, avoid synchronization overhead, and, compared to JumpSwitches, improve cache locality by placing the instrumentation close to the indirect call—obviating the need for additional jumps across the address space. Our solution performs better (see Section 8.2), is simpler, and does not require kernel code modifications.

10 CONCLUSION

We presented PIBE, a comprehensive defense against speculative control flow hijacking attacks in the kernel, while requiring no complex run-time modifications of the kernel code. We improve each of the individual defenses' performance over the state of the art, achieving overheads well below 5% for each of them. Our approach reduces overhead for comprehensive state-of-the-art defenses from 149% to just 10.6% on LMBench and from ~40% to around 6% on several application benchmarks, which makes software defenses a viable solution for kernels running on vulnerable hardware.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Margo Seltzer, and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753 VENI "PantaRhei", and by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides x86-64 kernel binaries for most of the kernel configurations we evaluated in the paper, along with scripts to configure LMBench, run and benchmark each kernel configuration and regenerate the syscall latencies and overheads discussed in the main tables of the paper. This allows the evaluation of our results on an Intel i7-8700K (Skylake) CPU or similar microarchitectures (e.g., Haswell).

We also provide source code for the tools used during the kernel build process (e.g., binutils, LLVM 10 framework extended with support for LVI-CFI and return retpolines), the code of our LLVM optimization passes and the kernel source code to regenerate the kernel binaries used in the workflow of our evaluation. We supply the user with scripts to regenerate our Apache and LMBench profiling workloads, rebuild the kernel binaries provided in the evaluation or customize the kernels with a user-specified selection of transient mitigations and optimization strategies.

Furthermore, we also provide portable Apache and LMBench profiling workloads to speedup the customization process without the necessity of creating your own profiling workloads.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Profile-Guided Indirect Call Promotion and Inlining.
- **Program:** Linux Kernel version 5.1.0 (included with the artifact).
- **Compilation:** Modified LLVM 10 framework (included with the artifact).
- **Transformations:** ICP and Inlining transformations implemented as LLVM passes.
- **Binary:** Kernel binary images included for X86_64. Source code and scripts to regenerate profiling workloads, kernel binaries.
- **Run-time environment:** Provided binaries are for Linux (Ubuntu 18.04) for x86-64.
- **Hardware:** If regenerating the profiling workloads, an Intel CPU with the Last Branch Record feature is needed.
- **Run-time state:** Test machine must be accessible via ssh (sshd service must start automatically during a reboot).
- **Execution:** We recommend an Intel Skylake i7-8700K for verifying x86-64 results.
- **Output:** Syscall latencies (micro-seconds) and median overheads relative to baseline.
- **Experiments:** Manual Linux shell scripts.
- **How much disk space required (approximately)?:** 6GB if regenerating binaries.
- **How much time is needed to complete experiments?:** 4h 30minutes if not regenerating binaries.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.4541351

A.3 Description

A.3.1 How to access. Our benchmarks, source code, scripts, profiling workloads and precompiled kernel binaries are available on Github: https://github.com/victorduta/pibe-reproduction. Alternatively, the artifact is available on Zenodo (DOI provided above).

A.3.2 Hardware dependencies. Our precompiled kernel binaries assume an x86_64 CPU (Intel or AMD). To aid in creating a PGO kernel configuration on any x86_64 machine the artifact comes with portable profiling workloads for Apache2 and LMBench (described in the main paper). To regenerate our Apache2 and LMBench workloads one needs an Intel CPU equipped with the Last Branch Record profiling feature (e.g., Skylake, Haswell, Nehalem). For low variance LMBench results, relative to the results presented in the paper, we suggest testing with an Intel i7-8700K (Skylake) CPU.

A.3.3 Software dependencies. Our kernel binaries, and workflow scripts assume an Ubuntu 18.04 x86_64 system using a GRUB2.0 bootloader. We suggest evaluating on similar Ubuntu flavors. When evaluating the artifact on machines running kernels newer than 5.1.0 please refer to the instructions in Grub2.md, from the root of our artifact. The file describes the steps that must be done to assure that our precompiled kernel images are first in the boot order.

The experiments must be run remotely (as the test machine will be rebooted multiple times) and require ssh connectivity to the test machine.The sshd service has to start automatically after a reboot and the IP address must be static (i.e., does not change after a reboot). Moreover, there must be a user with sudo privileges on the test machine (used when connecting to the machine). The machine used in the evaluation must be connected to the network via a cable connection (the kernel images do not enable wireless drivers).

A.4 Installation

Zenodo. You can download the archive with our artifact from Zenodo:

```
$ curl https://zenodo.org/record/4541351/
    files/victorduta/pibe-reproduction-v1
    .0.5.zip?download=1 --output pibe-
    reproduction.zip
$ unzip -a pibe-reproduction.zip
```

Github. You can pull our workflow scripts, profiling workloads, precompiled kernel binaries (used to reproduce the main results in the paper) and source code from GitHub:

```
$ git clone
    https://github.com/victorduta/pibe-reproduction.git
```

Once everything is saved on the test machine, from the root directory of our provided repository/archive, run the following shell command:

```
$ ./install_tools_and_deps.sh
```

This command will install the python packages used by our workflow scripts and will configure LMBench's OS microbenchmarks. If you plan to regenerate our precompiled kernel binaries then you must also compile the LLVM framework and binutils packages included with the artifact. This can be achieved by adding the *-full* parameter to the previous command. This will also install and configure Apache and build our LLVM optimization passes. The entire process takes around 70 minutes on a modern machine. To build all tools, you require around 9GB during installation (around 6GB after everything is installed and all auxiliary files are removed).

A.5 Experiment workflow

Once you installed the above packages on the test machine, from a remote machine, type in the command:

```
$ ./run_artifact.sh USER IP /path/to/repo
```

The user must reside on the test machine and must have sudo privileges. IP is the ip address of the test machine and must be persistent after a reboot. The path is an absolute path to the root directory of our provided artifact (on the test machine). Only *run_artifact.sh* (placed in the repository root) needs to be copied on the remote for the experiment to run. The experiment takes around 4h 30min to finish on a modern machine.

During the experiment, a series of kernel images, compiled with various transient execution defenses and optimization strategies, will be loaded on the test machine and benchmarked with LMBench. For details about the kernel configurations loaded by our experiment refer to *Experiments.md*, from the root of the artifact. Alternatively, if you also want to compile the kernels on the evaluation machine (instead of using our precompiled binaries) use the *run_compile_artifact.sh* script from the remote (with the same parameters as *run_artifact.sh*).

You will be prompted multiple times to input a password. To make the process passwordless please refer to the same *Experiments.md* file.

For low-variance results make sure that the system used in the evaluation does not execute other compute- or memory- intensive applications while the experiments are running.

A.6 Evaluation and expected result

Once the workflow script finishes, from the test machine (root directory of artifact), simply run:

```
$ ./generate_tables.sh
```

The command will output the *./paper/reproduced.pdf* file. The *pdf* file is structured in three sections, each presenting LMBench microbenchmark latency overheads for various kernel transient execution configurations, discussed in the main paper. Each section contains a paragraph, describing what measurements are presented in the section and how they map back to the relevant tables in the main paper. Additionally, each section contains two tables. One shows the results obtained on the test machine (while running the artifact) while the other shows the same results as presented in the main paper (obtained on a Skylake CPU). More details on how to map back results to the relevant tables in the main paper are also discussed in the *Results.md* file, placed in the root of the artifact.

The overhead of a kernel configuration on a specific microbenchmark is computed from median latencies and is expressed as a % value relative to the baseline latency for that specific microbenchmark (i.e., the LTO baseline kernel configuration discussed in the paper). The medians are selected out of 5 benchmarking rounds.

Expect significant overhead variance, on some microbenchmarks, for images that enable transient defenses but do not apply PIBE's optimizations, depending on the microarchitecture used in the evaluation. For example, a geometric mean overhead variance of -20% was observed on an Intel Xeon, relative to our Skylake measurements, for a kernel configuration that applies all transient mitigations but no optimizations (+20% overhead variance on an AMD 1950X for the same configuration). However, similar trends should be observed regardless of the microarchitecture used in the evaluation. For example, the overhead of kernels applying transient mitigations but no optimizations are high regardless of the microarchitecture being used. Furthermore, speedups of similar magnitude should be observed for images that also apply PIBE's optimizations. The LMBench overheads of images optimized with an Apache workload should be slightly higher than the overheads of images optimized with our LMBench workload.

A.7 Experiment customization

The artifact includes the *compile_install_kernel.py* script to regenerate the kernel configurations used in the experimental workflow or even create custom kernel configurations with a user-requested

selection of transient execution mitigations and optimization strategies (e.g., selecting workloads, selecting optimization budgets). The steps to regenerate kernel binaries or create new kernel configurations are discussed in detail in *Compilation.md* (placed in the root of the artifact).

To ease the process of creating a PGO kernel configuration on any x86_64 machine, the artifact comes with two predefined workloads for Apache (workload name is apache2) and LMBench (workload name is lmbench3) that can be used in the process of customizing your kernel configuration. However, we provide the user with the ability to regenerate his own Apache and LMBench workload provided the test machine uses an Intel CPU equipped with the Last Branch Record feature. Scripts and steps to regenerate the workloads to use in the process of customizing your own kernel configuration are discussed in detail in the *Profiling.md* file. The scripts can be modified to capture workloads obtained by other tools than Apache or LMBench.

REFERENCES

[1] 2018. ApacheBench. http://httpd.apache.org/docs/2.4/programs/ab.html

[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)*. USENIX Association, Renton, WA, 285–300.

[3] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F Sweeney. 2000. A Comparative Study of Static and Profile-Based Heuristics for Inlining. 35, 7 (2000), 52–64. https://doi.org/10.1145/351397.351416

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177. https://doi.org/10.1145/945445.945462

[6] Aaron B Brown and Margo I Seltzer. 1997. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. Association for Computing Machinery, New York, NY, USA, 214–224. https://doi.org/10.1145/258612.258690

[7] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. 2020. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI 2020-Proceedings of the 30th Great Lakes Symposium on VLSI 2020*. ACM/IEEE, Association for Computing Machinery, New York, NY, USA, 169–174. https://doi.org/10.1145/3386263.3407584

[8] 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*. Association for Computing Machinery, New York, NY, USA, 769–784. https://doi.org/10.1145/3319535.3363219

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. USENIX Association, USA, 249–266.

[10] Chandler Carruth. [n.d.]. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html. Accessed: 2020-07-26.

[11] Pohua P Chang and W-W Hwu. 1989. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/73141.74840

[12] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[13] J. Corbet. 2018. Finding Spectre vulnerabilities with smatch. https://lwn.net/Articles/752408/. Accessed: 2020-08-07.

[14] M. Giles. 2018. At Least Three Billion Computer Chips Have the Spectre Security Hole. https://www.technologyreview.com/2018/01/05/146411/at-least-3-billion-computer-chips-have-the-spectre-security-hole/. Accessed: 2020-08-13.

[15] J. Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: 2020-08-07.

[16] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1470–1486. https://doi.org/10.1145/3243734.3243797

[17] Intel. [n.d.]. Deep Dive: Load Value Injection. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-load-value-injection. Accessed: 2021-01-17.

[18] Intel. [n.d.]. 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.

[19] Intel. [n.d.]. Randpoline: A software mitigation approach for branch target injection attack. https://github.com/intelstormteam/Papers/blob/master/2019-Randpoline_A_Software_Mitigation_for_Branch_Target_Injection_Attacks_v1.42.pdf. Accessed: 2021-01-17.

[20] 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. https://doi.org/10.1109/SP.2019.00002

[21] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*. USENIX Association, USA.

[22] Michael Larabel. [n.d.]. The Spectre/Meltdown Performance Impact On Linux 4.20, Decimating Benchmarks With New STIBP Overhead. https://www.phoronix.com/scan.php?page=article&item=linux-420-stibp&num=4. Accessed: 2021-01-12.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[24] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[25] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, USENIX Association, USA, 279–294.

[26] Tolvanen Sami, Bill Wendling, and Nick Desaulniers. 2020. LTO, PGO, and AutoFDO in the kernel. In *Linux Plumbers Conference*.

[27] Robert W Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM* 20, 9 (1977), 647–654. https://doi.org/10.1145/359810.359830

[28] 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*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252

[29] Soniya Shah. [n.d.]. UPDATE: Vertica Test Results with Microcode Patches for the Meltdown and Spectre Security Flaws. https://www.vertica.com/blog/vertica-results-meltdown. Accessed: 2021-01-12.

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

[31] 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)*. USENIX Association, USA, 991–1008.

[32] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*. 1399–1417. https://doi.org/10.1109/SP40000.2020.00089

[33] 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. https://doi.org/10.1109/SP.2019.00087