Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations

Michael Flanders
mkf727@cs.washington.edu
University of Washington
USA

Reshabh K Sharma
reshabh@cs.washington.edu
University of Washington
USA

Alexandra E. Michael
aemichae@cs.washington.edu
University of Washington
USA

Dan Grossman
djg@cs.washington.edu
University of Washington
USA

David Kohlbrenner
dkohlbre@cs.washington.edu
University of Washington
USA

Abstract

With the end of Moore's Law-based scaling, novel microarchitectural optimizations are being patented, researched, and implemented at an increasing rate. Previous research has examined recently published patents and papers and demonstrated ways these upcoming optimizations present new security risks via novel side channels. As these side channels are introduced by microarchitectural optimization, they are not generically solvable in source code.

In this paper, we build program analysis and transformation tools for automatically mitigating the security risks introduced by future instruction-centric microarchitectural optimizations. We focus on two classes of optimizations that are not yet deployed: silent stores and computation simplification. Silent stores are known to leak secret data being written to memory by dropping in-flight stores that will have no effect. Computation simplification is known to leak operands to arithmetic instructions by shortcutting trivial computations at execution time. This presents problems that classical constant-time techniques cannot handle: register spills, address calculations, and the micro-ops of complex instructions are all potentially leaky. To address these problems, we design, implement, and evaluate a process and tool, `cio`, for detecting and mitigating these types of side channels in cryptographic code. `cio` is a backstop, providing verified mitigation for novel microarchitectural side-channels when more specialized and efficient hardware or software tools, such as microcode patches, are not yet available.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640400

ACM Reference Format:
Michael Flanders, Reshabh K Sharma, Alexandra E. Michael, Dan Grossman, and David Kohlbrenner. 2024. Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640400

1 Introduction

Recent research has revealed many types of information leakage from microarchitectural (uarch) optimizations, leading to a growing family of side-channel attacks on sensitive software. These attacks rely on architecturally-correct, but unanticipated, uarch behavior such as side effects of speculation [36, 54], prefetchers [52, 58], data-dependent instruction timings [9], cache behaviors [47, 64], and more.

These attacks arise from the accelerating adoption of exotic uarch optimizations driven by the slowing of Moore's Law and the end of Dennard scaling. Prior work by Sanchez Vicarte et al. [51] surveyed the architecture literature to identify likely-to-be-deployed uarch optimizations with novel security implications. Since that publication, one considered family of optimizations (data memory-dependent prefetchers) now is known to exist in multiple processors [30, 58]. Another, a limited form of value prediction that sometimes predicts 0 for the upper-half result of 64-bit divisions, has been documented in Intel CPUs [46], and a third (silent stores) is referenced by name in the RISC-V Instruction Set Manual [59].

While the vast majority of software neither attempts to be nor needs to be safe from such attacks, mitigation for side channels is generally an explicit goal for cryptographic libraries, kernels, and other security-conscious software. Community-driven rules [11], combining folklore and research, have developed over time to define 'safe' programming styles that avoid known sources of leakage. More formal analysis allows defining and evaluating if a program is actually *constant-time* (CT): the program's run time does not vary based on secret inputs. Generally, well-written cryptographic libraries provide a CT guarantee covering at least key material.

However, avoiding leakage by uarch optimizations specifically has been a significant, often confusing, challenge [2, 28, 62, 66]. With each publicly revealed optimization, we update the rules for what is 'safe', necessitating a rewrite or review of all our cryptographic software. This cycle is exacerbated by the dominant reactive pattern in security: attack research first discovers and publicizes the existence of a previously unknown uarch optimization, subsequent work builds effective attacks against multiple targets, and finally tools are proposed for defending against this specific optimization.

Our approach is to instead build tools for *proactive mitigation* that consider how software can mitigate the impact of future uarch optimizations as soon as they appear. As a proactive approach, we create general mitigation transform methods targeting a strong threat model. These general approaches can be tuned to a future optimization implementation when found in the wild. Additionally, with the growing interest in developing security-conscious compilers [13, 19, 38, 56, 57], looking forward at potential future side channels will help inform what capabilities such compilers might need.

Recent work by Sanchez Vicarte et al. [51] performed a security analysis of the computer architecture literature to identify a broad family of optimizations with security implications called *instruction-centric optimizations*, of which we focus on a subset in this work. We specifically consider optimizations that trigger only as a function of register values and memory values that are operands to a single instruction. We do not consider optimizations that examine other in-flight instructions or memory not being read/written to.

We specifically focus on two exemplar classes of optimizations: *silent stores* and *computation simplification*. Silent store optimizations opportunistically drop in-flight memory stores when the value being stored matches the value already in the memory location. Computation simplification covers a broad class of optimizations that propose shortcuts, at execution time, for arithmetic operations with trivial results, e.g., `r1 := r1 + r2` when `r2 = 0`. To our knowledge, neither silent stores nor the specific computation simplifications we examine have yet been deployed in microarchitectures. However, limited forms of computation simplification, e.g., small integer operands to division operations and power-of-two divisors for some floating-point operations, have been long deployed in most x86 processors, and the RISC-V Instruction Set Manual [59] mentions silent stores. We use these two families of optimizations as case studies as they present challenges not seen in classic constant-time techniques or Spectre-related mitigations. Since even register spills can be optimized by silent stores, detection, mitigation, and verification must happen at the assembly level. Further, since our mitigations insert and substitute assembly instructions, care must be taken that instructions added to mitigate one optimization do not leak under the other. Importantly, we show—for the first time—concrete examples of uarch side channel mitigations that exhibit a *composition safety* problem.

Silent stores and computation simplification are also generalized optimizations, with most proposals focusing on operations, not specific opcodes. Thus, they can conceptually apply to the majority of instructions in a compiled x86_64 binary, and mitigating every instance will impose drastic overheads. To reduce these overheads, we need analyses that are as precise as possible to eliminate as many instructions from needing mitigation as possible. This requires precise reasoning about values of registers and memory, which is difficult to scale to codebases such as cryptographic libraries.

To mitigate leakage, we build *instruction-centric transformations* that can be applied to instructions identified as potentially leaky, preventing any possibility of optimization occuring. These transforms substitute a leaky instruction with a sequence of non-leaky instructions that are *semantically equivalent* to the original instruction: the replacement instruction sequence preserves the software-observable behavior of the program. These mitigations have a straightforward application strategy, can be automatically and efficiently verified for correctness and safety, and lend themselves to automated verification of mitigated binaries using static analysis.

Our approach is embodied in our tool, `cio`—**c**ountering **i**nstruction-centric **o**ptimizations—which serves as a foundation for building mitigations for instruction-centric optimizations. Our intention is that research groups finding novel microarchitectural side-channels will either independently, or in collaboration with others, use their expertise in the specific uarch to help develop leakage descriptions and checking passes when releasing a new vulnerability. At this point, anyone with ideas for a transformation pass can build one using `cio` and our mitigation process. This process starts with designing transforms and verifying their safety and correctness, as well as the composition of multiple transforms. Next, mitigation developers create checkers that are synergistic with their transforms; these checkers are used both to determine which instructions to transform and to verify the safety of the final transformed binary. These transforms and checkers are then provided to `cio`, along with the source for the library or program to be hardened. `cio` finally produces a binary that *guarantees* that no secret input to an instruction can cause one of the chosen optimizations to trigger. Maintainers of cryptographic libraries can release the hardened binary for concerned users until a more long-term solution is found.

Contributions. In total, our contributions are:

- A process for mitigating instruction-centric side channels using transforms.
- `cio`, a tool that provides build system and compiler support for detecting leaks, applying transforms, and verifying the safety of mitigated code.

- Tools to automatically verify the safety and equivalence of transforms and their order of application (offline).
- Transform approaches for silent stores and computation simplification that are formally verified to be safe (does not leak) and correct (semantically equivalent).
- Checkers for silent stores and computation simplification that can detect leaky instructions in binaries and prove the safety of the final, mitigated binaries.
- To the best of our knowledge, the first concrete example of microarchitectural side-channel mitigations that exhibit a *composition safety* problem.
- An evaluation of the overhead of applying our mitigations to a cryptographic library—libsodium [24].

Organization of the paper. In the rest of this paper, we discuss the existing tools and relevant background (Section 2) and our threat model (Section 3). We then describe the overall design and approach of `cio` (Section 4) as well as two case studies in building transforms and mitigations for computation simplification and silent stores (Sections 5 and 6). We also cover additional tools we developed and the composition safety problem (Section 7). To understand the impact of our mitigations, we benchmark our entire process and all mitigations on libsodium (Section 8). Finally, we discuss our mitigations and limitations (Section 9).

Release of tools. We will make all tools and all of our evaluation code and data available publicly upon publication.

2 Background

Building constant-time code is difficult in the general case, and fantastically frustrating in the presence of uarch side-channels. This has not stopped security-conscious software from attempting to close off these attack vectors as quickly as they appear. For example, x86 uarch side-channels have resulted in 10 separate configuration options in the Linux kernel[1] for Spectre V2 mitigations alone, not to mention numerous flags for each of MDS [18, 55], SSB, L1TF [54], and even the retpoline Spectre mitigations themselves [62].

2.1 Tools for Constant-Time Code

Tools for constant-time programming have taken a number of different approaches. The vast majority of these tools consider the classic CT properties of no secret-dependent branches, known variable-time instructions, or memory accesses. Recent tools have included speculative versions of these properties, which is orthogonal to the problems `cio` considers.

Most of these tools [7, 8][2] are verification tools, which validate an implementation as meeting specific CT properties. To make building CT code itself easier, sensitive code can be written in domain-specific languages or language subsets [19, 60], which allow for compilation to a guaranteed CT binary. Alternatively, there also exist libraries for common functionality [48] needed in CT programs.

Most relevant to `cio` are tools that provide compiler transforms [15, 21, 25, 63], allowing a non-CT program to be transformed into one that is CT. Notably, Dinesh et al. [25] provide SynthCT, which takes in a defined 'safe' subset of an ISA, and can synthesize replacement instruction sequences for instructions outside of the safe subset. This process allows for instructions newly found to be variable-time with respect to operand values to be replaced automatically with ones that are not. The SynthCT approach is complementary to `cio`, as unlike `cio` SynthCT considers *all* instances of a given instruction dangerous, regardless of operand values.

2.2 Hardware Configuration for Side-channel Mitigation

Processor vendors have responded to many of the discovered microarchitectural side-channels by releasing microcode patches that allow configuration bits to enable or disable *specific* optimizations. These bits operate as either boot-time global settings, or as per-process OS-managed state bits similar to FPU configurations. As with FPU state flags, these bits are generally global state for the hardware thread, with all the difficulties that accompany that. FPU state flags themselves are notoriously difficult to manage in general with unexpected effects caused by loading libraries [26] and documented security impacts for web browsers [37].

For example, Intel responded to attacks abusing Speculative Store Bypass (SSB) optimizations by adding the SSB Disable (SSBD [3]) option, allowing privilged software to disable SSB by setting a configuration bit in an MSR. Conceptually, this approach is great for sensitive software: it can request that the kernel disable SSB if it presents a threat, and leave other optimizations enabled. In practice, it can be difficult to use: most Linux distributions provide a syscall to set the option for the entire process tree, but it can only be set system wide via registry settings in Windows.

Ideal hardware configuration bits would be easy to set and guarantee coverage for only specific program regions, allowing optimizations to make the most of the rest of the program. `cio` can complement any version of these configuration bits by identifying where to apply configuration bits rather than transformations.

2.3 Hardware Support for Constant-Time Instructions

Specifically relevant to the optimizations considered here, ARM and Intel have each announced *data-independent timing* standards for their processors. These standards provide a list of supported instructions and a run-time-configurable state bit to guarantee that the covered instructions' execution times are independent of their operand values. Broadly, these are intended to offer a manufacturer-backed promise of constant-time instruction execution when enabled.

[1] https://github.com/torvalds/linux/blob/5d0c230f1de8c7515b6567d9afba1f196fb4e2f4/arch/x86/kernel/cpu/bugs.c#L1383
[2] See https://crocs-muni.github.io/ct-tools/ for more.

ARM's Data Independent Timing (DIT) [5] standard was (initially [1][3]) released as part of the ARMv8.4-A standard in 2017, and offered the ability for any exception level to enable DIT by setting the `PSTATE.DIT` configuration bit at run time. Relevant to the optimizations we consider, DIT is guaranteed to cover most arithmetic instructions and memory stores. Unfortunately, at this time the only processors known to the authors to support `PSTATE.DIT` are the Apple M-series CPUs. This leaves software on most ARM-based processors unable to request constant-time behavior for arithmetic instructions.

In 2022, Intel released the Data Operand Independent Timing (DOIT) [6] standard, promising input-independent execution times for most arithmetic and store operations on both scalars and vectors. Unlike DIT, DOIT can only be enabled by the kernel (affecting all subsequent execution), and cannot be enabled by user-mode software. DOIT is also not guaranteed to operate as expected under speculative execution.[4] No operating system as of December 2023 allows for unprivileged software to enable DOIT at run time.

There are also open questions about how comprehensive such modes will be in practice. For example, Apple's Data-dependent Memory Prefetcher (DMP) is known to introduce security challenges [58] where a store's value does not affect that store's execution time, but may affect timing of future stores. This effect arguably does not violate the semantics of DIT, and we have experimentally validated that the experiments released as part of Sanchez Vicarte et al. demonstrate DMP activation with `PSTATE.DIT` set. Conversely, Intel's FAQ on similar prefetching behavior [30] explicitly states that DOIT disables any such DMP behavior when enabled. From this, it is clear that these modes are not 'magic': the architects implementing them must make the same decisions about what features are covered that a software defense does. Since the end-user cannot customize these modes, sensitive software must be able to fall back to a software defense when their threat model differs from the vendor's.

We believe that these standards are a step in the right direction for enabling sensitive software to execute safely. However, with no extant AMD standard, limited ARM support, required OS support for Intel, reliance on global state management, and space for creative interpretation in coverage, side-channel-aware software cannot fully rely on DIT/DOIT modes for protection. It is therefore critical to have a *software-only* approach that can verify side-channel-free execution in the presence of uarch optimizations of the types we consider here.

[3]This document was later edited to remove the DIT announcement.

[4]"Developers handling secret data that should only ever be processed in a data operand independent timing manner may need to consider speculative execution vulnerabilities. These vulnerabilities may cause the secret data to be handled in a data operand dependent manner and developers may need to apply additional mitigations." [6]

3 Threat Model

Since we specifically target uarch optimizations that do not have known in-the-wild implementations, we cannot rely on a detailed model of what aspects of program execution the attacker can observe. Thus, we assume a strong adversary capable of observing any time the optimization in question happens at run time. We do not consider transient instruction execution in our analyses. This means we may miss cases where secret inputs only ever reach instructions transiently. When these optimizations exist in real processors, the observer model may be relaxed based on implementation details to reduce the application of mitigations.

We assume that the target program is constant-time if executed on a processor *without* the considered optimization(s). The program must not leak information from known channels such as secret-dependent memory accesses, secret-dependent branches, or known variable time instructions such as `idiv`. We will not introduce any new side-channels of these types via our mitigations.

In practice, this means that we expect that our input program is a cryptographic library, already hardened against known attack vectors. That library now needs to be protected from uarch optimizations appearing on a new processor.

Silent Stores. We assume the adversary can detect if any individual store is silenced or not. We assume this is possible via measuring store queue pressure, memory bandwidth utilization, execution time, or cache state changes.

We consider all explicit stores that exist in the target program after compilation, including register spills and function argument pushes. We do not consider stores that occur outside of the program's control, e.g., the kernel saving registers on a system call or context switch.

Computation Simplification. We assume the adversary can detect any time an instruction's execution is optimized to a trivial case at run time, as a function of values in registers or memory. We assume this is possible by observing functional unit utilization, or measuring execution time. As with silent stores, we consider all relevant instructions that occur in the target program post-compilation.

4 Design of `cio`

Our tool, `cio`, provides a framework for mitigating uarch side channels created by instruction-centric optimizations. `cio` takes in the source for the library or program to be hardened, and produces a drop-in replacement binary. This process *guarantees* that chosen optimizations do not trigger as a function of secret data at run time. Thus, the final binary leaks no additional secret data on a processor with the chosen optimizations as compared to one without.

To mitigate a specific family of optimizations, `cio` requires a *transformation pass* and a *leakage checker* specific to the optimizations. We currently provide such components for two

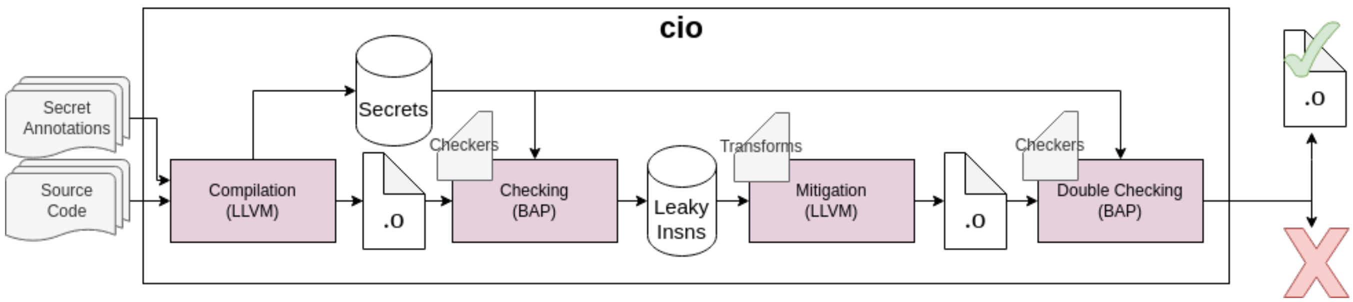

Figure 1. Internal structure and overall toolflow of `cio`.

optimizations: silent stores and computation simplification, though more can be added.

To run `cio` on a codebase, users first annotate sensitive arguments to public API functions, select the set of optimizations to prevent, and invoke `cio`. `cio` then performs four steps: compilation (Section 4.1), checking (Section 4.2), mitigation (Section 4.3), and finally double-checking (Section 4.4).

During this four-step process, `cio` coordinates three tools and their communication: a build system, LLVM [39], and BAP (the Binary Analysis Platform) [17]. Figure 1 shows the internal structure of `cio` and each of the steps it takes to produce mitigated binaries.

4.1 Step 1: Compilation

`cio` wraps a build system like `make`and starts by compiling the code base as usual using our modified LLVM. Our modified LLVM additionally saves user-provided secret annotations to files and reserves scratch registers for transforms to later use. Since the compiler only sees individual translation units, BAP will later use the annotations file to propagate secrets through the callgraph using an interprocedural taint analysis. Some of our transforms cannot use memory for scratch space (Section 6), so we must have scratch registers available. We reserve these registers during register allocation.

4.2 Step 2: Checking

After the compilation step, `cio` runs static program analyses on the compiled binary, and the results of these analyses are used by optimization-specific *checkers*. Checkers visit instructions, using the state and values produced by the analyses to check against a safety specification.

Our checkers use a *pruning* approach to detecting leaks: they assume every instruction could leak and then use their analyses to filter out or prune instructions that definitely do not leak. These pruned instructions will not be transformed by later mitigation transform passes.

This pruning approach helps in designing effective checkers that can scale up to run on cryptographic libraries. Leaky instruction-centric optimizations tend to target instructions that occur frequently in compiled code. To avoid transforming all of these instructions, effective checkers must precisely reason about operand values to prune as many non-leaky instructions as possible. Mitigation developers can trade-off between precision and speed by iteratively adding and tuning analyses—with our pruning approach, any stopping point during this process is safe.

We implement our checkers and program analyses using the Binary Analysis Platform (BAP) [17]. BAP lifts assembly instructions into BAP IR: a minimal, ISA-agnostic, intermediate language resembling CPU micro-code. BAP also provides tooling for data-flow, abstract interpretation, and symbolic analyses, which we use in our case studies.

4.3 Step 3: Mitigation

After the checkers have pruned all provably non-leaky instructions, the checkers emit a list of remaining, potentially-leaky instructions. `cio` then recompiles the cryptographic library, providing this list to optimization-specific transform passes in LLVM.

These passes apply transforms by substituting each potentially-leaky instruction with a semantically equivalent sequence of non-leaky instructions. The conditions required of transforms to preserve the program's observable behavior will be different for each ISA and optimization and can be defined to allow optimized versions of transforms. For example, x86_64 transforms do not need to set and clear unused status flags even if the original instruction sets and clears these flags.

We position these transform passes late in LLVM's compilation pipeline as a Machine IR (MIR) pass [16, 42]. LLVM's MIR is a low-level wrapper around ISA-specific assembly instructions; it is the final intermediate representation of translation units that transformation passes can be run on. Writing passes at the MIR level lets us catch low-level details like register spills that cannot be mitigated in source code. We also position our passes immediately before the assembly printing pass to avoid the possibility of later compiler optimizations breaking our passes' security guarantees.

4.4 Step 4: Double-Checking

In the final double-checking step, `cio` runs the same analyses and checkers from the checking step but this time expecting all instructions to be pruned as non-leaky. If no leaky instructions remain, then this step has verified that the final binaries are *safe* in that none of the targeted optimizations can occur as a function of secret data. If any potentially-leaky instructions remain, then the entire build process will be stopped, and no final binary will produced.

For checkers to verify the safety of the transformed binary, they must be able to verify transforms as a standalone sequence of instructions. As all potentially-leaky instructions will be transformed in the previous step, the instructions that need to be checked in the transformed binary will only be those introduced by transforms. This means that if checkers can verify transform instruction sequences in isolation under their most general input states, they will also be able to verify the safety of the final, transformed binary—all states reaching a transform site will be a subset of these most general states. We further discuss the co-design of checker analyses and transforms in Sections 5.4 and 6.2.

While we use additional tools to formally verify the safety and correctness of our transforms offline (Section 7), the implementation of our transform compiler passes and the rest of LLVM's pipeline are not verified. Thus, using checkers to verify the safety of the final binary improves confidence that the implementation of our transform passes in LLVM is correct.

4.5 Building Mitigations in `cio`

With this design in mind, we built complete mitigation systems for computation simplification and silent stores. Both mitigation systems are built using the four steps of `cio`, differing only in their checkers and transforms (see Figure 1).

5 Case Study: Computation Simplification

Computation simplification (CS) is a family of techniques that simplify the execution of instructions when operand values satisfy certain conditions [27, 51]. Such optimizations are common for floating-point operations in today's processors, with demonstrated security impacts [37].

Consider the operation `RAX := RAX * RCX`, which in x86_64 could be executed as the instruction `mul rax, rcx`.[5] If, at execution time, the value of `RCX` is 1, then the execution of this instruction could be *simplified* to a move, but not *eliminated* as the instruction has critical side effects such as status flags. Table 1 contains a complete list of the additional operations we consider, their corresponding instructions, and under what conditions they can be simplified; we handle all of the listed CS cases.

These simplifications create side channels when they occur as a function of sensitive data [51]. Using the same example, if `RCX` holds secret data and is either the value 1 or 3 at run time, then the case where `RCX = 1` causes a difference in execution behavior measurable by the adversary under our threat model. In most CS cases the difference likely manifests as reduced pressure on the ALU or reduced instruction execution time for more complex instructions.

Since we focus on general, proactive mitigation, we created three transform *approaches* for CS: pure arithmetic transforms, split-and-recombine, and conditional moves. These transforms wrap or replace unsafe instructions to prevent any optimizable operands from reaching the targeted instruction or any other instruction added by the transform.

Operation: CS case(s)	Instruction	Approach
X + Y: X or $Y = 0$ [27, 35]	`add` ($\leq$32-bit)	A
	`add` (64-bit)	SR
	`padd` (vector)	A
X − Y: $Y = 0$ [35]	`sub` ($\leq$32-bit)	A
	`sub` (64-bit)	SR
X × Y: X or $Y \in \{0, 1\}$ [27, 50]	`[i]mul` ($\leq$32-bit)	A
	`[i]mul` (64-bit)	SR
X & Y: X or $Y \in \{0, \texttt{0xFF...F}\}$	`and` (32-bit)	A
	`and` (8/16/64-bit)	SR
X ∥ Y: X or $Y \in \{0, \texttt{0xFF...F}\}$	`or` (32-bit)	A
	`or` (8/16/64-bit)	SR
X ⊕ Y: X or $Y \in \{0, \texttt{0xFF...F}\}$	`xor` (32-bit)	A
	`xor` (8/16/64-bit)	SR
X ≪ Y, X ≫ Y: $Y = 0$	`sh[lr]`	C
	`sa[lr]`	C
X ≪ Y, X ≫ Y: $X = 0$	`sh[lr]` ($\leq$32-bit)	A
	`sa[lr]` ($\leq$32-bit)	A
	`sh[lr]` (64-bit)	SR
	`sa[lr]` (64-bit)	SR
X ≫ Y: $X = \texttt{0xFF...F}$	`sar` ($\leq$32-bit)	A
	`sar` (64-bit)	SR

Table 1. CS optimization cases [10, 32, 33, 65], corresponding x86_64 instructions, and the transform method used. For approach, 'A' is an arithmetic transform, 'SR' is split-and-recombine, and 'C' is `cmov`.

[5] Using Intel syntax. All following assembly code also uses Intel syntax and references x86_64 instructions.

```
mov r11, rax
mov eax, eax
sub rax, 0x80000000
sub rax, 0x80000000
sub rcx, rax
mov ecx, ecx
mov rax, r11
```

(a) **`sub ecx, eax`**: arithmetic transform.

```
mov r10, 0x00010000
mov r10w, cx
mov cx, 0x1
mov r11, 0x00010000
mov r11w, ax
mov ax, 0x1
and rcx, rax
and r10, r11
mov cx, r10w
mov ax, r11w
```

(b) **`and rcx, rax`**: split-and-recombine transform.

```
sub eax, 0x0
setz r11d
cmovz r12d, eax
cmovz r10d, ecx
cmovz eax, r11d
sub ecx, eax
sub r11, 0x0
cmovnz ecx, r10d
cmovnz eax, r12d
```

(c) **`sub ecx, eax`**: `cmov` transform.

Figure 2. Examples of each of our CS transformation approaches. Registers `r10`–`r12` are scratch registers.

5.1 Arithmetic Transforms

Figure 2a shows our arithmetic transform approach applied to `sub ecx, eax`. Upon extending 32-, 16- or 8-bit operands to a larger register size, we can set high bits in the extended register to change the range of register values. For example, by zero-extending a 32-bit operand to 64 bits and setting the 33rd bit, we change its range of values from $0, 2^{32}-1$ to $2^{32}, 2^{33}-1$, eliminating unsafe values such as 0 or 1. We can then safely execute the original instruction over all 64 bits, and take the lowest 32 (or fewer, depending on the instruction bitwidth) bits of the result as the output of the transform. We do not need to do any additional post-processing on the output, because performing an arithmetic operation over 64 bits instead of 32 or fewer does not change the result in the lower bits.

The arithmetic approach for transforming potentially leaky instructions is preferred for most instructions, because it requires few scratch registers and is relatively short and efficient compared to our next two methods. However, this type of transformation does not work for instructions operating over 64-bit operands, as we do not have access to general purpose registers over 64 bits in size. In addition, some bitwise instructions can be transformed more efficiently with our split-and-recombine approach.

5.2 Split-and-Recombine

Our second approach, split-and-recombine, handles most of the remaining instructions, including those operating over 64-bit operands and most bitwise instructions.

Such a split-and-recombine transform for `and rcx, rax` is shown in Figure 2b. In this approach, we *split* each 64-bit operand register into two smaller pieces, each in their own registers, generally along a 48-/16-bit split. We avoid a 32-/32-bit split due to x86_64's register aliasing semantics—moving a value into the bottom 32 bits of a register clears the upper 32 bits, but moving a value into the bottom 8 or 16 bits does not clear the upper bits. Once the register values are split, we now have 'extra' bits in the full width of temporary registers. We set some of these extra bits to prevent unsafe values in both pieces of each split operand, such that the range of values for each register excludes unsafe values. We then perform the original instruction on the smaller registers and *recombine* the results.

This approach works naturally for bitwise instructions such as `and`, `or`, and `xor`. We also use the split-and-recombine approach to mitigate 64-bit arithmetic and shift instructions, where the splitting step remains as described above, but the recombination step becomes more complex. For example, recombining the operands for a shift instruction requires *adding* the two operands together because the shift amount may not be known at compile time and may not fall along neat 8- or 16-bit boundaries. This addition must in turn be made safe from CS optimization.

5.3 Conditional Move

Our final transform method, conditional move or `cmov`, is general enough to work for any instruction, but at a high cost.

This transform takes advantage of the guaranteed constant-time behavior of the `cmov` instructions [31] to generate logical branches in straight-line code. These instructions only perform the specified move if a status flag is set. For example, `cmovz eax, ecx` moves the value in `ecx` to `eax` only if the zero flag is set. Otherwise, the instruction is a no-op.

When an instruction is simplifiable, we can use the `cmov` instruction to write a substitute that has two logical branches: one for the simplified case and one for the normal case. In both cases, the target instruction is executed exactly once. In the simplifiable case, our transform substitutes static values into the target instruction, and then sets the output registers to the correct values after execution. In the non-simplifiable case, the `cmov` instructions are no-ops and do not change any inputs or outputs to the target instruction.

However, `cmov` transforms are costly. For each unsafe *value* of each operand, the `cmov` approach requires many instructions and at least one dedicated scratch space. In the case of instructions with multiple potentially unsafe values, this approach rapidly becomes infeasible. We therefore reserve `cmov` for the few instructions that the arithmetic and split-and-recombine approaches cannot handle.

Figure 2c shows how our `cmov` transform could mitigate the instruction `sub ecx, eax`. In practice, we only use `cmov` transforms to handle shift instructions with a dynamic shift size, and only for unsafe values of the shift size itself. We continue to use an arithmetic or split-and-recombine approach to handle unsafe values in the shifted register.

5.4 Checker

The CS checker visits all arithmetic, bitwise, and logical instructions and checks them against the CS safety specification, codified according to Table 1.

To verify transforms in isolation (Section 4.4) while still scaling to verify transformed cryptographic code, we use abstract interpretation [22]. Our CS checker first runs an interprocedural taint analysis, propagating developer-annotated secret arguments to all functions in the callgraph of the public API function. This taint analysis also prunes instructions that would not leak sensitive data if CS optimized.

Next, the checker runs an interval analysis [22] extended with machine-integer semantics, the trace partitioning domain [43], and the abstract memory domain from Miné [44]. The interval analysis with machine-integer semantics is already enough to verify most of our transforms; it is expressive enough to determine if an operand's value could be equal to 0, 1, or the all-one bitvector, and it also tracks the possible value ranges of variables. Tracking these value ranges lets the checker verify our arithmetic and split-and-recombine transforms, as they operate by setting high bits in operands, effectively shifting the range of values of an operand so as to exclude unsafe values.

To verify our `cmov` transforms, we extend the interval domain to be path sensitive using the trace partitioning domain [43]. The trace partitioning domain allows us to compute and maintain execution traces over the control flow graph on which abstract interpretation will then run. Though each transform has only one possible execution path, the `cmov` instruction effectively partitions the execution traces into two paths: one where an operand has an unsafe value and another where it does not. Using a pre-analysis, we partition abstract interpreter flow into two traces at each `cmov` instruction and later merge these paths when keeping them distinct no longer provides additional precision.

Since we focus on x86_64 in this work, many instructions we use or transform use register-indirect addressing, in which an instruction involves a computation plus a memory load or store (e.g., `add [eax], ecx` or `add eax, [ecx]`). To handle these instructions—as well as boost precision during initial checking—we extend the interval abstract domain to the abstract memory domain [44]. The abstract memory domain effectively runs a value-set analysis [12], uniformly representing pointers, register values, and memory contents as intervals with extra type information.

6 Case Study: Silent Stores

A *silent store* optimization occurs when an instruction attempts to store a value to a memory location that already holds that value [34, 40, 41]. Such a store can be skipped and never issued to the memory system, reducing memory traffic.

When stores are dropped based on a secret value, the secret value can be leaked through differences in execution time or uarch state observable by attackers. For example, zeroing a cryptographic key may vary in the number of stores issued depending on the number of zero bytes in the key.

The security implications of silent stores have been well studied in Sanchez Vicarte et al. [51] where the authors provide an implementation of silent stores using the gem5 simulator [14] and an attack against a classically constant-time implementation of Bitslice AES128 [49]. The attack demonstrates that under worst-case conditions, even a *single* attacker-prepared silent store is enough to leak a secret key.

To make matters worse, stores can be silenced in any form, including register spills, stack arguments, or even on saving register state for syscalls. Thus, it is impossible to avoid secret context being stored to memory during cryptographic computation without compiler support.

Mitigating silent stores is tricky, and our approach assumes the target uarch does not implement store buffer merging for nearby writes to the same memory location. Concretely, we assume that no such store merging occurs *before* the silent store implementation checks for the possibility of dropping the store. Some ARM processors have such merging[6] (but not silent stores) and might require a different transform.

6.1 Transforms

For a store of value `Y`, if the contents of the target memory location is the value `X`, then, under our threat model, we say a store of `Y` to that memory location can be silenced if `X = Y`.

The insight behind our silent store transforms is that, given bitvectors `X` and `Y`, we can always produce a third bitvector `Z` such that `X != Z` and `Y != Z`. We compute `Z` by concatenating the high bits of `X` and the low bits of `Y`, and then inverting the entire `Z` value. We then insert a *blinding* store of `Z` before the original store of `Y`. Since `X != Z`, the blinding store cannot be silent, and since `Y != Z`, the original sensitive store instruction cannot be silent.

Figure 3a shows our transform for a 64-bit wide store on x86_64: `mov [addr], rax`. Line 1 loads the `X` value, lines 2

[6] https://developer.arm.com/documentation/101924/0002/Memory-system/Store-buffer/Store-buffer-merging

```
1 mov r11, [addr]
2 mov r11b, al
3 not r11
4 mov [addr], r11
5 mov [addr], rax
```

(a) `mov [addr], rax`

```
1 mov r11b, [addr]
2 and r11b, 0xF0
3 mov r10b, al
4 and r10b, 0x0F
5 or r10b, r11b
6 not r10b
7 mov [addr], r10b
8 mov [addr], al
```

(b) `mov [addr], al`

Figure 3. SS transforms. `r10`/`r11` are scratch registers, `addr` is the store address, and `*addr` or `rax` may hold sensitive data.

and 3 compute the `Z` value, line 4 is our blinding store of `Z`, and line 5 is the original store that is now safe from silencing.

This transform uses x86_64's register aliasing semantics to compute the `Z` value: writing the bottom 8 or 16 bits of a register does not zero out the upper bits of the register. We use this to concatenate bits from `X` and `Y` by writing the low bits of one register directly into another (line 2).

This approach works for 16-, 32- and 64-bit-wide stores on x86_64 but not for 8-bit-wide x86_64 stores. In such cases, transforms can compute `Z` by selecting and concatenating bits using bitwise `and` and `or` (Figure 3b).

Note that our approach requires the use of at least one scratch register. Since all stores inserted by a transform must be silent store safe, we cannot use any scratch memory.

6.2 Checker

Our SS checker will visit all store instructions and check a safety specification: a store is safe from silencing if the store value and the value in memory could never be equal. To verify the safety of our silent store transforms in isolation, the checker needs analyses that can determine that our transform produces a blinding bitvector `Z` such that `Z != X` and `Z != Y`.

Our SS checker starts by using the same analyses as our CS checker, first an interprocedural taint analysis and then the abstract memory domain. Since the interval and abstract memory domain can express equality and inequality constraints, they can help prune some stores that definitely do not leak.

After these analyses have pruned all instructions that they can, we use a costlier relational analysis specific to our transform approach: for each remaining instruction, we compile this instruction, the N previous instructions in a backward slice [29], and the checker's safety specification to an SMT formula for checking in Z3 [23]. We found that $N = 40$ BAP IR instructions provides Z3 with enough context to verify the safety of our SS transforms in isolation. We can also tune the speed and precision of the analysis by changing N.

Although $N = 40$ might seem large, *double-checking* transformed binaries remains tractable. In the transformed binary, each store instruction not pruned by taint or abstract memory is a store introduced by a transform, and the previous $N = 40$ instructions in the slice will also belong to the same transform. This implies that if the checkers can tractably verify the transforms in isolation, then they can tractably verify the safety of the mitigated code.

However, during checking of the untransformed binary, it is likely that any 40 instructions contain many non-linear operations over wide bitvectors that are difficult for Z3 to reason about. To prevent these sequences from dominating the time to prune non-leaky instructions, we set a Z3 timeout bound slightly higher than required for double-checking.

7 Verifying Mitigations and Their Composition

We also formally verify properties of transforms while designing them. While our checkers could also be used to verify the safety of transforms, they are typically too coarse to also verify semantic equivalence, so we use separate, additional *offline verification*. This additional verification lets us verify the safety *and* semantic equivalence of transform instruction sequences without having to also co-design and implement checkers to receive feedback on transform designs.

Our offline verifiers use Serval [45], a framework for developing automated systems software verifiers on top of the Rosette solver-aided programming language [53]. Starting from equivalent symbolic CPU states, these verifiers interpret the original instruction and the transform, compiling both to SMT formulas. A correct transform must preserve the software-observable behavior of the program–this may not be exact equivalence of the final symbolic CPU states (Section 4.3). A safe transform should avoid leakage under the specified optimizations; this requires an additional safety specification that asserts the absence of leakage conditions as the interpreter visits each instruction.

Offline verification can also be used to solve the composition safety problem: is there an order the compiler can apply our transform passes that keeps our transforms safe and semantically equivalent under a given set of optimizations?

To determine a safe pass ordering—or if one exists—one can compute a dependency graph between transforms by running the verifiers over all transforms for the selected optimizations, along with each safety specification. Suppose this process finds an instruction in a transform for optimization `A` that leaks under optimization `B`.[7] If there is a transform for `B` such that the resulting composed transform is safe under both `A` and `B`, then we add a dependency between transform `A` and transform `B`. In this case, optimization `B`'s transform pass must run after optimization `A`'s transform pass. If there is no applicable such transform `B`, then there is no safe pass

[7] Note that `A` and `B` can be the same optimization, i.e., `A = B`.

ordering—the transform for `A` will leak under optimization `B`. Repeating this process until a fixpoint, the resulting dependency graph, if free of cycles, provides a safe order for transform passes.

Using this approach, we find that our SS transform passes must be run before our CS transform passes for the final binary to be safe under both optimizations. Our 8-bit SS transform in Figure 3b uses `and` and `or` instructions on unconstrained register operands, and `and` and `or` instructions can leak under CS (Table 1). In this case, we have a safe pass ordering, but had our CS transforms used 8-bit stores to scratch memory rather than using scratch registers, then there would have been a cycle in the dependency graph and no safe pass ordering.

8 Evaluation

`cio` affects both build time and execution time of the programs it hardens, varying both on the target program and the specific mitigations enabled. In this section, we explore three questions about the impact of our process for mitigating instruction-centric side channels:

- How much overhead does `cio` add to the usual build process? (Section 8.2)
- How effective are our checkers at pruning instructions that definitely do not leak? (Section 8.3)
- What run-time penalties are imposed from mitigating highly prevalent instruction-centric side channels? (Section 8.4)

We evaluate `cio` and our transforms on libsodium [24]. Libsodium is a cryptographic library that provides cryptographic primitives and high-level APIs for encryption, decryption, signatures, password hashing, and other algorithms. We configure libsodium with the `--disable-asm` flag (used by WebAssembly targets) to force reference implementations for most operations. These reference implementations do not use hand optimized assembly (see Limitations Section 9.6). Other than this flag, we build libsodium using its usual set of compilation flags. We ensure the libsodium test suite passes all tests on every version of libsodium we build.

All evaluation (compilation and run time) is performed on our system with an Intel(R) Xeon(R) Gold 6312U CPU (2.4GHz, microcode `v0xd000375`), 512GB of DDR4 (3200 MHz), and all storage on local SSDs. As we do not believe any Intel CPUs implement our target CS or SS optimizations, results in this section may underestimate run-time overheads on systems that do implement the target optimizations. On such systems, we would expect to see a lower baseline execution time due to those optimizations occuring at runtime. This would have the effect of increasing `cio`'s overheads as a ratio to the baseline.

8.1 Implementation

Our LLVM transform passes are written in 16,503 lines of C++ code. The checkers consist of 10,293 lines of OCaml, while our offline verifier consists of 5,343 lines of Racket. `cio` itself is written in 474 lines of Bash.

8.2 Build Process Overhead

The wall-clock time for `cio` to mitigate libsodium against all CS cases,[8] SS, and SS and all CS cases is shown in Table 2. We report the mean build times over 3 independent runs, with 2% or less observed variation in build times. Builds are executed with a single make job slot (`-j 1`) for both baseline and `cio` builds.

Table 2 shows that SS checking takes longer than CS checking, but CS double-checking takes much longer–nearly 25x more time–than SS double-checking. SS checking takes longer than CS checking because the SS checker's analyses extend the CS checker's analyses, and most Z3 solver calls made by the SS checker will time-out over the large, nonlinear bitwidth constraints in libsodium. Most of the difference in double-checking time is explained by the much higher number of instructions added by CS transforms; CS transforms are longer than SS transforms, and there are many more instructions needing CS transforms than SS transforms. Despite transforms adding more instructions, SS's double-checking time stays small as most Z3 solver calls will not time out when analyzing SS transforms.

	Compile	Check	Mitigate	Double-check
SS	388s	390s	381s	477s
CS	397s	265s	393s	11,836s
SS+CS	387s	391s	455s	13,293s

Table 2. `cio` build time breakdowns for libsodium reference implementations with different mitigations applied. CS stands for all CS optimizable cases of all instructions for all operations in Table 1. Each value is the mean of 3 single-job builds of libsodium in seconds. 'Compile' is the initial required build and also serves as a baseline for a libsodium build.

8.3 Checker Pruning Effectiveness

Table 3 shows the number of non-leaky instructions pruned by each analysis of the SS and CS checkers. Note that the final number of instructions transformed does not follow from the table alone—i.e., $Transformed \neq Total - Taint$ $Memory\ Domain\ Sym.\ Comp.$. This is because we prune additonal alerts for instructions that are spuriously flagged or unsupported (Section 9.3) *after* running the checkers. These spuriously flagged instructions are from BAP's lifting—e.g.,

[8] All CS optimizable cases of all instructions used by libsodium's reference implementations covering any operation in Table 1.

lifting sub instructions to add instructions causes our checkers to wrongfully flag these instructions as both operands of adds must be checked compared to only the left argument of subs. If we were to first filter out these instructions before running the checkers, the SS and CS checker still prune 323 and 6,159 instructions respectively.

	Total considered	Pruned by			Transformed
		Taint	Memory Domain	Sym. Comp.	
SS	2,695	1	428	23	1,879
CS	13,198	4,940	2,144	N/A	4,858

Table 3. Number of non-leaky instructions pruned by each analysis of the SS and CS checkers. 'Memory Domain' is the combined interval, memory, and trace partitioning domain (Section 5.4). Numbers for CS indicate pruning for all optimization cases for all operations in Table 1. Since these passes are ordered, later passes have fewer available sites to prune.

8.4 Overhead of Transformed Code

To understand the overhead we impose, we measured the end-to-end execution time and binary size impact for select subsets of our CS and SS transforms on core cryptographic operations in libsodium (Figure 4). Our baseline measurement is the run time of libsodium compiled with unmodified clang with no transforms applied. We also show the measured overhead of libsodium compiled with scratch registers reserved and no transforms applied.

Each primitive is run in an evaluation loop of 1000 iterations (100 for argon2id), with 25 warmup iterations and dropping outliers outside 1.5×interquartile range in post-processing. Run time is measured using x86_64's rdtsc and rdtscp instructions. Reported overheads are then normalized versus the mean run time of baseline libsodium.

We evaluate binary size impact by measuring the combined size in bytes of the text sections of all compiled libsodium object files. By this method, we measure binary sizes of 588 kB with SS transforms applied, 1,118 kB with CS transforms, and 1,327 kB with both SS and CS, as compared to 363 kB in non-mitigated libsodium.

9 Discussion

Overall, our transforms impose a high cost in both execution time and double-checking the mitigated binary. This result is unsurprising, given the massive instruction counts added by our mitigations. We also note that in a development setting, any build time costs would only be paid once, when cio is run on the release build of a system.

9.1 Pruning Effectiveness

Compared to the SS checker, the CS checker prunes a higher percentage of optimization-specific instructions. Many of the taint-pruned instructions are operations on constants; stack operations like push, pop, and lea; or instructions we do not transform such as call or ret. The interval-pruned instructions are those, already present in libsodium, whose operands are incidentally safe from CS optimizations. For example, in ed25519, values are commonly bitmasked, setting or clearing some bits, thereby removing some CS-unsafe values before further operations occur on those values.

On the other hand, the SS checker is fast to check and double-check libsodium, but does not prune many instructions. The 7 taint-pruned instructions are all stores of boolean constants or function pointers to global variables. The 21 symbolic-pruned instructions are additions or subtractions of a constant to a memory location, e.g., incrementing a stored counter. It makes sense that a non-relational interval analysis would not catch such instructions; incrementing a non-static interval value yields a value that can always be equal to its previous value, and adding a constant to the $\top$ interval is again equal to the $\top$ interval. We do not expect that many of the stores in libsodium are incidentally guaranteed to never store a value equal to the value already stored in memory, so it is not clear whether SS pruning numbers could be improved.

9.2 Run-Time Overheads

Figure 4 shows that despite pruning as many non-leaky instructions as possible, the overhead of transformed code can be significant and varies widely depending on the optimizations a uarch implements, the structure of the targeted code, and changes made by the compiler.

Consider preventing x86's LEA instruction from being optimizable on its scale multiplication or either of its additions. Most LEA instructions in chacha20-poly1305 are inserted for address calculation, but the structures of argon2id and ed25519 are amenable to compiler optimizations where several instructions are changed into a single LEA value computation. LEA transforms impose even more overhead if they need to preserve conditional flags, and LLVM tends to schedule such LEA instructions between sets and checks of conditional flags.

On the other hand, vector ADD and MUL transform overhead is less on argon2id compared to aesni256gcm. This is because the reference implementation of argon2id contains no vector instructions at the source-code level, but aesni256gcm is mostly vector operations. argon2id still has some overhead from vector transforms as some potentially-leaky vector instructions are added by compiler optimizations.

Our transform for 64-bit, 0- and 1-skip multiplication imposes a drastic 15.71x and 8.79x overhead on argon2id and ed25519. 64-bit multiplication is highly prevalent in these primitives, and the associated transform is composed of 116

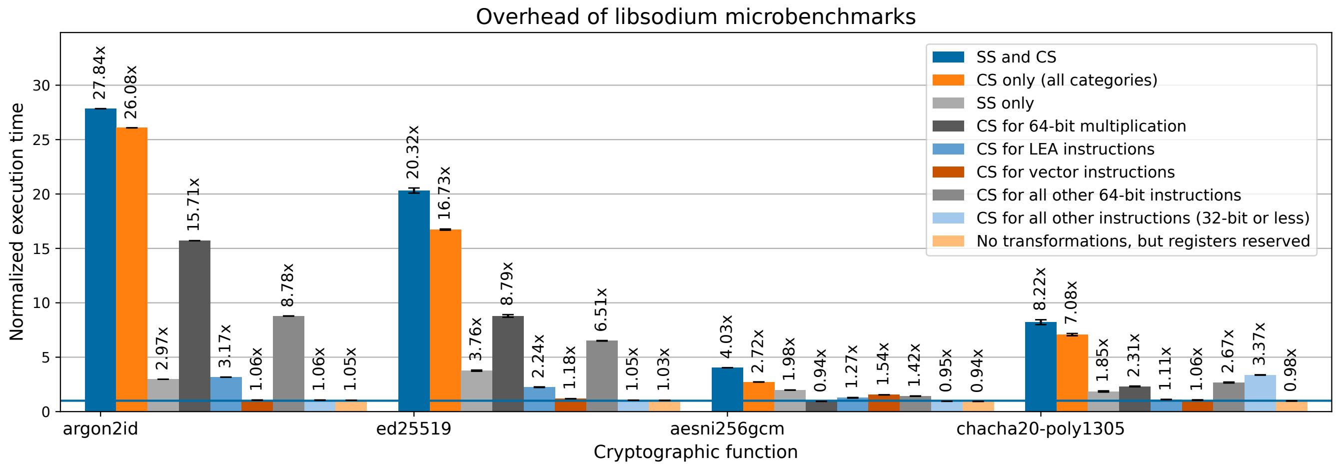

Figure 4. Normalized execution overhead of transformed libsodium code compared to the runtime of the non-transformed code built normally. Bar color represents different combinations of mitigations: 'CS for 64-bit multiplication' only includes CS transforms for 64-bit, 0- and 1-skip optimization on either operand. The overhead of scratch register reservation, without any transformations applied, is also shown for reference. Reported values are means over 1000 runs (100 for `argon2id`) with 25 warmup runs, and values outside 1.5×interquartile range dropped. Only encryption overheads are shown for `aesni256gcm` and `chacha20-poly1305`, as decrypt overheads are very similar.

instructions. Many of those instructions are guards against optimizations for all other used instructions: for example, the transform only uses left shifts in a non-0-skippable way. This imposes some additional overhead in Figure 4, as transforms for, e.g., 'CS for 64-bit multiplication' also pay overheads for many other optimizations.

We anticipate that future systems may experience different overheads when using `cio` and our mitigations. Such systems will likely not optimize the exact set of opcodes for which we have built transforms (Section 9.3), and details of specific uarch features could allow defending against less strict threat models, more pruning, and better-tailored transforms. Conversely, future systems may see even higher overheads, either as a result of optimizing even more opcodes that must then be transformed, or due to losing the run-time benefit of the optimizations themselves.

9.3 Instruction Support

We apply our CS and SS transform approaches to instructions flagged by our checkers on libsodium's reference implementation build for x86_64. Of these flagged instructions, we make a best effort to mitigate those that match proposed optimizations in the literature. This process is complicated because prior work generally discusses CS in terms of optimizing *operations*, e.g., addition or multiplication, rather than the many possible matching *instructions* or *opcodes*.

In general, we chose not to support instructions for one of three reasons:

1. We found no proposals for CS or SS optimizations that would apply to that instruction or the operation it implements. Examples: most vector operations except addition, multiplication, or stores; double-length shifts (`SHLD`, `SHRD`); rotation.
2. There are optimizations that apply to similar instructions or operations, but we judged that those optimizations were unlikely to benefit that specific instruction or class of instructions. Example: we transform `ADD` but not `ADC` (add with carry) due to the carry flag likely not being available early enough in the pipeline for CS optimization to be applied.
3. The instruction already is not constant-time, so there is no need to preserve such a property under CS or SS. Examples: `DIV` and `IDIV` [31].

9.4 Generalizing Beyond SS and CS

We believe that `cio` can be used to mitigate the remaining instruction-centric optimizations described in Sanchez Vicarte et al [51].

To mitigate leakage due to computation reuse [51], we could construct a checking pass that identifies potentially memoizable computations along paths. The transform pass might change a flagged instruction into a sequence of instructions that compute the same value but without matching a previous entry in a memoization table. Offline verification (Section 7) of this transform differs from CS and SS's offline verification as the set of required transforms is dynamic and dependent on the potentially memoizable computations in the target program.

As another example, an instance of value prediction [51] was recently discovered in Intel CPUs [46], in which 64-bit division with a 128-bit dividend speculates that the upper 64 bits of the dividend are zero. We believe we could prevent this optimization by using the split-and-recombine approach used for computation simplification (Section 5.2). The transform would split the division into multiple parts such that each part always has zero in the upper bits of the dividend, then recombine for the final quotient. Further, it appears that Intel's DOIT [6] bit may not cover this optimization.

cio can generally handle mitigations that require targeted substitution or insertion of instructions, so we believe that cio can also be used to mitigate leakage beyond instruction-centric optimizations. To mitigate data-at-rest leakage, for example, if we assume that secrets are never compiled into a binary, an instruction-centric approach could transform each store of potentially secret data into a store of data in a different format that is not leakable under the given microarchitecture. Offline verification can be used to verify that the alternate formatting of the data is always correctly computed, and double checking would ensure that all secret data has been stored in the alternate format. Compared to 'purely' instruction-centric mitigation, this approach changes the ABI of the program and would require additional mitigation or declassification at, e.g., library call boundaries.

9.5 Value-decay of software-only mitigation

Ideally, vendors will continue to address side channels early in the design process and provide features (DIT [5], DOIT [6], Zkt [4]) that allow software to better express the safety properties it needs to hardware. In that world, cio still works well, since the only thing that would change is that our transforms can switch to toggling the feature/mode bit. We would still like to have the performance improvements of uarch optimizations available as often as possible without leaking information, so cio's process of check, prune, transform, and double-check will still be valuable.

It is also worth considering the long tail of hardware and legacy systems that have interesting optimizations that are not yet public, and do not have configuration options. As the community discovers side channels on these systems, software-only mitigations will continue to be valuable.

9.6 Limitations

Libsodium provides more performant, vectorized implementations for some cryptographic functions, such as the Sandy2x implementation of ed25519 [20]. These implementations can include inline assembly, which are separate, difficult-to-handle constructs in LLVM MIR, or assembly files, which are not accessible in compiler passes. As a result, we mitigate and evaluate on libsodium's reference implementations, which include neither inline assembly nor assembly files. Supporting the more performant implementations would likely require switching to a binary rewriter [61]. Unfortunately, reserving our required registers would complicate this, and would require a full re-translation of the binary.

While we outline a process for safely composing transforms in Section 7, we manually combine SS transforms with their dependent CS transforms in our LLVM MIR pass. For example, the transform for the x86_64 opcode ADD64mr is written to be both SS and CS safe, and thus we do not have to compute pass orderings on the fly. Our checkers also do not consider transient inputs to instructions, and may prune instructions that can leak speculatively.

cio currently raises alerts while double-checking the safety of the transformed binary, alerting on 43 out of 184,659 instructions in our transformed libsodium. These remaining alerts are caused by known bugs in our analyzers and checkers that will be fixed soon.

10 Conclusion

While the outlook for novel microarchitectural leakage channels is somewhat dire, we have shown that compiler-based software-only mitigation is possible. We have found that the approach in cio can be applied to a variety of instruction-centric optimizations, and the mitigated code can be verified to be free of optimization-induced side channels. The costs for these mitigations are predictably high, in compilation time and execution time, but feasible for critical software.

Overall, this is fundamentally *good* news. If security researchers discover a significant leakage channel in current processors for an operation such as vector moves or basic arithmetic, cio can verifiably mitigate a mainstream cryptographic library immediately. The core challenge going forward is how to adapt to optimization classes not yet considered by cio. One promising direction is combining program synthesis techniques with our generalized transform approaches. As such, we encourage future security analyses for uarch optimizations to consider what generic transform approaches would mitigate that optimization.

In an ideal setting, software computing on secrets could configure and monitor hardware-backed constant-time configuration bits. Unfortunately we cannot always rely on the existing options (DIT and DOIT) as they do not cover all processors (or even vendors), and it is unclear what classes of optimizations will be covered in practice. Software-only mitigation serves as a necessary, albeit expensive, backstop that software like cryptographic libraries can rely on.

11 Acknowledgments

We thank the anonymous reviewers and our shepherd, Ashish Venkat, for their helpful feedback. We also thank Ivan Gotovchits and Benjamin Mourad from BAP for their guidance in building our analyzers and checkers, Zachary Tatlock and Max Willsey for their thoughts on transform ordering and synthesis, and Antoine Miné for answering questions about the abstract memory domain used by the checkers. This work was partially funded by NSF grants 2154183 and 2140004.

A Artifact Appendix

A.1 Abstract

This artifact contains our tooling for `cio` and scripts that reproduce our evaluation results. Since tooling for `cio` spans several different programming languages, libraries, and specific versions of tools, we recommend using our Docker image to set up an environment for running `cio` and our evaluation scripts. We provide additional instructions to set up our artifacts to reproduce the figures and tables in our evaluation section.

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

- **Algorithm:** Binary analysis, build system, static analysis
- **Program:** `cio`—drives LLVM, Binary Analysis Platform (BAP), libsodium; `eval.sh`—drives evaluation of `cio`
- **Compilation:** LLVM—included in Docker image
- **Binary:** Included in Docker image
- **Run-time environment:** Requires Docker; stack size increased via `ulimit -s 32768`
- **Hardware:** Single core x86_64
- **Metrics:** Run time; binary size; number of instructions pruned and transformed
- **Output:** Plot; text files containing data for tables. Compare against paper to see expected results
- **Experiments:** Automated via `eval.sh`; see `README` for details: https://github.com/counter-optimization/cio-artifact/README.md
- **How much disk space required (approximately)?:** 30GB total for the Docker image `tar` file, the image itself, and its outputs
- **How much time is needed to prepare workflow (approximately)?:** Once Docker image is downloaded, approximately 5 minutes
- **How much time is needed to complete experiments (approximately)?:** About 3 hours on a clean AWS instance with 32GB of memory; more depending on resources
- **Publicly available?:** Yes: https://github.com/counter-optimization/
- **Code licenses?:** BSD 3-Clause License
- **Workflow framework used?:** Bash script, autotools, Makefile, Python
- **Archived?:** Yes: https://doi.org/10.5281/zenodo.10594315

A.3 Description

All artifacts are available through the Github organization https://github.com/counter-optimization. `cio` and its evaluation scripts are available at https://github.com/counter-optimization/cio. The repo https://github.com/counter-optimization/checker contains code for our checkers, analyzers, and offline verification tools. https://github.com/counter-optimization/llvm-project contains our fork of LLVM with code added for secret argument annotations, mitigation passes, and scratch register reservation. The `cio` repo contains `cio` and an evaluation script, `eval.sh`, that generates Figure 4's plot and data for the tables in the evaluation section.

A.3.1 How to access.

All artifacts are available at https://github.com/counter-optimization/. We provide a downloadable Docker image with all dependencies needed to run `cio` and its evaluation scripts: https://homes.cs.washington.edu/~aemichae/cio-asplos24aec.tar.gz. The raw Dockerfile and accompanying `README` can be found at https://github.com/counter-optimization/cio-artifact.

A.3.2 Hardware dependencies.

`cio` requires an x86_64 machine as mitigations are currently only implemented for x86_64. The downloadable Docker image is about 8.4GB. Approximately 30GB of disk space is required for the combination of the downloadable image file, the image itself (once loaded by Docker), and the evaluation outputs. We recommend using a machine with around 32GB of memory.

Building the Docker image from the Dockerfile (as opposed to downloading it directly) requires approximately 115 GB of storage, due primarily to the size of LLVM, and can be expected to take several hours at least, depending on available resources. Therefore we strongly recommend downloading the pre-built Docker image from https://homes.cs.washington.edu/~aemichae/cio-asplos24aec.tar.gz.

A.3.3 Software dependencies.

All software dependencies are installed by the Docker image, which itself depends only on Docker. The dependency list below is for reference only.

Our evaluation requires a Linux machine with the OCaml base compiler v4.14.0, the Binary Analysis Platform (BAP) v2.5.0, our fork of LLVM, and libsodium v1.0.18-RELEASE. Our fork of LLVM is available at https://github.com/counter-optimization/llvm-project. For installing OCaml packages, we used the OCaml package manager, `opam`, to install the ocaml-base-compiler.4.14.0, BAP v2.5.0, Z3 v4.11.2, zarith v1.12, core v0.14.1, core_kernel v0.14.2, base v0.14.3, ocamlgraph v2.0.0, memtrace v0.2.3, dolog v6.0.0, splay_tree v0.14.0. The opam package providing Z3 bindings relies on Z3 with the same version number: v4.11.2.

Our evaluation scripts additionally require python3, matplotlib, and numpy.

A.4 Installation

We provide a Docker image that installs all required dependencies. See the `README` at https://github.com/counter-optimization/cio-artifact for instructions on setting up the Docker image.

For manual installation, all software dependencies need to be installed and our fork of LLVM needs to be built following the normal LLVM build process. After building our fork of LLVM, running `LLVM_HOME=/path/to/our/llvm/bins make all` in the `cio` repository will finish the installation of `cio` and evaluation dependencies.

A.5 Basic Test

The command `make test` runs `cio` on a simple C program. If run correctly, the basic test should display a log of running cio on the test file and end with the output `"1 + 1 = 2. Done!"`. See the `README` for details.

A.6 Experiment workflow

The `eval.sh` script, found in the root of the artifact Github repo, drives the evaluation process. The script builds, tests, and collects evaluation data for libsodium:

1. using the normal build process with unmodified `clang`
2. with scratch registers reserved
3. with all CS mitigations applied
4. with five different subsets of CS mitigations applied
5. with all SS mitigations applied
6. with all CS and SS mitigations applied

After these different builds, the script generates the plot for Figure 4 and outputs text to fill in the other tables in the Evaluation section.

By default, the `eval.sh` script does not run the double-checking phase (Section 4.4). To run full experiments with the double-checking phase, remove the `-skip-double-check` flag on line 168 of the Makefile. Be aware that enabling double-checking will cause the evaluation run to take roughly twice as long to complete (>6-8 hours).

A.7 Evaluation and expected results

A successful run of `eval.sh` will produce a figure, some tables, and some textual data on the command line. The files that `eval.sh` generates will be in a timestamped directory with suffix `-eval`; this directory will be referred to as `BUILD_DIR` and symbolically linked from `latest-eval-dir`.

Results for mitigation overhead (Figure 4) will be in the `benchmarks` subdirectory of `BUILD_DIR`, with the plot at `microbench-overheads.pdf`. Pruning numbers (Table 3) and compilation time/overhead numbers (Table 2) will be printed to stdout after `eval.sh` runs and should match the numbers in the paper's table exactly.

References

[1] Introducing 2017's extensions to the Arm Architecture, 2017. URL https://web.archive.org/web/20171107164628/https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture.

[2] Chromium: Cross-origin pixel reading and history sniffing via svg filter timing attack, 2017. URL https://bugs.chromium.org/p/chromium/issues/detail?id=686253.

[3] Speculative store bypass cve-2018-3639 intel-sa-00115, 2018. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html.

[4] RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions, 2022. URL https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar.

[5] ARM Developer documentation - DIT, Data Independent Timing, 2023. URL https://developer.arm.com/documentation/ddi0601/2023-03/AArch64-Registers/DIT--Data-Independent-Timing?lang=en.

[6] Intel Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance, 2023. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *Proceedings of USENIX Security 2016*, pages 53–70. USENIX, August 2016.

[8] Amazon. Constant time verification tests for s2n. URL https://github.com/aws/s2n-tls/tree/main/tests/sidetrail.

[9] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In Lujo Bauer and Vitaly Shmatikov, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2015*. IEEE Computer Society, May 2015.

[10] E. Atoofian and A. Baniasadi. Improving energy-efficiency by bypassing trivial computations. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 7 pp.–, 2005. doi: 10.1109/IPDPS.2005.253.

[11] Jean-Philippe Aumasson. Cryptocoding. URL https://github.com/veorq/cryptocoding.

[12] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 13*, pages 5–23. Springer, 2004.

[13] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the spectre era. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901, 2021. doi: 10.1109/SP40001.2021.00046.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL https://doi.org/10.1145/2024716.2024718.

[15] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

[16] Braun, M. 2017 LLVM Developers' Meeting: M. Braun "Welcome to the back-end: The LLVM machine representation". URL https://www.youtube.com/watch?v=objxlZg01D0&ab_channel=LLVM.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 463–469. Springer, 2011.

[18] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[19] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: A dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314605. URL https://doi.org/10.1145/3314221.3314605.

[20] Tung Chou. Sandy2x: New curve25519 speed records. In *International Conference on Selected Areas in Cryptography*, pages 145–160.

Springer, 2015.
[21] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P*, 2009.
[22] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973.
[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.
[24] Frank Denis. The sodium cryptography library, Jun 2013. URL https://download.libsodium.org/doc/.
[25] Sushant Dinesh, Grant Garrett-Grossman, and Christopher W Fletcher. Synthct: Towards portable constant-time code. In *ndss*, 2022.
[26] Brendan Dolan-Gavitt. Someone's been messing with my subnormals!, 2022. URL https://moyix.blogspot.com/2022/09/someones-been-messing-with-my-subnormals.html.
[27] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, Sara Baghsorkhi, and Josep Torrellas. SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 796–810, 2020. doi: 10.1109/MICRO50266.2020.00070.
[28] Dave Hansen. Linux kernel mailing list on doit patches. URL https://lore.kernel.org/lkml/c5809098-9066-d90d-1bcc-108a11525cac@intel.com/.
[29] Mark Harman and Robert Hierons. An overview of program slicing. *software focus*, 2(3):85–92, 2001.
[30] Intel. Data Dependent Prefetcher, 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html.
[31] Intel. Data operand independent timing instructions, 2023. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html.
[32] Md. Mafijul Islam and Per Stenstrom. Reduction of energy consumption in processors by early detection and bypassing of trivial operations. In *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 28–34, 2006. doi: 10.1109/ICSAMOS.2006.300805.
[33] Md. Mafijul Islam and Per Stenstrom. Energy and performance trade-offs between instruction reuse and trivial computations for embedded applications. In *2007 International Symposium on Industrial Embedded Systems*, pages 86–93, 2007. doi: 10.1109/SIES.2007.4297321.
[34] Ilhyun Kim and M.H. Lipasti. Implementing optimizations at decode time. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 221–232, 2002. doi: 10.1109/ISCA.2002.1003580.
[35] Soontae Kim. Reducing alu and register file energy by dynamic zero detection. In *2007 IEEE International Performance, Computing, and Communications Conference*, pages 365–371, 2007. doi: 10.1109/PCCC.2007.358915.
[36] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.
[37] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, pages 69–81, 2017.
[38] Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. Robust constant-time cryptography. POPL '23, 2023.
[39] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.
[40] K.M. Lepak and M.H. Lipasti. Silent stores for free. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 22–31, 2000. doi: 10.1109/MICRO.2000.898055.
[41] K.M. Lepak and M.H. Lipasti. On the value locality of store instructions. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 182–191, 2000. doi: 10.1145/339647.339678.
[42] LLVM developers. Machine ir (mir) format reference manual. https://llvm.org/docs/MIRLangRef.html.
[43] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*, pages 5–20. Springer, 2005.
[44] Antoine Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, page 54–63, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593362X. doi: 10.1145/1134650.1134659. URL https://doi.org/10.1145/1134650.1134659.
[45] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359641. URL https://doi.org/10.1145/3341301.3359641.
[46] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing, 2023.
[47] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA'06*, 2006.
[48] Thomas Pornin. Cttk: Constant-time toolkit. URL https://github.com/pornin/CTTK.
[49] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of aes. In *Cryptology and Network Security*, 2006.
[50] S.E. Richardson. Exploiting trivial and redundant computation. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 220–227, 1993. doi: 10.1109/ARITH.1993.378089.
[51] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2021. doi: 10.1109/ISCA52012.2021.00035.
[52] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243736.

URL https://doi.org/10.1145/3243734.3243736.

[53] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509586. URL https://doi.org/10.1145/2509578.2509586.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[55] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[56] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434330. URL https://doi.org/10.1145/3434330.

[57] Ashwin Prasad Shivarpatna Venkatesh, Aditya Bhat Handadi, and Martin Mory. Security implications of compiler optimizations on cryptography - A review. *CoRR*, abs/1907.02530, 2019. URL http://arxiv.org/abs/1907.02530.

[58] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022.

[59] Andrew Waterman and Krste Asanovic'. The risc-v instruction set manual, volume i: User-level isa, document version 20191213. 2019. URL https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[60] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–29, 2019.

[61] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

[62] Johannes Wikner and Kaveh Razavi. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, 2022.

[63] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[64] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE S&P*, 2019.

[65] J.J. Yi and D.J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 462–465, 2002. doi: 10.1109/ICCD.2002.1106814.

[66] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate slh: Taking speculative load hardening to the next level. In *USENIX Security*, 2023.