Protean: A Programmable Spectre Defense

Nicholas Mosier
Department of Computer Science
Stanford University
Stanford, USA
nmosier@stanford.edu

Hamed Nemati
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
hnnemati@kth.se

John C. Mitchell
Department of Computer Science
Stanford University
Stanford, USA
jcm@stanford.edu

Caroline Trippel
Departments of Computer Science and Electrical Engineering
Stanford University
Stanford, USA
trippel@stanford.edu

***Abstract*—We present the PROTEAN Spectre defense—the first to be altogether *comprehensive*, covering all side-channels and speculation; *programmer-transparent*, requiring no source modifications; and *programmable*, tailoring its hardware protections to software's security needs. Several Spectre defenses offer the first two features, but protect a hardware-defined subset of architectural state from transiently leaking. Meanwhile, many Spectre-vulnerable programs process secrets in ways that such rigid protections cannot both performantly and fully secure. PROTEAN overcomes this limitation through: (1) ProtISA, an ISA extension that allows software to tell hardware which architectural registers and memory bytes require protection from transiently leaking at each program point; (2) ProtCC, a compiler that automatically infers and programs ProtISA protections for vulnerable code with minimal user input; and (3) ProtDelay and ProtTrack, two alternative hardware mechanisms that performantly enforce software-defined ProtISA protections. By flexibly tailoring a hardware Spectre defense to a program's data protection needs, PROTEAN significantly reduces the overhead of fully securing vulnerable programs. With ProtDelay/ProtTrack, it averages 0.27x/0.18x and 0.42x/0.34x of the runtime overhead of the best secure baseline for programs with and without mixed security needs, respectively, at lower/comparable hardware complexity.**

***Index Terms*—Hardware security, hardware side-channel attacks, Spectre defenses, hardware-software codesign**

I. INTRODUCTION

Spectre attacks exploit hardware mispredictions to coerce victim programs into transiently *transmitting* (leaking) their secret data via microarchitectural side channels, threatening the security of all programs that hold secrets in architectural state. Common *Spectre-vulnerable* (hereafter, just *vulnerable*) applications include widely-deployed cryptographic libraries [27], [34], [105], web browsers [64], [68], [88], [111], and operating system (OS) kernels [65], [67], [68], [139].

To address this threat, dozens of Spectre defenses have been proposed. They vary in the types of side channels and speculation they address, the source code modifications they require, and the classes of vulnerable programs they *target* (i.e., are tailored to protect, §III-B). Many Spectre defenses are *noncomprehensive* (§X-1). These address only specific side channels (e.g., data caches) or speculation primitives (e.g., conditional branch prediction), leaving programs vulnerable to transient leaks involving others. Some defenses are comprehensive but *non-programmer-transparent* (§X-2). These require source code modifications, which limit their applicability and ease of adoption. *Comprehensive*, *programmer-transparent* defenses can fully secure vulnerable programs with minimal manual effort [13], [32], [138], [148]. Yet today, all such Spectre defenses target a single class of vulnerable programs (Tab. I), leading to performance or security limitations for other classes, which we address in this paper.

These limitations manifest for many production programs that do not fit squarely in any one of the four common classes of vulnerable code (§III-A, Fig. 2). An example is the nginx HTTPS web server (Fig. 1, left), which features code from all four classes. Because nginx contains *unrestricted* code, which *transmits* (leaks) secrets architecturally, the only prior defense capable of fully securing it is SPT's secure baseline (SPT-SB) [32], which blocks the transient transmission of *all* architectural state. However, SPT-SB overprotects nginx's *non-secret-accessing*, *static constant-time*, and *constant-time* components, incurring a huge and unnecessary performance hit (Tab. I). Other defenses, like STT [148] and SPT [32], that would fully secure much of nginx's code more performantly are inadequate to fully secure the application as a whole.

A. This Paper

We present the PROTEAN Spectre defense (Fig. 1, right), which is comprehensive, programmer-transparent, and—to address the limitations above—*programmable*. PROTEAN (1) makes its protections programmable via the *ProtISA* ISA extension (§I-A1); (2) retains programmer-transparency by automating their specification with the *ProtCC* compiler (§I-A2); and enforces them comprehensively in hardware with one of two mechanisms, *ProtDelay* or *ProtTrack* (§I-A3).

1) Programmable ProtSets with ProtISA: We observe that all Spectre defenses implicitly define, at each program point, a set of architectural state elements whose contents they promise to prevent from transiently leaking. We call this a

Defense	*ProtSet*	*Protection mechanism*	*Vulnerable program class*				
			ARCH	*CTS*	*CT*	*UNR*	*multiple (Fig. 1)*
NDA	all mem	AccessDelay	✓	✗	✗	✗	✗
SpecShield	all mem	AccessDelay	✓	✗	✗	✗	✗
STT	all mem	AccessTrack	153%	✗	✗	✗	✗
SPT	unxmitted state	AccessTrack†	✓	11%	36%	✗	✗
SPT-SB	all state	XmitDelay	✓	✓	✓	187%	177%
PROTEAN (ours)	**programmable**	**ProtDelay**	**20%**	**6%**	**22%**	**170%**	**48%**
		ProtTrack	**7%**	**4%**	**16%**	**169%**	**32%**

TABLE I: PROTEAN targets (§III-B) all vulnerable code classes (§III-A): non-secret-accessing (ARCH) $\subset$ static constant-time (CTS) $\subset$ constant-time (CT) $\subset$ unrestricted (UNR). It secures them more performantly than class-specific defenses [13], [32], [138], [148] that target (✓/x%), secure but do not target (✓/x%), or do not secure (✗) them. Percentages (from Tab. V) are runtime overheads for securing vulnerable programs with the most performant available defense. †Extends AccessTrack (§VI-B2).

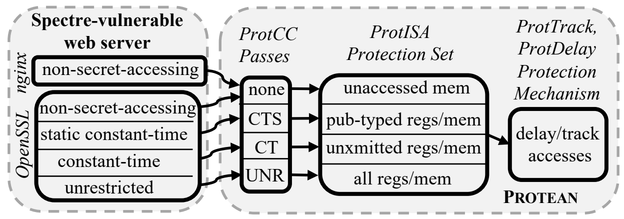


Fig. 1: PROTEAN can performantly and fully secure programs containing components from different vulnerable code classes (§III-A), e.g., the nginx HTTPS web server [100], which contains unrestricted, constant-time, static constant-time, and non-secret-accessing code. Prior defenses either do not secure this program (e.g., STT [148], SPT [32]) or nonperformantly secure it *as if* all components are unrestricted (SPT-SB [32]).

protection set (ProtSet). How a defense defines its ProtSet per program point dictates which vulnerable code classes it targets (i.e., is tailored to protect). Existing comprehensive, programmer-transparent Spectre defenses define their ProtSets via hardware-fixed rules, restricting them to target a single class (Tab. I). This limitation can be overcome by making ProtSets software-programmable, such that software—from any class or multiple classes—can specialize a defense's protections to target its specific data protection needs.

To this end, PROTEAN introduces ProtISA, which exposes a software-programmable, hardware-tracked ProtSet via a single instruction prefix, `PROT`. The presence of a `PROT` prefix for an instruction specifies that its output register and input memory operands contain data that must be *protected* from transiently leaking. The absence of `PROT` indicates conversely that protection is not needed for either. For example, a prefixed `PROT mov rax,rdx` protects output register `rax`; an unprefixed `mov rcx,[rsp]` unprotects `rcx` and the eight bytes of memory pointed to by `rsp`. Microarchitectural support for ProtISA adds protection tag bits to a few structures in an out-of-order processor, plus modest logic that uses these bits to track a program's software-defined architectural ProtSet as it is dynamically updated by instructions' `PROT` prefixes.

2) Automated ProtSet Programming with ProtCC: Prior attempts to make hardware-tracked ProtSets programmable have produced non-programmer-transparent defenses, which require manual source modifications (§X-2). In contrast, we observe that *automated* compiler analyses can conservatively (for security) yet sufficiently precisely (for performance) program a ProtSet for code with knowledge of only its class.

Thus, PROTEAN provides the ProtCC compiler, which extends LLVM [74] to support automatically programming ProtISA with a suitable ProtSet for code that belongs to or combines any of the four jointly exhaustive vulnerable code classes in Fig. 2. ProtCC inserts no `PROT` prefixes for non-secret-accessing code, as all instruction operands are guaranteed to be public; such code runs unmodified on PROTEAN hardware. For constant-time, static constant-time, and unrestricted code, ProtCC runs register-only data-flow analyses to infer a conservative ProtSet at each program point and then inserts `PROT` prefixes accordingly. ProtCC flexibly targets multi-class programs by allowing the user to specify code classes at a component (e.g., library) and/or function granularity; we do this for nginx (Fig. 1) in §VIII-B3. Also, while not our focus, ProtCC's inferred ProtSets can be refined though source-level secrecy annotations or user-defined analyses (§V-C).

3) Enforcing Programmed ProtSets in Hardware with ProtDelay/ProtTrack: A Spectre defense enforces its ProtSet (i.e., prevents the transient transmission of its contents) via a *protection mechanism.* All comprehensive Spectre defenses feature hardware protection mechanisms [13], [32], [35], [118], [138], [146], [148], as does PROTEAN. In particular, PROTEAN features two alternative hardware protection mechanisms, ProtDelay and ProtTrack, which extend state-of-the-art mechanisms AccessDelay (from NDA [138]) and AccessTrack (from STT [148]), respectively. Not having been designed with programmable ProtSets in mind, AccessDelay and AccessTrack provide only partial security and incur high performance overheads when applied to ProtISA programs directly. ProtDelay and ProtTrack address both issues; ProtTrack outperforms ProtDelay with higher hardware complexity.

4) Contributions: Our main contributions are as follows:

- **ProtISA:** A novel ISA extension that exposes a software-programmable, hardware-tracked ProtSet.
- **ProtCC:** LLVM compiler passes that automatically program ProtISA to target programs that belong to one or more of the four important code classes covering all vulnerable code.
- **ProtDelay/ProtTrack:** Performant hardware protection mechanisms for comprehensively enforcing ProtISA ProtSets with different performance-complexity tradeoffs.
- **PROTEAN (ProtISA+ProtCC+ProtDelay/ProtTrack):** A comprehensive, programmer-transparent, and programmable Spectre defense that can target any vulnerable program.
- **Security Evaluation:** We sketch a proof that PROTEAN fully secures all vulnerable programs, given an accurate classification of their components/functions. We then use an enhanced version of the AMuLeT fuzzer [41] to validate the security of PROTEAN and the three state-of-the-art Spectre defenses—STT, SPT, and SPT-SB—that serve as secure baselines in our performance evaluation, with their hardware

support all implemented in gem5. We find new gem5 side channels that leak the operands of division micro-ops and a corner-case bug in all defenses inherited from STT's gem5 implementation [40], which we fix.

- **Performance Evaluation:** We compare the performance of PROTEAN to the above secure baselines, atop a novel gem5 hybrid-core configuration resembling an Intel Alder Lake processor, when securing various single-class programs and multi-class nginx (Fig. 1). Averaged across single-class programs, PROTEAN (ProtDelay/ProtTrack) exhibits 0.42x/0.34x the runtime overhead of the best secure baseline. For nginx, it achieves 0.27x/0.18x the runtime overhead of the only applicable secure baseline, SPT-SB. Meanwhile, it exhibits less/comparable hardware complexity.

II. BACKGROUND

This section first provides background on hardware side-channel attacks, including Spectre attacks (§II-A). Then, we state our threat model assumptions (§II-B). Lastly, we review formal hardware-software security contracts (§II-C), which we use to validate our PROTEAN prototypes (§VII-B).

A. Hardware Side-Channel Attacks

In a hardware side-channel attack, a *transmitter* (unsafe instruction in the victim program) modulates a *channel* (hardware resource) in an operand-dependent manner, and a *receiver* (attacker) observes the channel modulation to infer the operand value [66]. Many hardware resources have been implicated as channels, e.g., caches [106], [144], [145], branch predictors [1], [38], functional units [10], [47], memory ports [8], and others [45], [90], [109], [132], [133], [135], [140], [143]. An attacker observes channel modulations via their effects on nondeterministic aspects of program execution [55], e.g., execution time [16], [49], [106], resource contention [1], [8], [10], [90], [109], and more [11], [54], [69], [87], [115], [130].

Transient execution attacks [25] are hardware side-channel attacks that exploit hardware faults or mispredictions to steer program secrets toward the *sensitive* (leaky) operands of transient transmitters. A *transient* instruction is bound to squash [25]; a *sequential* instruction is bound to commit. *Speculative* instructions eventually become transient or sequential.

Prior work [25] classifies transient execution attacks as Meltdown [79] or Spectre [68] if they exploit faults or mispredictions, respectively. We focus only on Spectre, since Meltdown has efficient hardware or microcode mitigations on recent processors [58], including the Intel Alder Lake processor we model in §VIII.

B. Threat Model

We assume a powerful attacker that observes a function of the data passed to transmitters' sensitive operands (§II-B1) and can induce arbitrary speculation in the victim (§II-B2).

1) Transmitter Assumptions: Like the prior works that produced the secure baselines we evaluate (§VIII), we assume that load [32], [148] and store [32] micro-ops fully transmit their address operands when they execute, and conditional/indirect branch micro-ops [32], [148] fully transmit their condition/target operand when they resolve. Unlike these works, we also assume that division micro-ops partially transmit both of their input operands when they execute (see §VII-B4b for details).

We assume these transmitters because they collectively modulate all known channels in the base gem5 O3 CPU model [21], upon which we implement PROTEAN. Our empirical security evaluation (§VII-B) flags division micro-ops as transmitters on the gem5 O3 CPU for the first time; prior work identified the others [41], [86], [148]. Nevertheless, PROTEAN is fully parametric in the set of transmitters it considers; it can be easily extended to handle new ones present on other or future CPUs to ensure a comprehensive defense.

2) Speculation Assumptions: Like prior work [13], [46], we consider an instruction to be speculative (i.e., possibly architectural or transient) until it reaches the head of the reorder buffer (ROB) in an out-of-order processor. We refer to this definition as the ATCOMMIT *speculation model*.[1] We assume ATCOMMIT because it is the strongest speculation model studied in prior work and captures *all types of speculation*, known [2], [3], [25], [62], [63], [111] or unknown. The weakest speculation model studied in prior work is CONTROL [142]. It defines an instruction to be speculative until all prior branches have resolved, modeling control-flow speculation only. In §VIII, we evaluate all defenses for all benchmarks under comprehensive ATCOMMIT and perform one case study under noncomprehensive CONTROL (§IX-A6).

C. Hardware-Software Security Contracts

Recent work proposes formal hardware-software *security contracts*, comprising an *observer mode* and an *execution mode*, to describe which victim executions an attacker is permitted to distinguish via hardware side-channels [48]. An observer mode defines what architectural state is exposed at each victim execution step. Two observer modes, `ARCH` and `CT`, capture standard software threat model assumptions [28], [48]. `ARCH` exposes all accessed data and thus models the assumption of non-secret-accessing code (§III-A). `CT` exposes all unsafe transmitter operands and thus models the assumption of constant-time code. An execution mode defines the control-flow and data-flow paths along which a victim program exposes observations. The `SEQ` execution mode encodes sequential paths, the standard programmer assumption [48].

An *adversary model* defines how an attacker may observe channel modulations to distinguish victim program executions on a microarchitecture [48]. We adopt a powerful timing-based adversary model in §VII-B, which surfaces all transmitters assumed by our threat model (§II-B1) on the gem5 O3 CPU.

A microarchitecture *upholds* (resp. *violates*) a security contract if for all (resp. some) programs whose contract executions are indistinguishable, their microarchitectural executions

[1]The term *speculation model* denotes an *attack model* from prior work [32], [142], [148]. ATCOMMIT is equivalent to the formal definition of the FUTURISTIC speculation model [142], but not its prior implementations [31], [40], [141], which classify an instruction as speculative until all prior instructions executed without faulting or mispredicting. Such implementations would miss memory order speculation [111], for example.

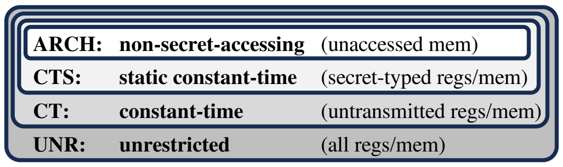

ARCH: non-secret-accessing (unaccessed mem)
CTS: static constant-time (secret-typed regs/mem)
CT: constant-time (untransmitted regs/mem)
UNR: unrestricted (all regs/mem)

Fig. 2: Four jointly exhaustive classes of vulnerable code we consider and what architectural state may hold secrets (in parentheses).

are indistinguishable (resp. distinguishable) per the adversary model. Existing comprehensive, programmer-transparent defenses uphold the `ARCH-SEQ` (NDA [138], SpecShield [13], STT [148]) or `CT-SEQ` (SPT [32]) security contracts. A defense that upholds `ARCH-SEQ` (resp. `CT-SEQ`) leaks no data beyond that which is accessed (transmitted) sequentially.

III. Motivating Programmable Spectre Defenses

This section motivates the need for a programmable Spectre defense like Protean. First, we partition vulnerable programs into four classes (§III-A). Then, we survey comprehensive, programmer-transparent Spectre defenses from the perspective of our novel notion of a ProtSet (§III-B). These defenses define their ProtSets via hardware-fixed rules, restricting them to target a single program class.

A. Spectre-Vulnerable Program Classes

Spectre attacks threaten the confidentiality of all programs that hold secret data in architectural registers (hereafter, referred to as registers) or memory. We call such programs *Spectre-vulnerable*, or simply *vulnerable*, and divide them into four increasingly broad classes (Fig. 2)—*non-secret-accessing*, *static constant-time*, *constant-time*, and *unrestricted*—forming a hierarchy, where each class is a subset of the next.

Non-secret-accessing (ARCH) code does not access secrets architecturally, i.e., it never holds secrets in registers architecturally. Spectre attacks can cause ARCH code to transiently load secrets into registers and transmit them. ARCH code includes untrusted JavaScript [44] and WebAssembly [50] running in a browser and eBPF bytecode [117] running in an OS kernel. ARCH code does not leak secrets on hardware that upholds the `ARCH-SEQ` security contract (§II-C).

Constant-time (CT) code may access but does not transmit secrets architecturally, i.e., it may hold secrets in registers that are not passed to transmitters' sensitive operands architecturally. Thus, all sensitive operands of architectural transmitters exclusively hold public data. Spectre attacks can steer architecturally or transiently accessed secrets toward transient transmitters in CT code, which includes many modern cryptographic services [4], [14], [17]–[19], [22], [27], [33], [37], [39], [89], [92], [110], [114], [120], [127], [151]. CT code does not leak secrets on hardware that upholds `CT-SEQ` (§II-C).

Static constant-time (CTS) code is constant-time code that only places secrets in registers that can be *statically typed* secret per standard secrecy typing rules [29], [97], [137]. These rules mandate that all sensitive transmitter operands are publicly-typed and all instructions with secretly-typed inputs have secretly-typed outputs. So, cryptographic declassification of secrets [122] and patterns like "`if (secret_key == 0) regenerate_key()`" are forbidden. CTS code includes explicitly-typed cryptographic code, such as code written in FaCT [29] or CT-Wasm [137] (without `declassify` operations), and untyped cryptographic code for which a valid typing is not provided but is known to exist [97], such as many primitives in OpenSSL [104] and libsodium [36].

Unrestricted (UNR) code refers to code that may architecturally access and transmit secrets, i.e., it can place secrets in any data register, including ones that are architecturally transmitted. Spectre attacks can coerce a non-constant-time program into transiently transmitting its secrets far more readily than the program does architecturally. Non-constant-time code includes cryptographic routines that are not written to be constant-time, including the OpenSSL [104] primitives evaluated in our UNR-Crypto suite (§VIII-B2). Also, much of the Linux kernel is non-constant-time, including page deduplication (e.g., Kernel Same-Page Merging [76]) or compression (e.g., Linux zswap [78] and zram [77]) modules.

Many important vulnerable applications span multiple classes—i.e., they are *multi-class* programs—because different components (e.g., libraries) and/or functions within the application process secret data in different ways. For example, the core logic for web servers like nginx [100] is non-secret-accessing [118], but it invokes static constant-time, constant-time, and unrestricted cryptographic primitives in OpenSSL to encrypt and decrypt network traffic (Fig. 1). As another example, the Linux kernel contains a mixture of non-secret-accessing (e.g., the ELF loader), constant-time (e.g., its cryptographic services), and unrestricted code. Spectre attacks can hijack *any* component of *any* class to transiently transmit secret data. Today's comprehensive, programmer-transparent Spectre defenses are not designed with these kind of multi-class programs in mind and thus exhibit performance or security limitations when defending them (Tab. I).

B. Protection Sets and Protection Mechanisms

We observe that all Spectre defenses implicitly define a *protection set (ProtSet)* and a *protection mechanism*. A defense's ProtSet may evolve as a program executes and contains, at each architectural execution step, the set of architectural state elements that the defense promises to prevent from transiently leaking. We say that a defense *protects* data in its ProtSet and *unprotects* data outside of it. A defense's protection mechanism is responsible for enforcing its ProtSet, i.e., ensuring that no data in the set transiently leaks.

A Spectre defense *secures* a particular program class (§III-A, Fig. 2) if for each architectural execution step of each program in the class, the defense's ProtSet always contains all architectural state elements that may hold secret data (in parentheses in Fig. 2). Because the four vulnerable program classes we identify form a hierarchy with respect to which subsets of architectural state may hold secrets, if a defense secures a broader program class (e.g., CT code), it also secures all narrower classes (e.g., CTS and ARCH code).

We say a defense *targets* a program class if it secures that class but not the next broader class in the hierarchy. This notion approximates how precisely a defense's ProtSet aligns with a program's protection needs. We find that defenses that target a class outperform those that merely secure it (Tab. I).

C. Hardware-Defined ProtSets (Prior Work)

Existing comprehensive, programmer-transparent Spectre defenses (Tab. I) define their ProtSets via hardware-fixed rules, so each exclusively targets *one* program class. STT [148], NDA [138], and SpecShield's [13] ProtSets always include all memory but no registers, so they target ARCH programs and secure no others (Fig. 2). SPT's ProtSet always includes all registers and memory bytes that have not been architecturally transmitted in the past by being passed to the sensitive operand of a transmitter, directly or indirectly via invertible arithmetic dependencies [32]. Thus, it targets CT programs and secures, but does not target, CTS and ARCH programs. SPT-SB's ProtSet always includes all registers and memory bytes [32]. Hence, it secures all four classes, but targets only UNR.

There is no comprehensive, programmer-transparent Spectre defense that targets multi-class programs (§III-A). Such a defense must be able to independently target each of these programs' constituent single-class components/functions, which is not possible with hardware-defined ProtSets.

D. Software-Programmable ProtSets (This Paper)

Rather than defining its ProtSet in hardware, our proposed PROTEAN Spectre defense makes its ProtSet *software-programmable*. Doing so enables it to target vulnerable programs of *all* classes, including those that are multi-class. In particular, PROTEAN consists of: ProtISA, an ISA extension that exposes a software-programmable, hardware-tracked ProtSet (§IV); ProtCC, a compiler that automatically, programmer-transparently programs it (§V); and ProtDelay and ProtTrack, two alternative hardware protection mechanisms that performantly, comprehensively enforce it (§VI).

IV. PROTISA

This section presents *ProtISA*, a novel ISA extension that exposes a programmable ProtSet (§IV-A) to software with a single new instruction prefix, PROT (§IV-B), and tracks this ProtSet in hardware with modest microarchitectural support (§IV-C). We introduce ProtISA for x86, as it is the only major ISA with instruction prefixes. However, ProtISA can be extended to work with any major ISA by storing PROT prefixes separately in an instruction metadata table [65]. Note that ProtISA simply enables software to define an architectural ProtSet for hardware to track. This is distinct from the AccessTrack/ProtTrack taint tracking protection mechanisms in §VI, which use ProtISA to identify access instructions.

A. Requirements for Programmable ProtSets

Our goal for ProtISA is to expose a ProtSet to software that compiler analyses can program automatically. Such compiler analyses (e.g., in §V-A and prior work [30], [97]) infer at each program point which accessed registers and memory bytes hold potentially secret or definitely public data. So, to program a ProtSet, these analyses need the ability to dynamically:

1) add/remove individual secret/public (a) registers and (b) memory bytes to/from the ProtSet when they are written to, i.e., protect/unprotect secret/public data when it is produced; and
2) remove individual public (a) registers and (b) memory bytes from the ProtSet when they are read from, i.e., unprotect (declassify) data that is newly inferred to be public some time after it has been produced.

Note that marking individual registers and memory bytes as protected when they are read from (i.e., classifying data some time after it has been produced) is futile and thus not a requirement. If data was previously protected, this does nothing. If previously unprotected, it may have already transiently leaked.

B. Programming a ProtSet with PROT

To satisfy the requirements above, ProtISA introduces a single new instruction prefix, PROT. *PROT-prefixed* instructions dynamically add their output registers to ProtISA's ProtSet, i.e., label them *protected*. *Unprefixed* instructions remove their output registers and any read memory bytes from the ProtSet, i.e., label them *unprotected*. Stores label written memory bytes protected/unprotected according to the protection of their data operand. ProtISA tracks protections at full register granularity; subregisters (e.g., al) inherit the protection of their corresponding full register (e.g., rax). For instructions that decode into multiple micro-ops, each micro-op inherits any PROT prefix on the instruction, and ProtISA's semantics are applied to each micro-op individually.

These simple and straightforward semantics offer surprising flexibility for software programming of a ProtSet. Below, we demonstrate how judicious use of PROT (or its absence) can achieve all required capabilities in §IV-A. We color-code protected/unprotected output registers and unprotected input memory in following examples and figures (Figs. 3 and 4).

1) Protect/unprotect register writes (req. 1a): To protect or unprotect the explicit (e.g., rax) and implicit (e.g., rflags) output registers of an instruction, PROT-prefix the instruction (left below) or do not (right).

```
PROT add rax, rdx         | add rax, rdx
// rax,rflags protected | // rax,rflags unprot
```

We conservatively handle subregister updates for unprefixed (resp. PROT-prefixed) instructions like (PROT) mov al,bl by not modifying the protection of (resp. protecting) the full output register, in this case rax.

2) Protect/unprotect memory writes (1b): The memory bytes written to by a store are protected or unprotected if its data operand is protected (left below) or unprotected (right).

```
// rax prot                 | // rax unprot
PROT add [rsp], rax         | mov [rsp], rax
// rax,[rsp],rflags prot | // rax,[rsp] unprot
```

A PROT prefix has no effect on a store unless it contains an internal load or arithmetic micro-op, which inherits the PROT prefix (left above).

3) Unprotect register reads (2a): To unprotect an instruction's input register (e.g., the data operand of the store `mov [rsp],rax` above), prepend an unprefixed identity register move (e.g., `mov rax,rax`). Register moves are cheap on modern hardware due to move elimination [60].

4) Unprotect memory reads (2b): To unprotect the memory read by a load, leave the load unprefixed (left below). However, `PROT`-prefixing a load does not protect the memory it reads (right), as doing so would be futile (§IV-A).

```
mov rax, [rsp]              | prot mov rax, [rsp]
// rax,[rsp] unprotected | // rax protected
```

C. Microarchitectural Support for ProtISA

ProtISA tracks its software-programmed, architectural ProtSet in a speculative, out-of-order microarchitecture by dynamically tagging three types of state with protection bits: rename map entries, load-store queue (LSQ) entries, and L1 data cache (L1D) lines. A set/reset protection bit indicates the tagged state is architecturally protected/unprotected.

1) Tracking Register Protection: ProtISA tracks its register ProtSet by extending each rename map entry with a protection bit. When an instruction is renamed, ProtISA updates the rename map entry of each output register with its allocated physical register identifier and marks the entry protected/unprotected if the instruction is `PROT`-prefixed/unprefixed.

2) Tracking Memory Protection: Precisely tracking memory protections would require a shadow memory [146], which is impractical [32]. Instead, ProtISA conservatively tracks its memory ProtSet through the LSQ and L1D only, as follows.

a) Protection-tagged L1D: Like SPT [32], we extend each byte in the L1D with one protection bit. Any memory bytes not present in the L1D (or LSQ) are assumed protected, i.e., L1D evictions cause ProtISA to forget what data was unprotected. L1D protection bits require $48\,\text{KiB}/8 = 6\,\text{KiB}$ per Intel Alder Lake P-core and $32\,\text{KiB}/8 = 4\,\text{KiB}$ per E-core in our evaluation (§VIII). Cacti [12] approximates the area required for a parallel L1D protection bit array for 22 nm technology (Cacti does not support Alder Lake's 10 nm) at $0.0418/0.0292\,\text{mm}^2$ for P-cores/E-cores, introducing 1.4% area overhead for a P-core/E-core's $3.0560/2.1527\,\text{mm}^2$ L1D.

b) Protection-tagged LSQ: Each LSQ entry is tagged with a single protection bit, indicating the protection of accessed memory bytes.

At execute, stores copy their data operand's protection bit into their LSQ entry's protection bit. At writeback, stores copy their LSQ entry's protection bit to the L1D protection bits corresponding to the written bytes.

Also at execute, loads either read from the L1D or forward from a store in the LSQ. An L1D load sets its LSQ entry's protection bit to the logical OR of the L1D protection bits of the bytes it reads. Forwarding loads simply copy both the protection bit from the store's LSQ entry into their own. At commit, loads with unprotected outputs clear the protection bit of all accessed memory bytes in the L1D.

D. Security Properties of ProtISA

Attackers cannot use ProtISA to architecturally unprotect victim secrets due to two key properties. First, an attacker cannot use ProtISA to architecturally unprotect (via an unprefixed instruction) architecturally inaccessible data, like data outside its sandbox. Second, ProtISA embeds the ProtSet directly into executable code (as `PROT` is an instruction prefix). An attacker cannot architecturally tamper with another program's ProtSet (e.g., by removing `PROT` prefixes or triggering out-of-bounds unprefixed loads) unless it can first overwrite or hijack victim code, a more severe security concern [125].

An attacker cannot use transient execution (e.g., by transiently jumping to an unprefixed load of secret memory) to architecturally unprotect registers or memory, since instructions only update protections on retired state when they commit. An attacker cannot exploit race conditions on shared memory accesses to improperly unprotect it, since ProtISA's memory protection tags shadow the L1D, which maintains coherence.

E. ProtISA's Interface with Protection Mechanisms

ProtISA's purpose is to enable software to tell hardware protection mechanisms (§III-B) what architectural state requires protection from (transient) hardware side-channel leakage. Thus, ProtISA must expose protection tags throughout the entire pipeline, including the out-of-order backend where transmitters execute/resolve (§II-B1). While ProtISA's tagging of the LSQ and L1D already exposes protection bits to the backend, its tagging of the rename map does not. Thus, at rename time, ProtISA also tags each renamed input and output physical register operand with its rename map entry's protection tag. Note that PROTEAN's hardware protection mechanisms (§VI) enforce that no data that ProtISA architecturally protects leaks *transiently*; future work could also prevent this data from leaking *architecturally* (akin to [146]).

V. PROTCC

This section presents *ProtCC*, a set of compiler passes that can automatically and programmer-transparently program ProtISA ProtSets to target any of the four classes of vulnerable code in Fig. 2. We explain ProtCC's passes (§V-A), compilation requirements (§V-B), and possible extensions (§V-C).

A. ProtCC Passes

ProtCC features four passes (one trivial), whose outputs are shown in Fig. 3b-e for an example program in Fig. 3a. ProtCC accommodates both single- and multi-class programs by allowing each component/function to be instrumented independently according to its corresponding (single) class (§III-A), which the user specifies via ProtCC compilation flags.

1) ProtCC-ARCH (no-op): Securing non-secret-accessing (ARCH) programs requires protecting architecturally unaccessed memory. Unmodified binaries, lacking `PROT` prefixes, naturally program this ProtSet by unprotecting only architecturally accessed memory. So, ProtCC-ARCH is a no-op pass.

```
static int A[];
int foo(int *p) {
  x = *p;
  y = 0;
  if (x >= 0)
    y = A[x];
  return y;
}
```

(a) C source

```
<foo>: # ARCH
1 mov Rx,[Rp]
2 mov Ry,0
3 cmp Rx,0 #rflags
4 jl .skip #rflags
5 mov Ry,[A+Rx]
.skip: ret
```

(b) ProtCC-ARCH

```
<foo>: # CTS
1 mov Rp,Rp
2 mov Rx,[Rp]
3 mov Ry,0
4 cmp Rx,0 #rflags
5 jl .skip #rflags
6 PROT mov Ry,[A+Rx]
.skip: ret
```

(c) ProtCC-CTS

```
<foo>: # CT
1 mov Rp,Rp
2 PROT mov Rx,[Rp]
3 mov Ry,0
4 PROT cmp Rx,0 #rflags
5 jl .skip       #rflags
6 mov Rx,Rx
7 PROT mov Ry,[A+Rx]
.skip: ret
```

(d) ProtCC-CT

```
<foo>: # UNR
1 PROT mov Rx,[Rp]
2 mov Ry,0
3 PROT cmp Rx,0 #rflags
4 jl .skip       #rflags
5 PROT mov Ry,[A+Rx]
.skip: ret
```

(e) ProtCC-UNR

Fig. 3: Assembly output for each ProtCC pass on example code. Instructions or `PROT` prefixes inserted by ProtCC are **bold**.

2) ProtCC-CTS: Recall that static constant-time (CTS) programs only place secrets in registers that can be statically typed secret per standard secrecy typing rules (§III-A). Thus, ProtCC-CTS can automatically infer a conservative secrecy typing for CTS code, *without* access to explicit secrecy labels, by: (i) initializing all registers to be secretly-typed and then iteratively (ii) applying such standard secrecy typing rules and (iii) resolving type errors by retyping the culprit register as public, until convergence. The authors of Serberus, a software Spectre defense [97], prove that this approach is guaranteed to find a conservative secrecy typing of the program (i.e., all secrets are typed as such). Specifically, our implementation of ProtCC-CTS applies type rules 2, 6, 7, 8 in Definition 4.5 of the Serberus paper. By forgoing its other type rules (which are specific to the strict subclass of CTS code Serberus considers), ProtCC-CTS may secretly-type some registers that Serberus will publicly-type, but never vice versa.

After inferring which registers are secretly-typed, ProtCC-CTS `PROT`-prefixes all instructions that define a secretly-typed register. Then, it inserts identity moves (§IV-B3) to architecturally unprotect each publicly-typed argument at function entry, ensuring all registers that may hold secrets are protected. To see why all secret memory is protected, recall again that memory can only be unprotected by an architectural load/store to/from an unprotected and therefore publicly-typed output/data register. Such memory must also hold public data.

Example (Fig. 3c): ProtCC-CTS initially types `Rx` secret, but changes it to public, because it is passed to the transmitter on line 6 (a load) along some control-flow path. At convergence, ProtCC-CTS types the definitions of `Rp`, `Rx`, `Ry` (line 3) as public and the definition of `Ry` (line 6) as secret. Thus, ProtCC-CTS inserts an identity move to unprotect `Rp` on line 1 and `PROT`-prefixes the (re)definition of `Ry` on line 6.

3) ProtCC-CT: Recall that constant-time (CT) programs never place secrets in registers that are architecturally fully (or partially) transmitted (§II-B1). Thus, ProtCC-CT can safely unprotect all registers that were fully transmitted in the *past* or will be in the *future* architectural execution. Based on this observation, ProtCC-CT implements two register-level data-flow analyses to compute the set of registers at each architectural program point that already fully leaked (or hold constant data) along all prior control-flow paths or are bound to leak along all future control-flow paths.

ProtCC-CT then (i) `PROT`-prefixes all instructions with any output register (including `rflags`) that is neither past-leaked nor bound-to-leak and (ii) inserts an identity move (§IV-B3) along every control-flow edge on which a register becomes newly bound-to-leak in order to label it unprotected. Case (i) ensures all registers that may hold secrets are protected, and case (ii) architecturally declassifies a previously protected register once it must hold public data.

Example (Fig. 3d): ProtCC-CT analyses find that `Rp` is bound-to-leak on function entry and `Rx` becomes bound-to-leak along the not-taken direction of the branch (line 5→6) since it will be architecturally transmitted by the load on line 7. Thus, ProtCC-CT inserts identity moves on lines 1 and 6 to unprotect `Rp` and `Rx`, respectively. `Ry` becomes past-leaked after line 3, since it holds constant data. However, `Ry` is no longer past-leaked when overwritten by the load on line 7; thus, ProtCC-CT `PROT`-prefixes it.

4) ProtCC-UNR: Recall that unrestricted programs may place secrets in any data register, including those that are fully transmitted (§III-A). Thus, we can only unprotect registers that *never* hold secret program data. Such registers include the *stack pointer* (e.g., `rsp` on x86), registers initialized with a constant, and registers computed solely from them (e.g., the frame pointer, or loop indices starting at 0). ProtCC-UNR `PROT`-prefixes all instructions with output registers except those whose output registers are one of the aforesaid registers.

Example (Fig. 3e): ProtCC-UNR only unprotects the zero-initialization of `Ry` on line 2, but `PROT`-prefixes all other instructions and thus protects all other registers and memory.

B. ProtCC Compilation Requirements

All secret-accessing code loaded into the address space, including any secret-accessing shared libraries and JITed code, must be compiled with the appropriate ProtCC pass. This protects all architectural state that may hold secret data. Failing to recompile such code allows secret data to leak transiently, even with PROTEAN's defenses enabled. Non-secret-accessing code does not require recompilation (§V-A1).

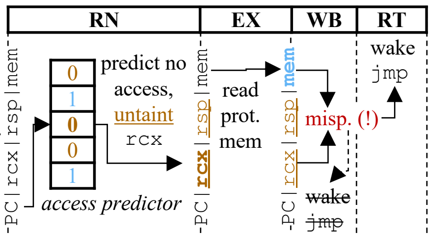
Code of an unknown class has multiple compilation options. For users that want PROTEAN to *guarantee* that no secrets will leak, it must be compiled with ProtCC-UNR, at a high performance cost (§IX). Users willing to permit transient leakage of a *formally bounded* amount of secret information for performance can compile code with ProtCC-CT to uphold `CT-SEQ` (§II-C), ProtCC-CTS to uphold `CTS-SEQ` (§VII-B), or forgo recompilation to uphold `ARCH-SEQ` (§II-C).

C. Possible ProtCC Extensions

Users willing to trade programmer transparency for additional performance can refine the ProtSets inferred by ProtCC

```
1 PROT mov rax,rdx
2 PROT add rax,1
3 mov rcx,[rsp]
4 jmp rcx
5 leak rax
```

(a) insecure, slow code under AccessDelay and AccessTrack



(b) ProtTrack's access misprediction path

Fig. 4: ProtDelay/ProtTrack, but not AccessDelay/AccessTrack, secure the code in (a) by stalling the transient access transmitter (line 5). For performance, ProtDelay speculatively wakes up line 2, and ProtTrack predicts line 3 is not an access, safely handling mispredictions as shown in (b).

through manual annotations or custom ProtCC passes. For the former, ProtCC can be extended to support the explicit annotation of variables, expressions, and memory as public. For the latter, users can either extend one of ProtCC's existing passes or provide a new pass entirely.

VI. ProtDelay and ProtTrack

This section introduces the *ProtDelay* and *ProtTrack* hardware protection mechanisms, which Protean uses to performantly and comprehensively enforce ProtISA ProtSets. ProtTrack optimizes for performance at the cost of higher hardware complexity; ProtDelay makes the opposite tradeoff.

A. Applying Access-Based Defenses to ProtISA ProtSets

ProtDelay and ProtTrack extend state-of-the-art hardware protection mechanisms that *delay* waking up the dependents of (AccessDelay, §VI-A1) or *taint-track* the outputs of (AccessTrack, §VI-A2) speculative *access instructions*, respectively. AccessDelay and AccessTrack (our terms) define access instructions as those that read potential secrets into output registers [148] and typically assume they are loads [13], [138], [148]. For ProtISA hardware, we define them as follows:

Definition 1. On ProtISA hardware, *access instructions* are instructions with protected register or memory inputs.[2] *Access transmitters* are both access instructions and transmitters (§II-A) and have a protected sensitive input.

As we discuss next, applying AccessDelay or AccessTrack *directly* to ProtISA's access instruction definition results in a partial and nonperformant defense, since these mechanisms were not designed with programmable ProtSets in mind.

1) AccessDelay: NDA and SpecShield propose the *AccessDelay* protection mechanism [13], [138], which allows all access instructions to execute and write back speculatively, but forbids them from waking up dependents until the access becomes non-speculative. This prevents transient access instructions (e.g., the `add` in Fig. 4a) from propagating their protected data inputs to dependent transmitters (e.g., the `leak` in Fig. 4a), which is sufficient to secure non-secret-accessing programs (NDA's and SpecShield's target). However, it does not prevent access transmitters (e.g., `leak rax`) from transmitting their own protected sensitive input (e.g., `rax`), a security violation for ProtISA programs, which can access secrets.

Furthermore, AccessDelay unnecessarily blocks speculative `PROT`-prefixed access instructions (e.g., `PROT mov rax,rdx` in Fig. 4a) from waking their non-transmitter dependents (e.g., `PROT add rax,1`), a performance issue. Such dependents can safely execute speculatively, since they re-access the protected register produced by the `PROT`-prefixed access. That is, such dependents are themselves speculative access instructions, whose wakeup of dependents AccessDelay will delay as needed.

2) AccessTrack: STT proposes the *AccessTrack* protection mechanism, which taints the output registers of access instructions and their dependents at rename, untainting them once the youngest access instruction they depend on becomes non-speculative [148]. AccessTrack delays the execution/resolution of transmitters with tainted sensitive inputs (§II-B1) until untainted. Like AccessDelay, AccessTrack allows access transmitters to transiently transmit untainted, yet protected, input registers, which do not arise in non-secret-accessing programs (STT's target) but do arise in ProtISA programs. For example, AccessTrack taints output `rax` of the access instruction `PROT add rax,1` and thus delays `leak rax`'s transmission of its tainted input. However, AccessTrack untaints `rax` once the `add` retires, allowing `leak` to transiently transmit its untainted but still protected input, `rax`, a security violation for ProtISA.

Plus, while AccessTrack requires identifying access instructions at rename, ProtISA cannot determine whether loads read protected memory until they execute. So, AccessTrack must conservatively taint the outputs of *all* loads. Its untaint broadcast logic [148] can further only untaint output registers at commit, preventing early recovery from false positives that read unprotected memory. Both are performance issues.

B. Adapting Access-Based Defenses for ProtISA ProtSets

ProtDelay and ProtTrack augment AccessDelay and AccessTrack, respectively, to extend their security guarantees to and improve their performance on ProtISA programs, as follows.

1) ProtDelay: For security, ProtDelay extends AccessDelay to additionally delay the transmission of access transmitters' protected sensitive operands, until they are non-speculative.

For performance, ProtDelay relaxes AccessDelay's wakeup delay logic to only delay wakeup of the dependents of unprefixed, but not `PROT`-prefixed, accesses. This is safe because ProtDelay will delay these dependent access instructions anyway, either in their execution/resolution if they are access transmitters or in their wakeup of dependents otherwise.

2) ProtTrack: For security, ProtTrack extends AccessTrack to also delay the speculative execution/resolution of access transmitters, like ProtDelay. SPT augments AccessTrack in the same way to target constant-time programs [32].

[2] ProtISA's definition of access instruction admits stores, whereas prior work's does not [148] because stores lack an output register.

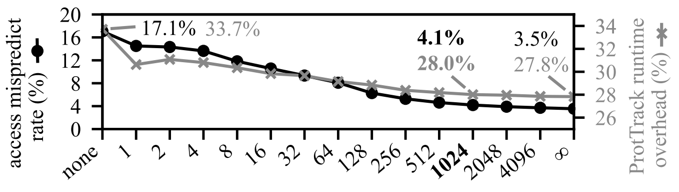


Fig. 5: ProtTrack access predictor sensitivity study (x-axis: number of predictor entries; we choose $n = 1024$). The access misprediction rate is the fraction of retired non-`PROT`-prefixed loads with unprotected outputs for which ProtTrack's access predictor guesses incorrectly at rename whether the load reads protected memory at execute. ProtTrack's runtime overhead is averaged across ProtCC-ARCH-/ProtCC-CT-compiled SPEC2017int benchmarks on our P-core configuration (§VIII-A), normalized to an unsafe baseline.

For performance, ProtTrack untaints loads that are predicted to read unprotected memory (§VI-B2a), ensuring security by falling back to ProtDelay when the prediction is wrong or an untainted load forwards from a tainted store.

a) Secure Access Predictor: ProtTrack uses a simple 1-bit predictor table with $n = 1024$ untagged entries, indexed by the 10 least significant bits of load program counters (PCs). Each entry stores whether a load accessed protected memory the last time it executed. The total predictor size is $1024/8 = 128$ bytes. We choose $n = 1024$ entries because our predictor sensitivity study (Fig. 5) shows that it attains a misprediction rate within 0.6% and a runtime overhead within 0.2% of an infinitely-sized predictor.

ProtTrack predicts whether a load is an access instruction at rename time as follows. First, it indexes into the access predictor. If the load's entry is *no-access* and its output is unprotected (computed in parallel by ProtISA also at rename, §IV-C1), ProtTrack predicts the load will not be an access instruction and predictively untaints the load's output (Fig. 4b). Otherwise, ProtTrack taints the load's output, as it is predicted to read protected memory.

b) Secure Misprediction Recovery: ProtTrack's access predictor can generate two types of mispredictions: *false negatives*, which predict *no-access* for a load that reads protected memory and are a security risk; and *false positives*, which predict *access* for a load that reads unprotected memory but are benign aside from reducing performance. ProtTrack handles false negatives by falling back to ProtDelay, i.e., delaying the wakeup of all non-access dependents until the load retires. This ensures mispredictions never cause data to speculatively propagate from protected memory to an untainted, unprotected register. Empirically, both types of mispredictions are rare (Fig. 5). We update the predictor with the load's actual access outcome (1 for *access*, 0 for *no-access*) when it retires.

c) Secure Tainted Store Forwarding: ProtTrack also falls back to ProtDelay for untainted loads that forward from stores of tainted data, but only stalls the load's wakeup of dependents until the store's data operand becomes untainted (rather than until the load commits). This prevents attackers from circumventing ProtTrack's register taint tracking by speculatively storing and reloading tainted data to/from memory.

3) Threat Model Considerations: ProtDelay and ProtTrack comprehensively block all transient leakage of protected data via all hardware side channels due to all types of speculation, since they are fully parametric in their assumed transmitters and adopt the ATCOMMIT speculation model (§II-B).

VII. SECURITY EVALUATION

This section evaluates PROTEAN's security. First, we sketch a proof that for all single- or multi-class programs (§III-A), PROTEAN blocks all transient leakage of secrets and thus all Spectre attacks [25] (§VII-A). Then, we empirically validate that our gem5 [21] prototypes of PROTEAN hardware uphold one security contract for arbitrary ProtISA binaries and one of three others for single-class ProtCC binaries (§VII-B).

A. Formal Security Proof Sketch

ProtCC ensures that the architectural ProtISA ProtSet always contains all architectural state that may hold secret data:

Lemma 1. *Given a program and a class-labeling of its functions/components, ProtCC programs ProtISA (via* `PROT` *prefixes) to always protect all architectural state that may hold secret data in all of the program's architectural executions.*
Proof sketch. See class-specific arguments in §V-A.

Then, ProtISA's hardware support tags protected all architectural state in ProtISA's architectural ProtSet (and thus all secret data, by Lem. 1):

Lemma 2. *The set of retired architectural registers and memory bytes that ProtISA's hardware support tags protected is a superset (due to implicitly tagging memory bytes outside the LSQ/L1D as protected) of those in ProtISA's ProtSet.*
Proof sketch. See ProtISA's hardware implementation (§IV-C).

Lastly, ProtDelay/ProtTrack block the transient transmission of all architectural state that is tagged protected by ProtISA's hardware support (and thus all secret data, by Lem. 2):

Lemma 3. *ProtDelay/ProtTrack prevent the transient transmission of data derived from retired protected-tagged state.*
Proof sketch. Such data can only leak via a transmitter's sensitive operand. If protected, ProtDelay/ProtTrack block its transmission until the transmitter retires (§VI-B). If unprotected, such data must depend on a speculative access instruction with an unprotected output. In this case, ProtDelay/ProtTrack block the data's transmission until the access instruction retires—at which point the data is officially architecturally unprotected.

Theorem 4. *Given a program and a class-labeling of its functions/components,* PROTEAN *prevents all transient leakage of secrets in all of the program's speculative executions.*
Proof sketch. By Lems. 1 and 2, the ProtISA microarchitecture tags protected all retired architectural state holding secret data. By Lem. 3, ProtDelay/ProtTrack prevent all retired protected state from leaking transiently. Thus, the combination, PROTEAN, prevents all transient leakage of secrets.

Contract	Instrumentation	Violations, out of 200		
		Unsafe	PROTEAN	
			ProtDelay	ProtTrack
`UNPROT-SEQ`	ProtCC-RAND	**189** (1)	**0** (0)	**0** (0)
`ARCH-SEQ`	ProtCC-ARCH	**200** (0)	**0** (2)	**0** (5)
`CTS-SEQ`	ProtCC-CTS	**200** (0)	**0** (1)	**0** (0)
`CT-SEQ`	ProtCC-CT	**200** (0)	**0** (16)	**0** (10)
`CT-SEQ`	ProtCC-UNR	**199** (1)	**0** (20)	**0** (17)

TABLE II: AMuLeT*-detected contract violations for ProtCC-RAND/-ARCH/-CTS/-CT/-UNR test binaries on unsafe, ProtDelay, and ProtTrack hardware. False positives (§VII-B1) are in parentheses.

B. Empirical Security Evaluation with AMuLeT*

We empirically assess the security of our gem5 prototypes of PROTEAN (ProtISA with ProtDelay/ProtTrack) and those of the secure baselines (STT [40], SPT [31], and SPT-SB [31]) that we compare PROTEAN's performance to in §VIII.

1) AMuLeT:* We conduct this evaluation with AMuLeT*,[3] which extends AMuLeT [41], a fuzzing framework for testing gem5 implementations of Spectre defenses against §II-C-style security contracts, with the following enhancements:

a) New test generator: Adds a new LLVM-IR test generator based on llvm-stress [85]. Unlike the assembly tests that AMuLeT generates, these IR tests can be compiled by ProtCC.

b) Enhanced `CT` observer mode: Extends AMuLeT's `CT` observer mode, which exposes instructions' PCs and accessed addresses, to also expose individual address registers,[4] matching our and prior defenses' transmitter assumptions (§II-B1).

c) New observer modes: Adds new `UNPROT` and `CTS` observer modes, which further extend `CT` to also expose all data held in ProtISA-unprotected and publicly-typed (§III-A) registers, respectively. These observer modes enable testing PROTEAN against the `UNPROT-SEQ` and `CTS-SEQ` contracts.

d) New adversary model: Adds a new adversary model (§II-C) that exposes to an attacker the cycle at which each instruction reaches each pipeline stage. This adversary model surfaces contract violations in STT/SPT/SPT-SB that AMuLeT's default cache+TLB adversary model does not, since it exposes fine-grained secret-dependent timing information observable to SMT receivers.

e) False positive detection: Largely automates detection of false-positive contract violations (§II-C) during post-processing. While AMuLeT filters false positives during fuzzing, it does not do so during post-processing. AMuLeT* classifies a contract violation as a false positive if the committed microcode sequences of the violating microarchitectural executions differ in their PCs or accessed addresses—indicating *sequential*, not transient, leakage. Such cases arise due to AMuLeT bugs or identical committed instruction sequences taking divergent microcode paths.

2) Experimental Setup: We use AMuLeT* to test the security of a series of (gem5 hardware configuration, ProtCC pass, security contract) triples. For each triple, we run 200 parallel instances of AMuLeT*, with each instance configured to randomly generate 200 LLVM-IR programs, instrumented with the triple's ProtCC pass. For 100 of the AMuLeT* instances, we test 140 random inputs per program under AMuLeT*'s default adversary model, which exposes data cache and TLB tags. For the other 100 instances, we test 5 inputs per program under our timing-based adversary model (§VII-B1). In total, we test up to $100 \times 200 \times (140 + 5) = 2,900,000$ executions per triple. Each AMuLeT* instance exits upon detecting its first contract violation or completing all tests.

3) Evaluated Hardware Configurations: To validate AMuLeT* itself, we use it to test the unmodified gem5 O3 CPU (base commit `8381e1c`), our unsafe baseline in §VIII.

To validate the security of the gem5 implementations of PROTEAN and the secure baselines, we base PROTEAN's and rebase STT/SPT/SPT-SB's onto the same aforesaid commit and configure all to adopt the ATCOMMIT speculation model (§II-B2). We then run AMuLeT* on them all in two stages. First, we configure all defenses to assume that loads, stores, and branches are transmitters, as these are the only gem5 transmitters documented in prior work (§II-B1). During early testing, AMuLeT* discovers a new transmitter (division micro-ops) and a new bug in all defense implementations. So, we extend all defense implementations to treat division micro-ops as transmitters and also apply a bug fix to them. Then, we rerun AMuLeT* on the fixed defense implementations, which we evaluate in §VIII.

4) Experimental Results: Tab. II presents final testing results for the unsafe baseline and PROTEAN specifically. We describe AMuLeT*'s findings for all experiments next.

a) Unsafe baseline: We use AMuLeT* to test the unsafe baseline against all security contracts with all ProtCC passes. As expected, it detects hundreds of violations for each contract (Tab. II). We expect most are true positives, since AMuLeT* filters many false positives (§VII-B1); however, due to their sheer quantity, we do not manually validate them all as such.

b) PROTEAN: We validate that PROTEAN upholds `UNPROT-SEQ` for ProtISA binaries and `ARCH-`/`CTS-`/`CT-SEQ` for ProtCC-ARCH/-CTS/-CT binaries. To obtain random ProtISA binaries for testing against `UNPROT-SEQ`, we compile each test with a ProtCC pass (designed for testing) that `PROT`-prefixes a random subset of instructions (ProtCC-RAND in Tab. II). PROTEAN upholds a stronger security property on ProtCC-UNR binaries than any §II-C-style security contract can capture, as such contracts assume sequentially-leaked data is always public, but in unrestricted code, it may be secret (§III-A). So, we test PROTEAN against `CT-SEQ`, the strongest contract implemented in AMuLeT*, on ProtCC-UNR binaries.

Early testing results: While testing an early implementation of PROTEAN, AMuLeT* uncovered two security issues that also impact all three secure baselines (STT/SPT/SPT-SB).

First, it detected that division micro-ops leak a function of their n-bit divisor and $2n$-bit dividend on the gem5 O3 CPU by conditionally faulting if the divisor is zero or the $2n$-bit quotient overflows the n-bit output register. No prior work has found that division micro-ops are transmitters on gem5.

[3] AMuLeT*: https://github.com/StanfordPLArchSec/protean-amulet.git.

[4] x86 memory operands can have two address registers, e.g., `mov rax, [rsp+rdi]`. AMuLeT exposes the sum `rsp+rdi`; AMuLeT* exposes `rsp` and `rdi` individually.

Second, AMuLeT* found a previously unknown corner-case bug in Protean's implementation of delayed branch resolution, which it borrows directly from STT/SPT/SPT-SB. In this implementation, the execute stage notifies the commit stage of the youngest branch misprediction detected in that cycle, regardless of whether it is tainted/protected or untainted/unprotected. As a result, an older tainted/protected branch can conditionally block a younger untainted/unprotected branch from initiating a squash depending on whether the former was mispredicted. This creates a secret-dependent squash signal that leaks the tainted/protected branch predicate. To restore security, we patched Protean and upstreamed fixes to STT [96] and SPT/SPT-SB [93] to instead separately notify the commit stage of the youngest *untainted/unprotected* branch misprediction and *all* tainted/protected branch mispredictions detected a given cycle.

Final testing results: AMuLeT* finds zero true-positive contract violations in our final, fixed gem5 implementation of Protean, out of 28.3 million executed tests. We evaluate the performance of this implementation of Protean in §VIII.

c) STT, SPT, and SPT-SB: Like prior work [41], we test STT and SPT against the `ARCH-SEQ` and `CT-SEQ` contracts, respectively, on unmodified binaries. We also test SPT-SB against `CT-SEQ` on unmodified binaries, for reasons discussed in §VII-B4b. We set a four-hour time limit for AMuLeT* instances, due to STT/SPT/SPT-SB's high host runtimes.

Early testing results: AMuLeT* finds 9/31/65 true-positive contract violations for STT/SPT/SPT-SB before patching them to fix the two security issues found in §VII-B4b.

Manual inspection results: We manually discover a corner-case implementation bug in SPT that AMuLeT* misses, which causes the defense to improperly untaint the output register and input memory of secret-accessing transient loads following a pending fault. SPT's implementation untaints the output registers of loads (and all other instructions) at rename before later tainting them upon insertion into the ROB. When a fault reaches commit and initiates a machine clear, the O3 CPU stops inserting dispatched (now transient) instructions into the ROB. Crucially, however, it may still insert loads into the LSQ and execute them. Because these loads bypass ROB insertion, their outputs remain incorrectly untainted. So, when they compute their address, SPT's shadow L1 propagation logic erroneously untaints the accessed memory bytes in the L1D, which may contain secret data.

AMuLeT* misses this likely because our LLVM-IR instrumentation for suppressing architectural faults (like AMuLeT's) has the unintentional side effect of suppressing most transient faults too. Suppressing architectural faults is needed to prevent our randomly-generated tests from crashing gem5.

To restore security, we upstream and locally apply a patch to SPT that unconditionally taints all output registers at rename [94]. This ensures that the outputs of any transient instructions bypassing the ROB are conservatively tainted, avoiding the above issue, while still allowing SPT to assign the correct taint for instructions that successfully enter the ROB.

Parameter	P-core	E-core
Clock frequency	3.4GHz	2.5GHz
Cores	8 cores	8 cores
Pipeline width	6-way fetch/issue/decode/rename	
ROB/LQ/SQ size	512/192/114 entries	256/80/50 entries
Predictors	4K-entry BTB, 16-entry RSB, TAGE BP	
Physical register file	280/332 int/FP regs.	213/207 int/FP regs.
L1D cache (private)	48KiB, 12-way	32KiB, 8-way
L1I cache (private)	32KiB, 8-way	64KiB, 8-way
L2 cache (private)	1.25MiB, 10-way	256KiB/2MiB,† 8-way
L3 cache (shared)	30MiB, 12-way	
Coherence protocol	Directory-based 3-level MESI protocol	

TABLE III: Our gem5 processor configuration resembling an Intel Alder Lake hybrid processor. Pipeline parameters (e.g., ROB size) are taken from prior studies [72], [73]. †gem5 does not readily support Alder Lake's mixed private/shared L2 for P-/E-cores, so we model both as private but assign each E-core a 2 MiB L2 for single-threaded workloads and a 256 KiB slice for multi-threaded ones.

Our SPT taint bug fix uncovers a latent performance issue in its implementation: 32-bit register writes (e.g., `mov eax,42`) implicitly zero the upper 32 bits of the full 64-bit register (e.g., `rax`) but do not untaint them, so subsequent uses of the full register (e.g., `mov [rsp+rax],0`) are considered tainted, degrading performance. To restore performance, we upstream and locally apply a patch to SPT that untaints the upper 32 bits of a register whenever its lower 32 bits are written to [95].

Final testing results: After applying the bug fixes in §VII-B4b and §VII-B4c, AMuLeT* finds zero contract violations for STT, SPT, and SPT-SB after running 2.7/0.3/1.2 million tests. We evaluate the performance of these fully patched secure baseline implementations in §VIII.

VIII. Performance Evaluation

In this section, we show that Protean outperforms state-of-the-art comprehensive, programmer-transparent defenses when securing a variety of vulnerable programs.

A. Experimental Setup

We implement ProtCC as a machine IR pass in LLVM [82].[5] We implement ProtISA and ProtDelay/ProtTrack for the gem5 O3 speculative, out-of-order processor [21].[6] We obtain a performance comparison between Protean and a given baseline on a given benchmark as follows.

1) Compilation: First, we use LLVM Clang and Flang 17 [83], [84] to compile each benchmark with and without the appropriate ProtCC pass (§V-A). We refer to the former as the *ProtCC binary* and the latter as the *base binary*. In the special case where the class is non-secret-accessing, the base binary is the ProtCC binary (§V-A1). We similarly compile base and ProtCC versions of LLVM's libc and libc++ libraries [80], [81] and statically link them with each benchmark as appropriate.

2) Processor Configuration: We configure gem5's O3 CPU to resemble a 12th Gen Intel® Core™ i9-12900KS Alder Lake processor [57], which features a hybrid architecture with 8 Golden Cove performance cores (*P-cores*) and 8 Gracemont efficiency cores (*E-cores*). See Tab. III for details.

[5]ProtCC: https://github.com/StanfordPLArchSec/protean-llvm.
[6]Protean gem5: https://github.com/StanfordPLArchSec/protean-gem5.

ARCH		STT	**PROTEAN**		CTS		SPT	**PROTEAN**	
			Delay	**Track**				**Delay**	**Track**
SPEC	*P-core*	1.369	1.299	1.089	*SPEC*	*P-core*	1.708	1.388	1.259
2017	*E-core*	1.143	1.110	1.051	*2017*	*E-core*	1.302	1.130	1.100
PARSEC		1.169	1.022	1.007	*PARSEC*		1.409	1.177	1.076
CT		**SPT**	**PROTEAN**		**UNR**		**SPT-SB**	**PROTEAN**	
			Delay	**Track**				**Delay**	**Track**
SPEC	*P-core*	1.708	1.527	1.422	*SPEC*	*P-core*	2.949	2.870	2.812
2017	*E-core*	1.302	1.212	1.176	*2017*	*E-core*	2.081	2.041	2.002
PARSEC		1.409	1.270	1.191	*PARSEC*		3.040	2.125	2.088

TABLE IV: Geometric mean normalized runtime of all eight of PROTEAN's single-class configurations on SPEC2017 and PARSEC.

3) Single-Thread Simulation: We simulate single-thread workloads on a single P-core or E-core using the SimPoints methodology [121]. We select up to 10 representative 50-million-instruction intervals (*simpoints*) for the base binary. We then use the PinCPU methodology [98] to translate each simpoint for the base binary into the equivalent simpoint of the ProtCC binary that corresponds to the same application region, using source locations as a progress marker. This ensures that we evaluate PROTEAN on the same part of the benchmark as our baselines. We compile both the base and ProtCC binaries with `-O2 -g` to ensure the availability of source locations.

We resume non-PROTEAN baselines from the base binary's simpoints and PROTEAN from the ProtCC binary's translated simpoints. To resume from a simpoint, we warm up each simulated design for 10 million instructions before the instruction interval begins, and then record execution statistics until the instruction interval ends. Finally, we take a weighted sum of each simpoint's simulated execution time to obtain the benchmark's estimated overall execution time.

4) Multi-Thread Simulation: We simulate multi-thread benchmarks end-to-end on a full Alder Lake configuration, with 8 P-cores and 8 E-cores (Tab. III).

5) Evaluated Spectre Defenses: We evaluate the unmodified O3 CPU (our unsafe baseline) and state-of-the-art comprehensive, programmer-transparent Spectre defenses with hardware-defined ProtSets (our secure baselines)—STT, SPT, and SPT-SB—on base binaries. We omit NDA and SpecShield (§III-C), since STT is more performant [13], [138], [148]. We run all evaluated defenses under the comprehensive AT-COMMIT speculation model (§II-B2), unless otherwise stated.

We evaluate PROTEAN using either its ProtTrack or ProtDelay protection mechanism, running ProtCC-ARCH/-CTS/-CT/-UNR binaries, for eight total defense configurations. We suffix PROTEAN to denote the protection mechanism in use and which ProtCC pass the binary was compiled with (e.g., PROTEAN-Track-CT for ProtTrack on a ProtCC-CT binary). We omit a protection mechanism (e.g., PROTEAN-CT) to denote both mechanisms paired with the same ProtCC pass (e.g., PROTEAN-Delay-CT and PROTEAN-Track-CT).

We run the rebased implementations of STT, SPT, and SPT-SB described in §VII-B3, with all security and performance fixes described in §VII-B4b and §VII-B4c applied. We evaluate the performance impact of these fixes in §IX-A7.

B. Experiments

We evaluate PROTEAN on seven general-purpose, single-class, and multi-class benchmark suites.

1) General-Purpose (SPEC2017, PARSEC): To gauge PROTEAN's performance when securing general-purpose code, we evaluate all PROTEAN configurations and all baselines on the single-thread SPEC CPU® 2017 speed (SPEC2017) [24] benchmarks with *reference*-size inputs and the multi-thread PARSEC [20] benchmarks with *simsmall*-size inputs. We omit the `cam4` SPEC2017 benchmark which exceeds a 2-day timeout and `roms` due to a gem5 crash. We omit PARSEC's `bodytrack`, `facesim`, `freqmine`, `raytrace`, `vips`, `x264`, and `streamcluster` due to gem5 crashes, compile errors, and 5-day timeouts.

2) Single-Class: We evaluate PROTEAN on representative single-class workloads to demonstrate that PROTEAN outperforms all state-of-the-art defenses that target each class.

ARCH-Wasm: We compare PROTEAN-ARCH to STT on the subset of the SPEC CPU 2006 benchmarks that compile to sandboxed WebAssembly [50], on *reference*-size inputs. We use wasi-sdk [136] to compile to WebAssembly, wasm2c to transpile it to C, and Clang to compile it to a native executable.

CTS-Crypto/CT-Crypto: We compare PROTEAN-CTS/-CT to SPT on cryptographic primitives that are static constant-time (from [97]) and constant-time (from [32]).

UNR-Crypto: We compare PROTEAN-UNR to SPT-SB on three OpenSSL cryptographic primitives—modular exponentiation (`ossl.bnexp` in Tab. V), Diffie-Hellman key exchange (`ossl.dh`), and elliptic curve addition (`ossl.ecadd`)—that are *not* constant-time (curated by us).

3) Multi-Class (NGINX): To showcase how ProtCC enables PROTEAN to performantly secure complex multi-class applications, we compare PROTEAN to SPT-SB on the nginx HTTPS web server [100], which mixes all four classes (Fig. 1). The main nginx executable does not access secrets, so we compile it with ProtCC-ARCH (§V-A1). Instead, it delegates secret computation to OpenSSL [104], which contains all classes of code (including non-constant-time, §III-A). We compile OpenSSL with ProtCC-UNR for security, except for the 2/7/1 hottest ARCH/CTS/CT functions, which we identify via runtime profiling, classify manually by examining the source, and compile with ProtCC-ARCH/-CTS/-CT for performance. Like prior work [118], we use `siege` [124], running directly on the host, to connect a variable number of clients c, each issuing a varying number of requests r, to the gem5-simulated server.

IX. PERFORMANCE RESULTS

Fig. 6, Tab. IV, and Tab. V present the results to the experiments described in §VIII.

A. General-Purpose Workloads (Fig. 6, Tab. IV)

Fig. 6 plots per-benchmark normalized runtimes for PROTEAN-Track-ARCH/-CT and STT/SPT on SPEC2017 (on a P-core) and PARSEC. Tab. IV reports the geomean normalized runtimes for all defenses and core types. On a P-core, PROTEAN-Track-ARCH/-CT averages less than one-fourth/two-

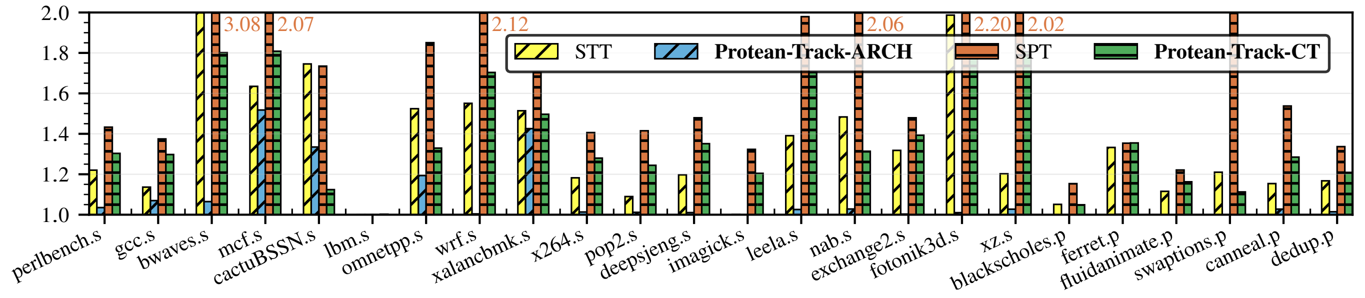


Fig. 6: Normalized runtime of **PROTEAN-Track-ARCH/-CT** versus STT/SPT on the SPEC2017 benchmarks (*.s) on a P-core and PARSEC (*.p) benchmarks on the full Alder Lake configuration.

ARCH-Wasm	**STT**	**PROTEAN**		**CT-Crypto**	**SPT**	**PROTEAN**	
		Delay	**Track**			**Delay**	**Track**
bzip2	2.771	1.201	1.110	bearssl	1.444	1.278	1.265
mcf	3.270	1.429	1.333	ctaes	1.372	1.129	1.006
milc	3.729	1.212	1.017	djbsort	1.264	1.256	1.238
namd	2.860	1.042	1.006	geomean	1.358	1.219	1.163
libquantum	2.002	1.344	1.003	**UNR-**	**SPT-**	**PROTEAN**	
lbm	1.364	1.001	1.001	**Crypto**	**SB**	**Delay**	**Track**
geomean	2.533	1.195	1.072	ossl.bnexp	2.590	2.628	2.624
CTS-Crypto	**SPT**	**PROTEAN**		ossl.dh	3.137	2.566	2.548
		Delay	**Track**	ossl.ecadd	2.897	2.916	2.913
hacl.chacha20	1.008	1.068	0.999	geomean	2.866	2.699	2.690
hacl.curve25519	1.026	1.071	0.991	**Multi-Class**	**SPT-**	**PROTEAN**	
hacl.poly1305	1.158	1.000	1.001	**Web Server**	**SB**	**Delay**	**Track**
sodium.salsa20	1.000	1.115	1.115	nginx.c1r1	2.746	1.569	1.408
sodium.sha256	1.174	1.073	1.073	nginx.c2r2	2.776	1.469	1.310
ossl.chacha20	1.096	1.002	1.002	nginx.c1r4	2.778	1.472	1.311
ossl.curve25519	1.109	1.118	1.101	nginx.c4r1	2.775	1.473	1.311
ossl.sha256	1.367	1.062	1.061	nginx.c4r4	2.787	1.428	1.267
geomean	1.112	1.063	1.042	geomean	2.772	1.481	1.321

TABLE V: Normalized runtime of PROTEAN on representative single-class and multi-class workloads, simulated on a P-core.

thirds the overhead of STT/SPT, demonstrating how PROTEAN can more performantly uphold established security contracts (`ARCH-SEQ` and `CT-SEQ`, §II-C). While less performant than PROTEAN-Track, PROTEAN-Delay outperforms all baselines on average in each class and core configuration, despite ProtDelay's comparative hardware simplicity. We investigate class-specific reasons *why* PROTEAN secures single-class programs more performantly than prior defenses in §IX-B.

1) SPT-SB vs. PROTEAN-*UNR:* Notably, SPT-SB incurs a 3.0x slowdown on PARSEC (Tab. IV), whereas PROTEAN-UNR incurs a much smaller 2.1x slowdown. To determine why, we studied `blackscholes.p`, in which SPT-SB and PROTEAN exhibit the largest gap in slowdown factor (3.4x vs. 1.2x), and found that all top ten transmitters stalled by SPT-SB are fixed-offset stack accesses (e.g., `mov rax, [rsp]` or `ret`)—instructions that ProtCC-UNR *avoids* stalling by unprotecting the stack pointer (§V-A4).

2) ProtCC Overhead: We evaluate the runtime and code size overhead of compiling with ProtCC's three non-trivial passes (§V-A) on the SPEC2017int benchmarks running on a P-core. We find that ProtCC-CTS/-CT/-UNR binaries average 8.1%/20.2%/6.0% code size overhead and 2.7%/5.3%/1.3% runtime overhead when PROTEAN's protections are disabled. ProtCC-CT binaries exhibit high code size and runtime overhead primarily because they insert more identity moves than ProtCC-CTS (ProtCC-ARCH/-UNR do not insert any). However, the gem5 O3 CPU does not implement move elimination to elide identity moves (§IV-B3); ProtCC-CT/-CTS binaries would exhibit lower runtime overhead on hardware that does.

3) Protection-Tagged L1D Variants: We find that ProtISA's protection-tagged L1D (§IV-C2a) is critical to PROTEAN's performance, but tracking memory protection beyond the L1D has diminishing returns. When disabled, PROTEAN-Track-ARCH/-CT's overhead increases from 12.0%/46.3% to 32.2%/57.0% on the SPEC2017int benchmarks running on a P-core. When the protection-tagged L1D is replaced with a shadow memory that perfectly tracks memory protection, PROTEAN-Track-ARCH/-CT's drops to 4.5%/39.5%.

4) AccessDelay and AccessTrack: To approximate the performance of AccessDelay/AccessTrack (§VI-A) under ProtISA, we disable ProtTrack's access predictor (§VI-B2) and ProtDelay's selective wakeup optimizations (§VI-B1). Doing so adds a substantial +87.6%/+15.5% runtime overhead averaged across ProtCC-ARCH-/ProtCC-CT-compiled SPEC2017int benchmarks running on a P-core.

5) P-core vs. E-core: All evaluated defenses exhibit lower overhead on an E-core than on a P-core for SPEC2017. We attribute this to the E-core's shorter speculation windows due to its smaller ROB size (Tab. III).

6) CONTROL *Speculation Model:* PROTEAN-Track-ARCH/-CT average 8.2%/21.7% runtime overhead for SPEC2017int running on a P-core under the noncomprehensive CONTROL speculation model (§II-B2), whereas STT/SPT average 13.6%/18.4%. PROTEAN-CT only performs worse than SPT due to ProtCC-CT's instrumentation overhead (5.3%, §IX-A2).

7) Secure Baseline Bug Fixes: Our security fixes for STT/SPT/SPT-SB add +1.5%/+11.4%/+1.6% additional runtime overhead on SPEC2017int running on a P-core. Our SPT performance fix reduces the overhead to only +5.4%.

B. Single-Class Workloads (Tab. V)

PROTEAN-Track exhibits less than one twentieth the runtime overhead of STT on ARCH-Wasm, less than one half the overhead of SPT on both CTS-Crypto and CT-Crypto, and slightly outperforms SPT-SB on UNR-Crypto. PROTEAN-Delay outperforms all baseline defenses by smaller margins.

1) ARCH-Wasm: To understand PROTEAN's strong outperformance of STT on non-secret-accessing code, we identified and examined the top ten transmitters that STT cumulatively delayed the longest in the highest-weight simpoint for the benchmark on which it performed the worst (`milc`). All of them are loads dereferencing a pointer loaded from memory by the preceding instruction, like `mov ptr, [mem]; mov data, [ptr]`. Since STT unconditionally taints `ptr` until the first load retires, it effectively serializes all load-load dependencies. But, only 10% of those top ten dependencies access protected data in ProtISA's protection-tagged L1D. So, PROTEAN only stalls 10% of those load-load dependencies, allowing the other 90% to execute speculatively.

2) CTS-Crypto: PROTEAN-CTS outperforms SPT because ProtCC-CTS is able to unprotect all publicly-typed registers—i.e., registers that are fully or partially transmitted, through direct or transitive register dependencies—whereas SPT can only unprotect the strict subset of publicly-typed registers that are fully transmitted. Thus, SPT protects more data, so more protected data reaches transmitters, which SPT must therefore stall more often than PROTEAN.

3) CT-Crypto: PROTEAN-CT outperforms SPT because ProtCC-CT can detect and unprotect architecturally bound-to-leak data at compile time before it leaks (§V-A3), allowing dependent transmitters to execute speculatively earlier. In particular, since SPT cannot detect bound-to-leak data under ATCOMMIT (§II-B2), it must wait until the first transmitter to leak this data becomes non-speculative (i.e., reaches the head of the ROB) before it can unprotect the transmitted register.

4) UNR-Crypto: Notably, PROTEAN-UNR outperforms SPT-SB on PARSEC (§IX-A1) much more than UNR-Crypto. We attribute this to the fact that PARSEC contains comparatively more stack accesses than OpenSSL non-constant-time cryptography, so the latter sees less of a performance boost from ProtCC-UNR's unprotection of the stack pointer.

C. Multi-Class NGINX (Tab. V)

PROTEAN-Delay/-Track average less than one-third/one-fifth the runtime overhead of SPT-SB when securing the nginx HTTPS web server. By compiling different components with different ProtCC passes, PROTEAN is able to target its ProtSet and thus defense to the code currently executing. In contrast, SPT-SB—the only prior defense capable of securing nginx—protects all nginx code as if it is unrestricted.

X. RELATED WORK

1) Noncomprehensive Defenses: Many Spectre defenses are noncomprehensive, assuming a restricted set of channels or speculation types and thus still allowing secrets to transiently leak. Some defend against cache-based channels only [6], [7], [61], [66], [75], [99], [107], [112], [113], [126], [142]. MI6 [23] and Dolma [86] defend against cross-core and same-thread, but not cross-SMT-thread, leakage. Many software defenses target a single speculation type, like conditional [15], [26], [30], [52], [101], [102], [108], [122], [123], [131], [149], [152] or indirect [9], [53], [129] branch prediction; some target multiple but not all [43], [56], [59], [70], [97], [99], [103], [119], [134] (e.g., memory order speculation [111]).

2) Non-programmer-transparent Defenses: OISA [146], ProSpeCT [35], ConTExT [118], and SpectreGuard [42] rely on explicit source-level secrecy annotations to label memory as secret/public and prevent transient leaks of secret memory. Like ProtISA, these defenses expose programmable ProtSets. Unlike ProtISA, they cannot be automatically programmed by standard compiler analyses (§V-A), since they do not support dynamically protecting [35], [42], [118] or unprotecting [35], [42] written memory (requirement 1, §IV-A), or unprotecting read memory [35], [42], [118], [146] (requirement 2). SafeBet [46], Okapi [116], and Perspective [65] require programmers to modify memory layout, ensuring secrets are not placed in architecturally accessed cache lines or pages, since these can transiently leak unless explicitly reset.

3) Defense Optimizations: Prior work proposes security-preserving optimizations for layering on top of Spectre defenses (like PROTEAN). Rather than stalling transmitters with tainted inputs, SDO [147] and Doppelganger Loads [71] speculatively execute them along predicted hardware paths or with predicted addresses, respectively. InvarSpec [150], Clearing Shadows [128], and Levioso [51] enable earlier, safe execution through compile-time transformations [128] or by communicating static analyses to hardware [51], [150]. ReCon [5] marks memory words unprotected if they were previously leaked via load-load dependencies.

XI. CONCLUSION

We present the PROTEAN Spectre defense—the first that is comprehensive, programmer-transparent, and programmable. PROTEAN builds on the insight that performantly securing diverse real-world applications requires software-programmable, rather than hardware-defined, protections against transient leakage. ProtISA realizes this programmability via the `PROT` prefix, introducing an architectural protection set that standard compiler passes like ProtCC can automatically program and adaptations of standard hardware protection mechanisms like ProtDelay/ProtTrack can performantly enforce. With only one set of hardware changes, PROTEAN fully secures Spectre-vulnerable code with lower runtime overhead than prior comprehensive, programmer-transparent defenses. Future research on ProtISA programming, e.g., using more advanced analyses or user hints, stands to reduce PROTEAN's overhead further.

ACKNOWLEDGMENTS

We thank Baris Kasikci and the anonymous reviewers for their constructive comments and feedback. We also thank Christopher W. Fletcher for his careful review of our fixes to the evaluated secure baselines. This work was supported in part by the National Science Foundation (NSF), under awards 2153936 and 2236855 (CAREER); the Defense Advanced Research Projects Agency (DARPA) under contract W912CG-23-C-0025 and subcontract from Galois, Inc.; and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We gratefully acknowledge gifts from Apple, Intel, and Amazon.

REFERENCES

[1] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[2] Advanced Micro Devices, Inc., "Security analysis of amd predictive store forwarding," https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2021.

[3] Advanced Micro Devices, Inc., "Technical guidance for mitigating branch type confusion," https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion.pdf, AMD, Tech. Rep., 2022, accessed: April 18, 2024.

[4] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx," in *Network and Distributed System Security Symposium*, 2018.

[5] P. Aimoniotis, A. B. Kvalsvik, X. Chen, M. Själander, and S. Kaxiras, "Recon: Efficient detection, management, and use of non-speculative information leakage," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[6] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 592–606.

[7] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020.

[8] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[9] N. Amit, F. Jacobs, and M. Wei, "Jumpswitches: Restoring the performance of indirect branches in the era of spectre," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 285–300.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[11] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," *19th USENIX Security Symposium*, 2010.

[12] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, 2017.

[13] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164.

[14] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021, pp. 777–795.

[15] J. Baumann, R. Blanco, L. Ducruet, S. Harwig, and C. Hriţcu, "Fslh: Flexible mechanized speculative load hardening," in *2025 IEEE 38th Computer Security Foundations Symposium (CSF)*. IEEE, 2025, pp. 569–584.

[16] D. J. Bernstein, "Cache-timing attacks on aes," The University of Illinois at Chicago, Tech. Rep., 2005, https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[17] D. J. Bernstein, "The poly1305-aes message-authentication code," in *Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds., 2005.

[18] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., 2006.

[19] D. J. Bernstein, "djbsort," https://sorting.cr.yp.to, 2024, accessed: April 17, 2024.

[20] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[22] bitcoin-core, "ctaes," https://github.com/bitcoin-core/ctaes, 2020.

[23] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 42–56.

[24] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[25] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree at https://transient.fail/.

[26] C. Carruth, "Cryptographic software in a post-spectre world. talk at the real world crypto symposium." https://chandlerc.blog/talks/2020_post_spectre_crypto/post_spectre_crypto.html, 2020, accessed October 2022.

[27] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[28] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 666–680.

[29] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 69–76.

[30] R. Choudhary, A. Wang, Z. N. Zhao, A. Morrison, and C. W. Fletcher, "Declassiflow: A static analysis for modeling non-speculative knowledge to relax speculative execution security measures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2053–2067.

[31] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt)," https://github.com/FPSG-UIUC/SPT, 2022, accessed: 2025-10-21.

[32] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[33] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*, 2009.

[34] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS 2021-Network and Distributed Systems Security*, 2021.

[35] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "Prospect: Provably secure speculation for the constant-time policy," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[36] F. Denis, "libsodium," 2019, https://github.com/jedisct1/libsodium.

[37] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *Proc. VLDB Endow.*, 2019.

[38] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[39] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional encryption using intel sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[40] C. W. Fletcher and J. Yu, "Speculative taint tracking (stt)," https://github.com/cwfletcher/stt/, 2024, accessed: 2024-11-22.

[41] B. Fu, L. Tenenbaum, D. Adler, A. Klein, A. Gogia, A. R. Alameldeen, M. Guarnieri, M. Silberstein, O. Oleksenko, and G. Saileshwar, "Amulet: Automated design-time testing of secure speculation countermeasures," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 32–47.

[42] J. Fustos, F. Farshchi, and H. Yun, "Spectreguard: An efficient data-centric defense mechanism against spectre attacks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[43] L. Gerhorst, H. Herzog, P. Wägemann, M. Ott, R. Kapitza, and T. Hönig, "Verifence: Lightweight and precise spectre defenses for untrusted linux kernel extensions," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024, pp. 644–659.

[44] Google LLC. (2024) V8 javascript engine. Accessed: 2024-11-22. [Online]. Available: https://v8.dev

[45] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security'18*, 2018.

[46] C. Green, C. Nelson, M. Thottethodi, and T. Vijaykumar, "Safebet: Secure, simple, and fast speculative execution," *arXiv preprint arXiv:2306.07785*, 2023.

[47] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *International Conference on Information Security and Cryptology*. Springer, 2009, pp. 176–192.

[48] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy*, 2021.

[49] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games—bringing access-based cache attacks on AES to practice," *2011 IEEE Symposium on Security and Privacy (S&P)*, 2011.

[50] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[51] A. Hajiabadi, A. Agarwal, A. Diavastos, and T. E. Carlson, "Levioso: Efficient compiler-informed secure speculation," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[52] A. Hajiabadi and T. E. Carlson, "Cassandra: Efficient enforcement of sequential execution for cryptographic programs," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 78–91.

[53] L. Hetterich, M. Bauer, M. Schwarz, and C. Rossow, "Switchpoline: A software mitigation for spectre-btb and spectre-bhb on armv8," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 217–230.

[54] N. Homma, T. Aoki, and A. Satoh, "Electromagnetic information leakage for side-channel analysis of cryptographic modules," *2010 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2010.

[55] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "RTL2MμPATH: Multi-μpath synthesis with applications to hardware security verification," in *Proceedings of the Fifty-Seventh IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 57, 2024.

[56] Intel, "Analysis of Speculative Execution Side Channels," 2018, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[57] Intel, "Intel® core™ i9-12900ks processor," 2022, https://ark.intel.com/content/www/us/en/ark/products/225916/intel-core-i9-12900ks-processor-30m-cache-up-to-5-50-ghz.html.

[58] Intel Corporation, "Affected processors: Guidance for security issues on intel® processors," https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html, 2024.

[59] H. Jang and Y. Shin, "Microcfi: Microarchitecture-level control-flow restrictions for spectre mitigation," *IEEE Access*, vol. 11, pp. 138 699–138 711, 2023.

[60] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 216–225.

[61] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019.

[62] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "Flop: Breaking the apple m3 cpu via false load output predictions," in *USENIX Security*, 2025.

[63] J. Kim, D. Genkin, and Y. Yarom, "Slap: Data speculation attacks via load address prediction on apple silicon," in *S&P*, 2025.

[64] J. Kim, S. van Schaik, D. Genkin, and Y. Yarom, "ileakage: Browser-based timerless speculative execution attacks on apple devices," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2038–2052.

[65] T. H. Kim, D. Rudo, K. Zhao, Z. N. Zhao, and D. Skarlatos, "Perspective: A principled framework for pliable and secure speculation in operating systems," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 739–755.

[66] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[67] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2399–2416.

[68] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2019, pp. 1–19.

[69] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*, 1999.

[70] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 39–53.

[71] A. B. Kvalsvik, P. Aimoniotis, S. Kaxiras, and M. Själander, "Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[72] C. Lam. (2021) Gracemont: Revenge of the atom cores. Chips and Cheese. [Online]. Available: https://chipsandcheese.com/p/gracemont-revenge-of-the-atom-cores

[73] C. Lam. (2021) Popping the hood on golden cove. Chips and Cheese. [Online]. Available: https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[74] 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.* IEEE, 2004, pp. 75–86.

[75] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[76] Linux Kernel Documentation, "Kernel samepage merging (ksm)," https://docs.kernel.org/admin-guide/mm/ksm.html, 2025.

[77] Linux Kernel Documentation, "zram," https://docs.kernel.org/admin-guide/blockdev/zram.html, 2025.

[78] Linux Kernel Documentation, "zswap," https://www.kernel.org/doc/html/latest/admin-guide/mm/zswap.html, 2025.

[79] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *CoRR*, vol. abs/1801.01207, 2018, https://arxiv.org/abs/1801.01207.

[80] LLVM Project, ""libc++" C++ Standard Library," Software, 2023, https://libcxx.llvm.org.

[81] LLVM Project, "The LLVM C Library," Software, 2023, https://libc.llvm.org.

[82] LLVM Project, "The LLVM Target-Independent Code Generator," 2023, https://www.llvm.org/docs/CodeGenerator.html.

[83] LLVM Project, "Clang: a c language family frontend for llvm," https://clang.llvm.org/, 2024, accessed: April 17, 2024.

[84] LLVM Project, "The flang compiler," https://flang.llvm.org/, 2024, accessed: April 17, 2024.

[85] LLVM Project, *llvm-stress — generate random .ll files*, LLVM Project, 2025, last updated October 12, 2025. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-stress.html

[86] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability." in *USENIX Security Symposium*, 2021, pp. 1397–1414.

[87] S. Mangard, "A simple power-analysis (SPA) attack on implementations of the aes key expansion," *5th International Conference on Information Security and Cryptology (ICISC)*, 2003.

[88] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, “Spectre is here to stay: An analysis of side-channels and speculative execution,” 2019. [Online]. Available: https://arxiv.org/abs/1902.05178

[89] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, “Oblix: An efficient oblivious search index,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[90] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations,” *International Journal of Parallel Programming*, vol. 47, no. 4, 2019.

[91] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, P. C. van Dyken, C. H. Tomkins-Tinch, V. Sochat, J. Forster, F. G. Vieira, C. Meesters *et al.*, “Sustainable data analysis with snakemake,” *F1000Research*, vol. 10, p. 33, 2025.

[92] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, “The program counter security model: Automatic detection and removal of control-flow side channel attacks,” in *Information Security and Cryptology - ICISC 2005*, D. H. Won and S. Kim, Eds., 2006.

[93] N. Mosier, “SPT: Fix pending squash bug,” GitHub Pull Request #4 in repository `FPSG-UIUC/SPT`, December 2025, merged into master. [Online]. Available: https://github.com/FPSG-UIUC/SPT/pull/4

[94] N. Mosier, “SPT: Taint all registers at rename,” GitHub Pull Request #6 in repository `FPSG-UIUC/SPT`, December 2025, merged into master. [Online]. Available: https://github.com/FPSG-UIUC/SPT/pull/6

[95] N. Mosier, “SPT: Untaint upper 32 bits of 32-bit destination registers,” GitHub Pull Request #5 in repository `FPSG-UIUC/SPT`, December 2025, merged into master. [Online]. Available: https://github.com/FPSG-UIUC/SPT/pull/5

[96] N. Mosier, “stt: Fix pending squash bug,” GitHub Pull Request #10 in repository `cwfletcher/stt`, December 2025, merged into master. [Online]. Available: https://github.com/cwfletcher/stt/pull/10

[97] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, “Serberus: Protecting cryptographic code from spectres at compile-time,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4200–4219.

[98] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, “Fast, accurate, and novel performance evaluations with pincpu,” Presentation at the gem5 Workshop, 52nd Annual International Symposium on Computer Architecture (ISCA 2025), June 2025, video: https://www.youtube.com/watch?v=v8A-HU0ElZ0. Slides: https://www.gem5.org/_pages/static/events/isca_2025/01-pincpu-gem5-2025-06-21-export.pdf.

[99] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, “Swivel: Hardening WebAssembly against spectre,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[100] NGINX, “nginx,” https://nginx.org, 2025, accessed: 2025-08-01.

[101] E. J. Ojogbo, M. Thottethodi, and T. Vijaykumar, “Secure automatic bounds checking: prevention is simpler than cure,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

[102] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, “You shall not bypass: Employing data dependencies to prevent bounds check bypass,” *CoRR*, vol. abs/1805.08506, 2018. [Online]. Available: http://arxiv.org/abs/1805.08506

[103] S. A. Olmos, G. Barthe, C. Chuengsatiansup, B. Grégoire, V. Laporte, T. Oliveira, P. Schwabe, Y. Yarom, and Z. Zhang, “Protecting cryptographic code against spectre-rsb (and, in fact, all known spectre variants),” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[104] “OpenSSL: Cryptography and SSL/TLS toolkit,” 2021, https://www.openssl.org/.

[105] OpenSSL Technical Committee, “Spectre and meltdown attacks against openssl,” *OpenSSL Blog*, 2022. [Online]. Available: https://www.openssl.org/blog/blog/2022/05/13/spectre-meltdown/

[106] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of AES,” *2006 The Cryptographers’ Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[107] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, “Hidfix: Efficient mitigation of cache-based spectre attacks through hidden rollbacks,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[108] M. Patrignani and M. Guarnieri, “Exorcising spectres with secure compilers,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 445–461.

[109] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks,” in *USENIX Security’16*, 2016.

[110] T. Pornin, “Bearssl,” `https://bearssl.org`, 2018, accessed: April 17, 2024.

[111] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1451–1468.

[112] G. Saileshwar and M. K. Qureshi, “Cleanupspec: An” undo” approach to safe speculation,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[113] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Efficient invisible speculative execution through selective delay and value prediction,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 723–735.

[114] S. Sasy, S. Gorbunov, and C. W. Fletcher, “Zerotrace: Oblivious memory primitives from intel sgx,” *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 549, 2018.

[115] A. Sayakkara, N.-A. Le-Khac, and M. Scanlon, “A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics,” *Digital Investigation*, vol. 29, pp. 43–54, 2019.

[116] P. Schmitz, T. Jauch, A. Wezel, M. R. Fadiheh, T. Tiemann, J. Heller, T. Eisenbarth, D. Stoffel, and W. Kunz, “Okapi: Efficiently safeguarding speculative data accesses in sandboxed environments,” in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, 2025, pp. 1203–1218.

[117] J. Schulist, D. Borkmann, and A. Starovoitov, “Linux Socket Filtering aka Berkeley Packet Filter (BPF),” https://www.kernel.org/doc/Documentation/networking/filter.txt, 2018.

[118] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, “Context: A generic approach for mitigating spectre,” in *Network and Distributed System Security Symposium*, 2020.

[119] M. Schwarzl, C. Canella, D. Gruss, and M. Schwarz, “Specfuscator: Evaluating branch removal as a spectre mitigation,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 293–310.

[120] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, “Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[121] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002. [Online]. Available: https://doi.org/10.1145/605432.605403

[122] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O’Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, “Spectre declassified: Reading from the right place at the wrong time,” Cryptology ePrint Archive, Paper 2022/426, 2022, https://eprint.iacr.org/2022/426. [Online]. Available: https://eprint.iacr.org/2022/426

[123] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean, “Typing high-speed cryptography against spectre v1,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1592–1609.

[124] J. Software, “siege,” https://github.com/JoeDog/siege, 2025, accessed: 2025-08-01.

[125] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[126] M. Taram, A. Venkat, and D. Tullsen, “Context-sensitive fencing: Securing speculative execution via microcode customization,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[127] S. Tople and P. Saxena, “On the trade-offs in oblivious execution techniques,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds., 2017.

[128] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean, “Clearing the shadows: Recovering lost performance for invisible speculative execution through hw/sw co-design,” in *Proceedings*

of the ACM International Conference on Parallel Architectures and Compilation Techniques, 2020, pp. 241–254.

[129] P. Turner, "Retpoline: a software construct for preventing branch-target-injection." https://support.google.com/faqs/answer/7625886, 2018, accessed October 2022.

[130] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008.

[131] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proc. ACM Program. Lang.*, 2021.

[132] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022.

[133] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. 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)*. IEEE, 2021, pp. 347–360.

[134] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2504–2519, 2019.

[135] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS '17*, 2017.

[136] WASI-SDK Contributors, "Wasi sdk: Wasi-enabled webassembly c/c++ toolchain," https://github.com/WebAssembly/wasi-sdk, 2024, accessed: 2024-06-24.

[137] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[138] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[139] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[140] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[141] M. Yan, I. Cai, and J. Yu, "Invisispec-1.0," https://github.com/mjyan0720/InvisiSpec-1.0, 2018, accessed: 2025-12-10.

[142] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[143] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE S&P*, 2019.

[144] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," *23rd USENIX Security Symposium*, 2014.

[145] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *IACR'16*, 2016.

[146] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[147] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[148] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[149] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate slh: Taking speculative load hardening to the next level," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7125–7142.

[150] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas, "Speculation invariance (invarspec): Faster safe execution through program analysis," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1138–1152.

[151] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[152] Y. Zhu, W. Huang, and Y. Xiong, "Lightslh: Provable and low-overhead spectre v1 mitigation through targeted instruction hardening," *arXiv preprint arXiv:2408.16220*, 2024.

Appendix A
Artifact Appendix

A. Abstract

Our artifact is a Docker image containing the full source code and precompiled binaries for our gem5 implementation of PROTEAN's ProtISA and ProtDelay/ProtTrack as well as our LLVM-based implementation of PROTEAN's ProtCC. This artifact also provides scripts to run a subset of our PROTEAN performance and security experiments for the purpose of artifact evaluation as well as all experiments necessary to reproduce all results tables and figures in the paper for the interested reader.

B. Artifact check-list (meta-information)

- **Program:** User must provide the ISO image for:
  - SPEC CPU2006 (`cpu2006-1.2.iso`, *reference* inputs, 3.8 GiB)

  Included in the artifact:
  - nginx (v1.29.0, 0.2 GiB)
  - OpenSSL (v3.5.1, 2.6GiB)
  - HACL* (v0.4.5, 3.0 GiB)
  - BearSSL (commit `3d9be2`, 0.2 GiB)
- **Binary:** Prebuilt binaries in Docker image.
- **Run-time environment:** Docker (v20+) on Linux host kernel (v3.17+).
- **Metrics:** Simulated runtime, normalized to unsafe baseline (performance evaluation) and number of true-positive security contract violations (security evaluation).
- **Output:** PDF tables `table-v.pdf` (performance results) and `table-ii.pdf` (security results). Expected performance and security results can be found in Tab. V and Tab. II, respectively.
- **Experiments:** Run Python scripts within Docker container.
- **How much disk space required (approximately)?** 45 GiB (Docker image: 30 GiB; experiments: 15 GiB).
- **How much time is needed to prepare workflow (approximately)?** 15 minutes, for downloading and loading Docker image.
- **How much time is needed to complete experiments (approximately)?** 22 hours on a single core, or 2 hours on an unlimited number of cores.
- **Publicly available?** Yes.
- **Code licenses:** BSD 3-Clause.
- **Workflow automation framework used:** Snakemake [91].
- **Archived:** https://doi.org/10.5281/zenodo.17857895

C. Description

1) How to access: The artifact Docker image can be found at https://hub.docker.com/r/nmosier/protean. PROTEAN source code and instructions for building the docker image from source can be found at https://github.com/StanfordPLArchSec/protean.

2) Hardware dependencies: The artifact requires a x86-64 host machine.

3) Software dependencies: Docker is required to launch the artifact Docker image, in which all experiments are run. Users must provide a copy of the SPEC CPU2006 benchmarks in the form of an ISO image (e.g., `cpu2006-1.2.iso`). All other software dependencies are preinstalled in the provided Docker image.

D. Installation

To download, load, and start up the artifact Docker image, simply run the following shell commands. Commands executed on the host are prefixed with `$`; those executed within the container are prefixed with `#`.

```
$ docker pull nmosier/protean:latest
$ docker run --name protean-container -it nmosier/
    protean:latest /bin/bash
$ docker cp /host/path/to/cpu2006.iso protean-
    container:/protean/cpu2006.iso
# ./extract-spec-cpu2006-iso.sh
```

Note that you must execute the `docker cp` command in a different terminal while the docker container is running. These commands are also provided in https://github.com/StanfordPLArchSec/protean, where they can be easily copied and pasted.

E. Experiment workflow

The Docker image contains Python scripts in the home directory (`/protean`) to run the necessary experiments to reproduce partial or full copies of all results tables and figures in the paper. Each script (e.g., `table-v.py`) is named after the paper table/figure it reproduces (e.g., Tab. V) and saves the results table/figure as a PDF (e.g., `table-v.pdf`) that can be compared against the version presented in the paper.

F. Evaluation and expected results

Since reproducing all results tables and figures (§A-G1) takes approximately 500 days of total host runtime, we provide instructions for reproducing a scaled-back subset of performance (§A-F1) and security (§A-F2) results.

1) Performance results: To reproduce a representative subset of our key performance results in Tab. V, run the following shell command:

```
# ./table-v.py --bench={lbm,hacl.poly1305,bearssl,
    ossl.bnexp,nginx.c1r1}
```

Description: This simulates the unsafe baseline, PROTEAN (both ProtDelay and ProtTrack), and the appropriate secure baseline on the one benchmark for each class-specific suite in Tab. V that exhibits the shortest host runtime (specified with `--bench`).

Output: The script outputs a PDF, `table-v.pdf`, containing the normalized simulated runtimes of PROTEAN and the secure baseline formatted into a table resembling Tab. V. The results of omitted benchmarks are marked with `-`.

Comparing to paper results: Normalized runtimes reported in the artifact may vary slightly (less than a ± 0.07 difference in normalized runtime, i.e., $\pm 7\%$ runtime overhead) from normalized runtimes presented in the paper's Tab. V.

2) Security results: To reproduce our qualitative empirical security results in Tab. II, run the following shell command:

```
# ./table-ii.py --instrumentation=rand
```

Description: This tests the unsafe baseline and PROTEAN (both ProtDelay and ProtTrack) against the `UNPROT-SEQ` security contract on randomly `PROT`-prefixed ProtISA binaries with the following scaled-down parameters compared to §VII-B2. For both the adversary models we evaluate (original AMuLeT's default adversary model and AMuLeT*'s timing-based adversary model), we run 5 AMuLeT* instances (rather than 100) per defense configuration, 50 programs per instance, and 50 (resp. 3) inputs per program for AMuLeT's default (resp. AMuLeT*'s timing-based) adversary model.

Output: The script outputs a PDF, `table-ii.pdf`, containing the number of true-positive and false-positive violations detected for each of the three defense configurations (unsafe baseline, PROTEAN with ProtTrack, and PROTEAN with ProtDelay). The results of omitted security experiments are marked with `-`.

Comparing with paper results: The generated table should report (i) *at least one* true-positive violation for the unsafe baseline, demonstrating that AMuLeT* successfully detects Spectre vulnerabilities, and (ii) *zero true-positive* violations for PROTEAN with ProtTrack and ProtDelay. The number of false-positive violations can be ignored, since they do not indicate a true security issue and their count may vary from those reported in Tab. II due to reduced testing parameters and randomness between runs.

G. Experiment customization

Beyond the main artifact evaluation, we provide additional scripts for running all experiments necessary to generate complete paper versions of all results tables and figures for the interested reader (§A-G1). These scripts can also be configured via command-line options to only include the results of specific experiments (§A-G2). Our evaluation infrastructure can be extended to run new benchmarks (§A-G3), benchmark new hardware-software codesigns (§A-G4), and validate the security of new hardware-software Spectre defenses (§A-G5).

1) Generating complete results tables and figures: We provide scripts that run experiments for and generate PDFs of complete results tables and figures in the paper:

```
# ./table-i.py  --all # Generates Tab. I.
# ./figure-4.py --all # Generates Fig. 5.
# ./table-ii.py --all # Generates Tab. II.
# ./figure-6.py --all # Generates Fig. 6.
# ./table-iv.py --all # Generates Tab. IV.
# ./table-v.py  --all # Generates Tab. V.
```

Note that `table-i.py`, `figure-4.py`, `figure-6.py`, and `table-iv.py` require the user to provide and extract an ISO image of the SPEC CPU2017 benchmarks using the following commands:

```
$ docker cp /host/path/to/cpu2017.iso protean-
    container:/protean/cpu2017.iso
# ./extract-spec-cpu2017-iso.sh
```

All scripts above accept an optional `--expected` flag, which instructs the script to generate the figures using canonical paper results (saved in `./bench/paper` and `./amulet/paper`), rather than artifact-reproduced results.

2) Generating partial results tables and figures: All scripts in §A-G1 support selecting a specific subset of program classes (with `--program-class` for `table-iv.py`) or benchmarks (with `--bench` for all others) to run, as an alternative to running all experiments with `--all`. For example, the following command generates a partial version of Fig. 6 showing only results for the `perlbench` SPEC CPU2017 and the `blackscholes` PARSEC benchmarks:

```
# ./figure-6.py --bench={perlbench.s,blackscholes.p}
```

These flags are documented in each respective script's help dialog (printed when passed the option `--help`).

3) Running new benchmarks: To add a new benchmark to our performance evaluation, please see `bench/example/README.md`, which contains instructions for adding a toy example benchmark.

4) Evaluating the performance of other hardware-software codesigns: To facilitate future research into hardware-software codesign, we document how researchers can easily adapt our performance evaluation infrastructure to evaluate their own hardware-software prototypes in `bench/HW-SW.md`.

5) Evaluating the security of other hardware-software Spectre defenses: We detail in `amulet/HW-SW.md` how security researchers can easily adapt/extend our security evaluation infrastructure to test their own hardware-software Spectre defense prototypes.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae