Serberus: Protecting Cryptographic Code from Spectres at Compile-Time

Authors' version; to appear in the Proceedings of the IEEE Symposium on Security and Privacy (S&P) 2024

Nicholas Mosier
nmosier@stanford.edu
Stanford University
Stanford, California, USA

Hamed Nemati
hamed.nemati@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

John C. Mitchell
jcm@stanford.edu
Stanford University
Stanford, California, USA

Caroline Trippel
trippel@stanford.edu
Stanford University
Stanford, California, USA

***Abstract*—We present SERBERUS, *the first comprehensive mitigation* for hardening constant-time (CT) code against Spectre attacks (involving the PHT, BTB, RSB, STL, and/or PSF speculation primitives) on *existing hardware*. SERBERUS is based on three insights. First, some hardware control-flow integrity (CFI) protections restrict transient control-flow to the extent that it may be comprehensively considered by software analyses. Second, conformance to the accepted CT code discipline permits two code patterns that are unsafe in the post-Spectre era. Third, once these code patterns are addressed, all Spectre leakage of secrets in CT programs can be attributed to one of four classes of *taint primitives*—instructions that can transiently assign a secret value to a publicly-typed register. We evaluate SERBERUS on cryptographic primitives in the OPENSSL, LIBSODIUM, and HACL* libraries. SERBERUS introduces 21.3% runtime overhead on average, compared to 24.9% for the next closest state-of-the-art software mitigation, which is less secure.**

1. Introduction

The *constant-time (CT) programming* approach [1]–[14] was designed to support the safe execution of secret-processing programs, like cryptographic code [15]–[17], in the face of hardware side-channel attacks. Concretely, CT programming requires that only *safe* instructions, which create operand-independent hardware resource usage, process secrets. *Unsafe* instructions, i.e., information *transmitters* (or simply *transmitters*) [18], typically include control-flow, memory, and variable-time (e.g., floating-point [19] or integer division [12]) instructions.

Unfortunately, common hardware optimizations enable *transient execution*[1] to *steer* secrets towards the operands of (transient) transmitters, circumventing CT protections. In *Spectre attacks* specifically (our focus, §2.2), transient execution results from control- or data-flow *mispredictions* at runtime [20]. On modern processors, there are five well-documented sources of such (mis)predictions, called *speculation primitives* [21]: conditional branch prediction (PHT) [22], indirect branch prediction (BTB) [22], return address prediction (RSB) [23], store-to-load forwarding prediction (STL) [24], and predictive store forwarding (PSF) [25]–[27].[2]

Mitigation	Leakage	Proof	PHT	BTB	RSB	STL	PSF
INTEL-LFENCE [28]	-	-	⌫	-	-	-	-
LLVM-SLH [29]	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	-	-	-	-
RETPOLINE [30]	-	-	-	⌫	↑	-	-
IPREDD [31]	-	-	-	⌫	-	-	-
SSBD [32]	-	-	-	-	-	⌫	⌫
PSFD [33]	-	-	-	-	-	-	⌫
F+RETP+SSBD	-	-	⌫	⌫	-	⌫	⌫
S+RETP+SSBD	$[\![\cdot]\!]_{\text{arch}}$	✗	◐	⌫	-	⌫	⌫
BLADE [34]	$[\![\cdot]\!]_{\text{ct}}$	✓	●	-	-	-	-
ULTIMATESLH [35]	$[\![\cdot]\!]_{\text{ct}}$	✗	●	-	-	-	-
SWIVEL-CET [36]	$[\![\cdot]\!]_{\text{mem}}$	✗	●	●	●	⌫	⌫
SERBERUS (ours)	$[\![\cdot]\!]_{\text{ct}}$	✓	●	●	●	●	⌫

TABLE 1: SERBERUS versus mitigations for existing hardware. *Leakage:* SERBERUS targets the CT leakage model $[\![\cdot]\!]_{\text{ct}}$; CT code is insecure under $[\![\cdot]\!]_{\text{mem}}$ and $[\![\cdot]\!]_{\text{arch}}$ [25]. *Speculation primitives:* ● complete mitigation without disabling speculation; ⌫ disables speculation primitive (⌫ for implicitly disabled); ◐ incomplete mitigation; ↑ creates opportunities for speculation primitive to introduce transient execution; - no mitigation. SERBERUS is the only defense for the CT leakage model to mitigate leakage due to *all* speculation primitives.

Hardening CT programs against *all* Spectre attacks involving *any combination* of the aforementioned speculation primitives is hard. Suitable hardware mitigations have been proposed [37]–[40], but they require complex design changes that limit their adoption. Several mitigations target existing hardware (Tab. 1). However, none, nor any combination, is suitable for securing CT code.

This Paper. We present SERBERUS,[3] *the first comprehensive mitigation* for hardening CT code against Spectre attacks involving any combination of the PHT, BTB, RSB, STL, and PSF speculation primitives on *existing hardware*. It mitigates the first four primitives in software and disables only the last primitive (PSF) in hardware due to a clear

1. Transient execution refers to the execution of instructions that are never architecturally committed [20].

2. Abbreviations are borrowed from recent work which surveys Spectre attacks and software defenses [20], [25].

3. SERBERUS is named after Cerberus, a three-headed dog (representing its three mitigation passes) of Greek mythology with a serpentine tail (our hardware model ASP) guarding the gates of Hades to prevent the dead (transient executions) from escaping (exfiltrating secrets) to the overworld (via transmitters).

performance advantage. An implementation of SERBERUS is readily deployable in our compiler LLSCT,[4] which produces more performant (on average) and more secure binaries for cryptographic routines than state-of-the-art mitigations. SERBERUS is based on three key insights.

Insight 1: Hardware model. Mitigating Spectre attacks in software is challenged by (i) the impracticality of reasoning about unconstrained transient control-flow [41] and (ii) the overhead of managing unconstrained transient data-flow [42]. Like prior work [36], we observe that some lightweight (negligible overhead) hardware *control-flow integrity (CFI)* protections, like Intel CET [43], constrain transient control-flow to the extent that it may be comprehensively considered by software analyses. Thus, SERBERUS requires that such CFI protections are enabled in the target hardware. Moreover, we show *for the first time* that while Spectre-STL[5] can be efficiently mitigated in software, Spectre-PSF cannot. Thus, SERBERUS requires the PSFD speculation control, available on Intel [33] and AMD [44] processors, to disable PSF.

Insight 2: Programming contract. We focus on hardening CT programs against Spectre attacks [34], [38], [42], since they already avoid leaking secrets during *sequential* (i.e., non-transient) execution. However, we observe that CT programs, as defined previously [45], may contain two code patterns that are unsafe in the post-Spectre era and limit mitigated program performance: (i) transmitters with secret operands that never execute sequentially (e.g., "`if (0) leak(secret)`"), and (ii) procedure calls or returns with secret arguments, which can often leak via BTB or RSB misprediction. Thus, we introduce a strengthening of CT programming, called *static constant-time (CTS)*, which requires that (i) all program variables have a static security type, and (ii) all call and return arguments are public. SERBERUS requires a CTS program as input.

Insight 3: Taint primitives. We find that *all* Spectre leakage in CTS programs can be attributed to *taint primitives*, or instructions that transiently assign a secret value to a publicly-typed register. We prove that there are exactly *four* classes taint primitives (Fig. 1) for our hardware model (§2.3.2), which assumes (i) Intel's CET protections, (ii) Intel's RRSBAD and PSFD speculation controls [46], (iii) Intel's DOIT Mode [47], and (iv) the PHT, BTB, RSB, and STL speculation primitives. This result motivates SERBERUS's design and its *correctness proof*: it consists of three compiler passes, each of which mitigates Spectre leakage due to one or more classes of taint primitives.

Contributions. Our contributions are as follows:

- **ASP operational semantics (§3):** We define an operational semantics, called ASP, that encodes all (sequential and transient) control- and data-flow that a program may exhibit when it runs on a microarchitecture satisfying our hardware model (§2.3.2). ASP avoids explicitly modeling the hardware structures that give rise to speculation primitives, capturing a wider range of implementations than prior work [42], [48].
- **Static Constant-Time programming (§4):** We propose CTS programming, a strengthening of CT for the post-Spectre era. CTS forbids two legal CT code patterns (§4.2) that preclude efficient Spectre mitigation in software.
- **Taint primitives (§4):** We define and characterize *taint primitives*, instructions that cause a security type violation (§4.4.2) when transiently executed. Using ASP, we prove that taint primitives are *necessary* for Spectre leakage of secrets in CTS programs and that CTS programs exhibit exactly four classes of taint primitives.
- **SERBERUS mitigation (§5):** We propose SERBERUS, the *first comprehensive mitigation* for existing hardware that hardens CT code (satisfying CTS) against Spectre attacks that exploit PHT, BTB, RSB, STL, and/or PSF. SERBERUS offers the *first software mitigation for STL* that does not simply disable it (Tab. 1); only PSF is the only speculation primitive it disables. Using ASP, we prove that SERBERUS mitigates all Spectre leakage of secrets in CTS programs.
- **LLSCT compiler (§6):** We build LLSCT, a custom fork of LLVM 14.0.4 that implements SERBERUS.
- **Case study (§6–§7):** We evaluate LLSCT alongside two compositions of state-of-the-art mitigations [28]–[30], [32], F+RETP+SSBD and S+RETP+SSBD, on cryptographic primitives in the OpenSSL, Libsodium, and HACL* libraries. LLSCT produces more performant code than the state-of-the-art on most benchmarks, introducing *less than 8% average overhead* on large-buffer benchmarks.

4. LLSCT is open source and available at https://github.com/nmosier/llsct.

5. Spectre-STL denotes Spectre leakage enabled by the STL speculation primitive.

2. Background

2.1. Hardware Side-Channel Attacks

Hardware side-channel attacks involve a victim program (the *sender*) running on vulnerable hardware that leaks secrets to an attacker (the *receiver*). Such hardware contains *unsafe* instructions, or *transmitters* [18], whose execution creates operand-dependent hardware resource usage. A receiver infers the value(s) of a transmitter's *sensitive* operand(s) by measuring hardware resource usage through various means, such as execution time [49], [50] and more [19], [51]–[71].

Constant-time (CT) programming is the prevailing approach for countering hardware side-channel leakage in software [1]–[14], [72]. In short, CT programs do not supply secrets to sensitive transmitter operands in *any* sequential execution. CT has been widely adopted in the context of cryptographic code [15]–[17], [73], which must process (and not leak) secrets. Thus, a variety of tools and techniques have been proposed to support CT code generation [13], [14], [72], [74] and verification [45], [75]–[78]. Prior work surveys many such approaches [79].

2.2. Spectre Attacks

Transient execution attacks circumvent CT protections by steering secrets towards the sensitive operands of (transient) transmitters [22], [80]. These attacks leverage two high-level mechanisms for creating transient execution at runtime on modern processors: (i) *faulting* instructions, and (ii) control- or data-flow *mispredictions*. These mechanisms give rise to *Meltdown* and *Spectre* attacks, respectively [20].

In general, Meltdown attacks (e.g., Meltdown [80], Foreshadow [81], LVI [82], MDS [83]–[85]) can be efficiently mitigated through microcode updates or modest hardware changes [86]. In contrast, despite a plethora of research proposals for mitigating Spectre attacks in hardware [18], [37], [39], [40], [87]–[98], extreme complexity limits their adoption. Thus, we focus exclusively on mitigating *Spectre* attacks on *existing hardware* designs.

2.2.1. Speculation Primitives. Spectre attacks [22]–[24], [26], [42] are characterized according to distinct sources of control- and data-flow (mis)prediction on modern processors, called *speculation primitives* [20], [21]. Predictions introduce *speculative execution*, which may turn out to be *sequential* (when predictions are correct) or *transient* (when incorrect).

Control-Flow Prediction. To exploit instruction-level parallelism (ILP) in sequential applications, modern processors resolve a variety of program dependencies at runtime. Among these are *control dependencies* which arise due to control-flow instructions (e.g., conditional branches, indirect branches) whose *condition* and/or *target* decide the program counter (PC) of the next instruction to fetch.

To avoid frontend stalls, processors dedicate a significant amount of circuitry to control-flow prediction, giving rise to three notable speculation primitives. PHT (responsible for Spectre v1 [22] and v1.1 [99]) denotes conditional branch prediction, which diverts speculative execution following a *conditional branch* towards one of two control-flow paths (i.e., the taken or not taken path). BTB (responsible for Spectre v2 [22]) denotes indirect branch prediction, which diverts speculative execution following an *indirect branch* towards one of many control-flow paths. RSB (responsible for SpectreRSB [23]) denotes return address prediction, which operates similarly to BTB except that it diverts speculative execution following a *return*. In general, unconstrained indirect branch and return address prediction may direct speculative control-flow to *any* program instruction or even to the *middle* of an instruction [100].

Data-Flow Prediction. *Data-flow dependencies* through memory present another barrier to exploiting ILP. Specifically, loads block the execution of younger dependent instructions until they have retrieved their data. Two notable speculation primitives result from data-flow prediction mechanisms designed to accelerate the execution of loads. STL (responsible for Spectre v4 [24]) denotes store-to-load forwarding prediction, whereby a load may forward data from an older same-address store before all prior stores have resolved their addresses. That is, a *load* may speculatively read from any same-address *uncommitted* store or the most recent same-address *committed* store. PSF (responsible for Spectre PSF [26], [27], [42]) denotes predictive store forwarding, whereby a load may forward data from an older store before the load or store has resolved its address. With PSF, a *load* may speculatively read from any prior *uncommitted* store.

2.2.2. Software Mitigations for Spectre Attacks. Tab. 1 summarizes state-of-the-art software mitigations[6] for Spectre attacks, which we characterize below.

Leakage Models. One distinction among software Spectre mitigations is the *leakage model* they target (§3.2). A mitigation's leakage model captures what an attacker can observe when a victim program runs on hardware of interest. The most prevalent leakage models in the literature [25] are the CT leakage model ($[\![\cdot]\!]_{\mathrm{ct}}$) [26], [34], [41], [42], [101]–[104] and the sandbox isolation leakage model ($[\![\cdot]\!]_{\mathrm{arch}}$) [104], [105]. We adopt the CT leakage model in this paper due to our focus on mitigating Spectre leakage in cryptographic code [25], [104]. The CT leakage model exposes the control-flow and sequence of accessed memory addresses in a program's execution as *observations* to an attacker. The sandbox isolation model additionally exposes all values loaded from memory, which is unsuitable for modeling CT programs which need to access secrets. A less common leakage model, $[\![\cdot]\!]_{\mathrm{mem}}$, models a receiver that can only observe accessed memory addresses [104].

Blocking Speculation Primitives. The Spectre mitigations in Tab. 1 can be further classified as coarse- or fine-grained. Coarse-grained mitigations ⌫ disable culprit speculation primitive(s). We explain several examples below.

One Spectre-PHT mitigation proposed by Intel (INTEL-LFENCE) inserts an LFENCE after every conditional branch [28]. An LFENCE is a *serializing instruction*, which guarantees that any younger instructions will not be executed (even transiently) until all older ones commit. This mitigation is complete under $[\![\cdot]\!]_{\mathrm{ct}}$, but incurs a huge ($\approx 440\%$) overhead for typical software [106]. A higher-performance Spectre-PHT mitigation, *speculative load hardening (SLH)*, masks the addresses or return values of loads inside in a conditional branch with the branch predicate [29]. LLVM offers the only well-established SLH implementation (LLVM-SLH). However, it targets $[\![\cdot]\!]_{\mathrm{arch}}$ and thus is incomplete for CT code [103], [107]. UltimateSLH [35] extends LLVM's SLH implementation to the $[\![\cdot]\!]_{\mathrm{ct}}$ leakage model, with a performance cost.

Spectre-BTB can be mitigated with RETPOLINE [30] or Intel's IPREDD (i.e., IPRED_DIS_U) speculation control [31]. RETPOLINE replaces all indirect branches with returns that direct mispredictions to a "safe" target.[7] Setting IPREDD disables indirect branch prediction.

No complete nor deployable mitigations of Spectre-RSB exist under $[\![\cdot]\!]_{\mathrm{ct}}$. Intel has proposed an (incomplete)

6. "Software mitigations" are those that are deployable on *existing hardware*.

7. We assume the victim process sets the RRSBAD speculation control (§2.3.2), which prevents attacks from circumventing RETPOLINE [108].

technique called *RSB stuffing* to "reduce the likelihood of an [RSB] underflow from occurring" [109].

Spectre-STL and Spectre-PSF can be mitigated with SSBD and PSFD speculation controls [46], respectively, offered by Intel [46] and AMD [44].

Preventing Secret-Dependent Transmitters. A couple fine-grained Spectre mitigations have also appeared in the literature. These approaches ● explicitly prevent secrets from being supplied to transmitters during transient execution, while leaving speculation primitives enabled.

Blade [34] is a complete Spectre-PHT mitigation for CT WebAssembly under $[\![\cdot]\!]_{\mathrm{ct}}$ that frames LFENCE insertion as a min-cut data-flow problem. Swivel-CET [36] uses a control-flow tracking technique called *register interlocking* to mitigate Spectre attacks involving PHT, BTB, and/or RSB in WebAssembly programs; however, it targets $[\![\cdot]\!]_{\mathrm{mem}}$ and thus is insecure for CT code.

Layering Spectre Mitigations. State-of-the-art mitigations SLH, RETPOLINE, and SSBD incur modest overhead when deployed in isolation to mitigate Spectre attacks involving a *single* speculation primitive. Since there are no comprehensive mitigations for Spectre under $[\![\cdot]\!]_{\mathrm{ct}}$, §6-7 compare SERBERUS's performance to composite baselines, F+RETP+SSBD and S+RETP+SSBD, which combine INTEL-LFENCE (F) and LLVM-SLH (S) with RETPOLINE and SSBD.

2.3. Threat Model

2.3.1. Software Model. We *provably* harden trusted CTS (a slight strengthening of CT, §4.3) victim code against Spectre attacks that arise on processors satisfying the constraints of our hardware model (§2.3.2). We assume a powerful attacker that can directly observe the sensitive operands of transmitters executed (sequentially or transiently) by the victim and fully control speculation within the victim process.

2.3.2. Hardware Model. We assume the victim is running on a processor that soundly refines our abstract speculative processor model, ASP (§3). That is, all (sequential and transient) control- and data-flow that a program can exhibit on the hardware is captured by ASP. Real-world processors that qualify include Intel Alder Lake N (and other, §A.10) x86 CPUs, which feature: (i) Intel CET, (ii) RRSBAD and PSFD speculation controls [46], (iii) Data Operand Independent Timing (DOIT) [47], and (iv) the PHT, BTB, RSB, and STL speculation primitives.

Indirect Branch Tracking. Intel CET's *indirect branch tracking (IBT)* feature helps SERBERUS constrain *forward-edge* control-flow speculation. IBT requires that an ENDBR instruction be placed at all valid indirect branch targets. The processor raises an exception, or blocks transient execution, on an indirect jump to a non-ENDBR (§A.10). SERBERUS ensures ENDBRs are only placed at a program's procedure entrypoints, so the set of possible predicted targets for an indirect branch is the set of procedure entrypoints.

Shadow Stack and RRSBAD. Intel CET's *shadow stack (SHSTK)* and RRSBAD speculation control help SERBERUS constrain *backward-edge* control-flow speculation. RSB predictions are typically sourced from a hardware return stack buffer; (sequential or transient) calls push return addresses onto the return stack buffer which are popped off on return predictions. SHSTK ensures that transient out-of-bounds stores to return addresses on the software stack cannot hijack speculative control flow [100]. The RRSBAD (i.e., RRSBA_DIS_U) speculation control [46] prevents a processor from falling back to the indirect branch predictor to service return predictions on return stack buffer underfills [108]. Together, SHSTK and RRSBAD guarantee that the set of possible predicted targets for a return is exactly the set of instructions that immediately follow program calls.[8]

PSFD. We find that mitigating Spectre-PSF in software incurs significant performance overhead for the cryptographic workloads we evaluate (§7), while disabling PSF with Intel's PSFD speculation control introduces negligible overhead. We advocate for the latter strategy in this paper.

Data Operand Independent Timing. Intel's DOIT Mode [47] guarantees that the instructions which CT programming regards as safe (e.g., ADD, XOR, MUL) exhibit operand-independent timing. Given the guarantees of DOIT, our SERBERUS implementation assumes that the following instructions are transmitters: conditional branches, indirect branches, loads, stores, and division instructions. However, SERBERUS is *parameterized* by a set of user-identified transmitters to encompass others that may emerge [14], [66].

3. ASP: An Operational Semantics for an Abstract Speculative Processor

We define a novel operational semantics, called ASP, to support the design and prove the security of our Spectre mitigation SERBERUS. ASP consists of a *leakage model* and an *execution model* [25]. Its leakage model (§3.2) refines the CT leakage model. Its execution model (§3.3-§3.4) defines a *non-deterministic abstract speculative processor* that executes assembly-style programs as a series of state transitions. ASP captures the PHT, BTB, RSB, and STL speculation primitives. We show in §A.6 how to extend ASP to also capture PSF. However, since SERBERUS assumes it is disabled (for performance, §7), we omit it from core ASP.

3.1. ASP Preliminaries

We first define basic syntax, ASP's configurations (i.e., system states), and its assembly-style programs.

3.1.1. Storage and Labeled Data.

Registers. ASP has a finite set of general-purpose registers $\mathcal{R}_{\mathrm{gpr}}$, a stack pointer SP, a program counter PC, and a zero register ZR. We denote the set of all registers as $\mathcal{R} = \mathcal{R}_{\mathrm{gpr}} \cup \{\mathtt{SP}, \mathtt{PC}, \mathtt{ZR}\}$.

8. Note that the Linux kernel performs RSB filling on context switches to prevent cross-address-space Spectre-RSB attacks [110].

Values and security labels. ASP computes on security-labeled data. $\mathcal{V} \subseteq \mathbb{Z}$ is the set of data values that registers and data memory locations can assume ($0 \in \mathcal{V}$); $\mathcal{L} = \{\text{PUB}, \text{SEC}\}$ is the set of security labels, where PUB and SEC mark a public (low) and secret (high) value, respectively. $\mathcal{V}_\mathcal{L} = \mathcal{V} \times \mathcal{L}$ denotes the set of *labeled values*. We use either subscript v_l or pair notation (v, l) to attach a label $l \in \mathcal{L}$ to value $v \in \mathcal{V}$. Given a function $o : \mathcal{V}^n \to \mathcal{V}$ over unlabeled values, we define the labeled equivalent $o_\mathcal{L} : \mathcal{V}_\mathcal{L}^n \to \mathcal{V}_\mathcal{L}$ as $o_\mathcal{L}((v_1, l_1), \ldots, (v_n, l_n)) = o(v_1, \ldots, v_n)_L$ where the out-label $L = \text{SEC}$ iff *any* in-label $l_i = \text{SEC}$. All arithmetic operations in ASP propagate security labels in this way (§3.4.6), as in prior work [42].

Memory Addresses. We denote the set of mapped instruction and data addresses with $\mathcal{M}_I, \mathcal{M}_D \subseteq \mathcal{V}$, respectively. Defined by programs (§3.1.3) when execution starts (§3.3.4), these maps are fixed thereafter.

3.1.2. Configurations. A configuration of ASP is a five-tuple $C = (R, D, S, CS, T)$, where:

- $R : \mathcal{R} \to \mathcal{V}_\mathcal{L}$ is the labeled *register file* contents.
- $D : \mathcal{M}_D \to \mathcal{V}_\mathcal{L}$ is the labeled *data memory* contents.
- $S \in (\mathcal{M}_D \times \mathcal{V}_\mathcal{L})^*$ is the *speculative store set*, a list of (data address, labeled value) pairs.[9]
- $CS \in \mathcal{M}_I^*$ is the *call stack*, i.e., a list of return addresses.
- $T \in \{\text{SEQ}, \text{T}\}$ is the *transient execution bit*, which defines whether the configuration is transient ($T = \text{T}$) or sequential ($T = \text{SEQ}$). Instructions that take transient transitions (§3.3.3) set $T \leftarrow \text{T}$.

An *initial configuration* has the form $C_0 = (R_0[\texttt{PC} \leftarrow 0_{\text{PUB}}], D_0, (), (), \text{SEQ})$. $\mathcal{C}$ is the set of all configurations.

3.1.3. Instructions, Programs, Security Policies. ASP defines nine *instructions*:

$$\mathcal{I} = \texttt{JMP}\, d \mid \texttt{BNZ}\, r_{\text{src}}, d \mid \texttt{CALL}\, r_{\text{src}} \mid \texttt{RET} \mid \texttt{ENDBR} \mid \texttt{LFENCE}$$
$$\texttt{LD}\, [r_a + d], r_{\text{dst}} \mid \texttt{ST}\, [r_a + d], r_{\text{src}} \mid \texttt{OP}_o\, r_{\text{dst}}, \vec{r_s}$$

where $d \in \mathbb{Z}$; $r_{\text{src}}, r_a, r_{s,i} \in \mathcal{R} \setminus \texttt{PC}$; and $r_{\text{dst}} \in \mathcal{R}_{\text{gpr}} \cup \{\texttt{SP}\}$.

A *program* is a four-tuple $\mathcal{P} = (\mathcal{M}_I, \mathcal{M}_D, P, \mathbf{C}_0)$, where $\mathcal{M}_I$ and $\mathcal{M}_D$ (§3.1.1) are the mapped instruction and data addresses, respectively, and:

- $P : \mathcal{M}_I \to \mathcal{I}$ is the read-only *instruction memory* contents. We write $I \mapsto \mathbb{I}$ to mean $P(I) = \mathbb{I}$.
- $\mathbf{C}_0 \subseteq \mathcal{C}$ are the initial configurations of the program.

Each initial configuration $C_0 \in \mathbf{C}_0$ of program $\mathcal{P}$ defines the initial memory and register contents, which implicitly specifies the *security policy* (i.e., a labeling of all initial program values). Our security-typeability requirement (§4.3.2) constrains the security policy of initial configurations.

3.2. ASP's Leakage Model

Observations. ASP realizes the CT leakage model, or $[\![\cdot]\!]_{\text{ct}}$ (§2.2.2), by defining a set of four transmitters. When they execute (§3.3), transmitters expose their sensitive operand(s) as an observation taken from *observation set* $\mathcal{O}$:

$$\mathcal{O} = \varepsilon \mid \texttt{bnz}\, v_l \mid \texttt{call}\, v_l \mid \texttt{ld}\, A_l \mid \texttt{st}\, A_l$$

Observation `bnz` v_l exposes the labeled condition of a conditional branch (`BNZ`); `call` v_l exposes the labeled target of a call (`CALL`); `ld`/`st` A_l exposes the labeled address operand of a load/store (`LD`/`ST`). All other instructions are safe and expose the empty observation ε.

Return instructions (`RET`) are not transmitters, since they cannot leak *new* secrets on processors with a SHSTK (§2.3.2). On such designs, a secret return address implies a secret `PC` at some prior call (the one that pushed it onto the return stack buffer or SHSTK). A secret `PC` implies the execution of a prior `CALL` or `BNZ` with a secret sensitive operand. Thus, the secret already leaked.

In §A.6, we describe how to extend ASP with a fifth transmitter, division instructions (`DIV`). Without loss of generality, we omit `DIV` from core ASP. Our SERBERUS implementation (§6) *does* assume `DIV`s are transmitters.

Declassification. While *labeled* transmitter operands are exposed as observations, ASP declassifies transmitter operands (by assigning labels to PUB or stripping labels entirely) before using them to compute the next configuration (§3.4). Doing so eliminates *secondary* (repeated) leaks of the same secret while retaining *primary* (initial) leaks. Declassification is sound (i.e., it does not result in missed Spectre leakage), since SERBERUS ensures the program never passes a secret to a transmitter in the first place.

Execution model	PHT	BTB	RSB	STL	PSF	IBT	SHSTK	RRSBAD
Guarnieri et al. [102]	✓							
Cauligi et al. [42]	✓	✓*	✓*−	✓	✓			✓
Vassena et al. [34]	✓							
Fabian et al. [48]	✓		✓−	✓−				✓
ASP (ours)	✓	✓	✓	✓	✓*	✓	✓	✓

TABLE 2: A comparison of formal speculative execution models. ✓ indicates the model captures that speculation primitive (PHT, BTB, RSB, STL, PSF), speculative CFI protection (IBT, SHSTK), or speculation control (RRSBAD); ✓− indicates the model restricts the behavior of the speculation primitive (§3.3.1); ✓* indicates the speculation primitive is captured in an *extension* of the core execution model (e.g., §A.6 for ASP).

3.3. ASP's Execution Model: Overview

ASP captures all possible speculative executions of programs running on a design that satisfies our hardware assumptions (§2.3.2), using a *sequential semantics* (δ_{seq}, §3.3.2) and a *transient semantics* (δ_{t}, §3.3.3).

3.3.1. Limitations of Prior Execution Models. Several execution models have been proposed to support software analyses that detect or mitigate Spectre leakage in CT programs (Tab. 2) [25]. Two limitations of these models motivate us to design something new. First, none capture the semantics of Intel CET protections (IBT and SHSTK). Second, despite capturing various combinations of the five speculation primitives we target, these models make highly

9. X^* denotes the set of all sequences of elements from set X.

specific (and even unrealistic) assumptions about their microarchitectural implementations. For example, Fabian et al. [48] model STL by transiently "deleting" stores, which misses Spectre leakage on realistic designs.[10] Cauligi et al. [42] model RSB using an unrealistic infinite-size stack structure and thus miss leakage due to RSB overfills.

3.3.2. Sequential Transitions. Given a configuration C and instruction memory P, the *sequential (seq) transition* function $\delta_{\text{seq}}(C, P) = (C', O)$ returns a successor configuration C' and observation $O \in \mathcal{O}$. We say that "C sequentially yields C' exposing O," written $C \to^{O}_{\text{seq}} C'$. Each configuration has *exactly one* sequential successor, as there is only one sequential execution of a program.

3.3.3. Transient Transitions. The *transient (t) transition* function $\delta_{\text{t}}(C, P) \subseteq \mathcal{C} \times \mathcal{O}$ returns a *set* of (configuration, observation) pairs, capturing all transient execution steps that may arise on behalf of some speculation primitive (§2.2.1). We say "C transiently yields C' exposing O," written $C \to^{O}_{\text{t}} C'$, if $(C', O) \in \delta_{\text{t}}(C, P)$.

We say an instruction $I \in \mathcal{M}_I$ is a *speculation primitive* if there exists a configuration $C = (R, D, S, CS, T)$ satisfying $R(\texttt{PC}) = I$ that has both a transient transition (i.e., $\delta_{\text{t}}(C, P) \neq \emptyset$) and a non-faulting sequential transition (i.e., $\delta_{\text{seq}}(C, P) \neq (C, O)$, §3.4.4). In ASP, speculation primitives are instances of `BNZ`, `CALL`, `RET`, or `LD`. *Speculative execution* encompasses both sequential and transient execution (§3.3.4). Thus, we say C (speculatively) yields C' exposing O, written $C \to^{O} C'$, if $C \to^{O}_{\text{seq}} C'$ or $C \to^{O}_{\text{t}} C'$.

3.3.4. Traces. A *trace* of a program $\mathcal{P} = (\mathcal{M}_I, \mathcal{M}_D, P, \mathbf{C}_0)$ from an initial configuration $C_0 \in \mathbf{C}_0$ is the sequence of transitions $e = C_0 \to^{O_0} C_1 \to^{O_1} \cdots \to^{O_{n-1}} C_n$. We use subscript-$i$ notation to denote components of a configuration $C_i = (R_i, D_i, S_i, CS_i, T_i)$ at step i of a trace. We say $I_i = R_i(\texttt{PC})$, i.e., I_i is the address of the instruction executing step i. A trace $e = C_0 \to^{O_0} \cdots \to^{O_{n-1}} C_n$ is *transient* (resp. *sequential*) *at step* i if $T_i = \text{T}$ (resp. $T_i = \text{SEQ}$). If i is not specified, assume $i = n$, i.e., the last step in a trace. We say an *instruction* I_i executes transiently (resp. sequentially) if the trace is transient (resp. sequential) at step $i + 1$.

3.4. ASP's Instruction Execution Semantics

We specialize ASP's transition functions per instruction.[11] Assume current configuration $C = (R, D, S, CS, T)$, program $\mathcal{P} = (\mathcal{M}_I, \mathcal{M}_D, P, \mathbf{C}_0)$, and current instruction address $I = R(\texttt{PC})$. If I is unmapped ($I \notin \mathcal{M}_I$), the processor halts (i.e., $\delta_{\text{seq}}(C, P) = (C, \varepsilon)$ and $\delta_{\text{t}}(C, P) = \emptyset$).

10. Consider "`a=1; a=0; if (a /*L1*/) b = secret; if (a /*L2*/) leak(b);`" in which `secret` transiently leaks only if L1 skips over `a=0` (and reads from `a=1`), but L2 reads from `a=0`.

11. We use color to distinguish *sequential* and *transient* semantics, so readers should view the paper in color for a better experience.

3.4.1. Conditional Branches (PHT). The `BNZ` instruction captures the PHT speculation primitive. Its *sequential* transition takes the branch by adding a fixed displacement d to the `PC` if the value in register r is nonzero; otherwise, it falls through to the next instruction. The *transient* transition does the opposite, modeling a branch misprediction. Both transitions expose the labeled branch condition via observation $\texttt{bnz}\, R(r)$ and then declassify it before computing the branch target.

$$\textsc{Cond. Br.}\quad I \mapsto \texttt{BNZ}\, r, d \qquad v_l = R(r) \qquad c = (v \neq 0) \qquad O = \texttt{bnz}\, v_l$$
$$I'_{\text{seq}} = I + 1 + c \cdot d \qquad I'_{\text{t}} = I + 1 + (1 - c) \cdot d \qquad R'_{\text{t}} = R[\texttt{PC} \leftarrow (I'_{\text{t}})_{\text{PUB}}]$$
$$R'_{\text{seq}} = R[\texttt{PC} \leftarrow (I'_{\text{seq}})_{\text{PUB}}] \quad C'_{\text{t}} = C[R \leftarrow R'_{\text{t}}; T \leftarrow \text{T}] \quad C'_{\text{seq}} = C[R \leftarrow R'_{\text{seq}}]$$
$$\delta_{\text{seq}}(C, P) = (C'_{\text{seq}}, O) \qquad \boldsymbol{\delta_{\text{t}}(C, P) = \{(C'_{\text{t}}, O)\}}$$

3.4.2. Calls (BTB, IBT). The `CALL` and `ENDBR` instructions together model the BTB speculation primitive and Intel's IBT. `CALL`'s *sequential* transition checks whether the target instruction at $I'_{\text{seq}} = R(r)$ is an `ENDBR`. If so, it jumps to I'_{seq}; if not, execution halts due to a CFI violation. A *transient* transition may jump to *any* $I'_{\text{t}} \mapsto \texttt{ENDBR}$ in the program ($I'_{\text{t}} \neq I'_{\text{seq}}$). All transitions push the return address $I_r = I + 1$ onto the callstack CS, expose the (sequential) target via observation $\texttt{call}\, R(r)$, and declassify the resulting `PC`.

$$\textsc{Call}\quad I \mapsto \texttt{CALL}\, r \qquad (I'_{\text{seq}})_l = R(r) \qquad I_r = I + 1$$
$$\mathbf{I}'_{\text{t}} = \{I'_t \mid I'_t \mapsto \texttt{ENDBR}\} \setminus I'_{\text{seq}} \qquad CS' = (CS :: I_r) \qquad O = \texttt{call}\, R(r)$$
$$R'_{\text{seq}} = R[\texttt{PC} \leftarrow (I'_{\text{seq}})_{\text{PUB}}] \qquad \mathbf{R}'_{\text{t}} = \{R[\texttt{PC} \leftarrow (I'_{\text{t}})_{\text{PUB}}] \mid I'_{\text{t}} \in \mathbf{I}'_{\text{t}}\}$$
$$\delta_{\text{seq}}(C, P) = (C[R \leftarrow R'_{\text{seq}};\ CS \leftarrow CS'], O)$$
$$\boldsymbol{\delta_{\text{t}}(C, P) = \{(C[R \leftarrow R'_{\text{t}};\ CS \leftarrow CS';\ T \leftarrow \text{T}], O) \mid R'_{\text{t}} \in \mathbf{R}'_{\text{t}}\}}$$

The sole purpose of the `ENDBR` instruction is to mark valid targets of calls; it executes as a no-op.

$$\textsc{Endbr}\quad I \mapsto \texttt{ENDBR} \qquad R'_{\text{seq}} = R[\texttt{PC}{+}{+}]$$
$$\delta_{\text{seq}}(C, P) = (C[R \leftarrow R'_{\text{seq}}], \varepsilon) \qquad \boldsymbol{\delta_{\text{t}}(C, P) = \emptyset}$$

3.4.3. Returns (RSB, SHSTK). The `RET` instruction captures the RSB speculation primitive. If the callstack is empty, the processor halts execution. Otherwise, it pops the sequential return address I'_{seq} off the callstack. The *sequential* transition jumps to I'_{seq}; the *transient* transition jumps to *any callsite in the program*, i.e., any instruction address I'_{t} immediately following a `CALL`.

$$\textsc{Return}\ I \mapsto \texttt{RET} \quad (CS' :: I'_{\text{seq}}) = CS \quad R'_{\text{seq}} = R[\texttt{PC} \leftarrow (I'_{\text{seq}})_{\text{PUB}}]$$
$$\mathbf{R}'_{\text{t}} = \{R[\texttt{PC} \leftarrow (I'_{\text{t}})_{\text{PUB}}] \mid I'_{\text{t}} - 1 \mapsto \texttt{CALL}\} \setminus R'_{\text{seq}} \quad C' = C[CS \leftarrow CS']$$
$$C'_{\text{seq}} = C'[R \leftarrow R'_{\text{seq}}] \quad \mathbf{C}'_{\text{t}} = \{C'[R \leftarrow R'_{\text{t}}; T \leftarrow \text{T}] \mid R'_{\text{t}} \in \mathbf{R}'_{\text{t}}\}$$
$$\delta_{\text{seq}}(C, P) = \begin{cases} (C, \varepsilon) & \text{if } CS \text{ empty} \\ (C'_{\text{seq}}, \varepsilon) & \text{otherwise} \end{cases} \qquad \boldsymbol{\delta_{\text{t}}(C, P) = \mathbf{C}'_{\text{t}} \times \{\varepsilon\}}$$

ASP captures a wider range of transient behaviors due to RSB than prior work (§3.3.1). Our approach captures all return predictor implementations that can predict previously seen return addresses only (e.g., return stack buffers).

Shadow Stack. `CALL`/`RET` semantics and the callstack CS together model Intel's SHSTK. By maintaining the call stack outside of data memory, ASP prevents stores

from overwriting return addresses, transiently or otherwise. When ASP executes a RET, it ensures that *sequential* control returns to the correct callsite; *transient* control may return to another, incorrect callsite.

3.4.4. Loads and Stores (STL). A LD/ST instruction computes its labeled effective address $A_l = R(r_a) + d_{\text{PUB}}$ and exposes it via observation $\texttt{ld}/\texttt{st}\ A_l$. Before performing the memory access, A_l is declassified by stripping its label to produce A (recall that data addresses are unlabeled, §3.1).

Store. Consider a store (ST). If A is unmapped, then it has one *sequential* transition that halts execution (modeling a fault) and one *transient* transition that executes as a no-op (modeling out-of-order execution). If A is mapped, then the store has one *sequential* transition that appends the (declassified address, value) pair $(A, R(r))$ to the speculative store set and no transient transitions. Data memory D is not updated until a speculation fence executes (§3.4.5).

$$\frac{\begin{array}{l}\textsc{Store}\quad I \mapsto \texttt{ST}\,[r_a + d], r \qquad A_l = R(r_a) + d_{\text{PUB}} \qquad O = \texttt{st}\ A_l \\ S'_{\text{seq}} = (S :: (A, R(r))) \qquad R' = R[\texttt{PC++}] \\ C'_{\text{seq}} = C[R \leftarrow R'; S \leftarrow S'_{\text{seq}}] \qquad \mathbf{C}'_{\text{t}} = \{(C[R \leftarrow R'; T \leftarrow \text{T}], O)\}\end{array}}{\delta_{\text{seq}}(C,P) = \begin{cases}(C'_{\text{seq}}, O) & \text{if } A \in \mathcal{M}_D \\ (C, O) & \text{if } A \notin \mathcal{M}_D\end{cases} \quad \boldsymbol{\delta_{\text{t}}(C,P)} = \begin{cases}\emptyset & \text{if } A \in \mathcal{M}_D \\ \mathbf{C}'_{\text{t}} & \text{if } A \notin \mathcal{M}_D\end{cases}}$$

Load. Consider a load (LD), whose semantics captures the STL speculation primitive. If A is unmapped, then it has one *sequential* transition that halts execution (modeling a fault) and one *transient* transition that assigns zero to its output register (modeling recent Intel processors, including our workstation [82]). If A is mapped, then the load has one *sequential* transition that reads from the most recent same-address store. It may have more than one transient transition, each of which reads from a distinct same-address store in the speculative store set *or* from data memory.

$$\frac{\begin{array}{l}\textsc{Load}\quad I \mapsto \texttt{LD}\,[r_a + d], r \qquad ((A_1, (v_1)_{l_1}), \ldots, (A_n, (v_n)_{l_n})) = S \\ A_l = R(r_a) + d_{\text{PUB}} \qquad D_{\text{seq}} = D[A_1 \leftarrow (v_1)_{l_n}; \cdots; A_n \leftarrow (v_n)_{l_n}] \\ v_{\text{seq}} = D_{\text{seq}}(A) \qquad \mathbf{v}_{\text{t}} = \{D(A)\} \cup \{(v_i)_{l_i} \mid A_i = A\} \qquad O = \texttt{ld}\ A_l \\ R'_{\text{seq}} = R[\texttt{PC++}; r \leftarrow v_{\text{seq}}] \qquad \mathbf{R}'_{\text{t}} = \{R[\texttt{PC++}; r \leftarrow v_{\text{t}}] \mid v_{\text{t}} \in \mathbf{v}_{\text{t}}\}\end{array}}{\delta_{\text{seq}}(C,P) = \begin{cases}(C[R \leftarrow R'_{\text{seq}}], O) & \text{if } A \in \mathcal{M}_D \\ (C, O) & \text{if } A \notin \mathcal{M}_D\end{cases}}$$

$$\boldsymbol{\delta_{\text{t}}(C,P)} = \begin{cases}\{(C[R \leftarrow R'_{\text{t}}; T \leftarrow \text{T}], O) \mid R'_{\text{t}} \in \mathbf{R}'_{\text{t}}\} & \text{if } A \in \mathcal{M}_D \\ \{(C[R \leftarrow R[\texttt{PC++}; r \leftarrow 0_{\text{PUB}}]; T \leftarrow \text{T}], O)\} & \text{if } A \notin \mathcal{M}_D\end{cases}$$

As with RSB, ASP models STL in an *implementation-agnostic* manner. It captures a *superset* of the STL behaviors captured by prior execution models (§3.3.1).

NCA/CA accesses. We partition memory accesses into two classes which differ in their ability to introduce secrets into transient computation, as we show in §4–5.

Definition 3.1. A memory access $I \mapsto \texttt{LD}/\texttt{ST}\,[r_a + d], r$ is *constant-address (CA)* if $r_a \in \{\texttt{ZR}, \texttt{SP}\}$; otherwise, I is *non-constant-address (NCA)*. Furthermore, if $r_a = \texttt{ZR}$, I is a CA *global access*, and displacement d is the fixed address of a global variable. If $r_a = \texttt{SP}$, I is a CA *stack access*, and d gives the frame offset of a stack variable.

3.4.5. Speculation Fence. ASP features a *speculation fence* instruction, LFENCE, which halts transient execution but allows sequential execution to proceed. Its semantics depends on whether the current configuration is sequential ($T = \text{SEQ}$) or transient ($T = \text{T}$). If it is *sequential*, LFENCE drains all stores in the speculative store set S to data memory, thereby restricting the set of stores that loads may transiently forward data from (§3.4.4). If it is *transient*, execution halts.

$$\frac{\begin{array}{l}\textsc{Speculation Fence}\quad I \mapsto \texttt{LFENCE} \qquad R'_{\text{seq}} = R[\texttt{PC++}] \\ ((A_1, (v_1)_{l_1}), \ldots, (A_n, (v_n)_{l_n})) = S \\ D'_{\text{seq}} = D[A_1 \leftarrow (v_1)_{l_1}; \cdots; A_n \leftarrow (v_n)_{l_n}] \\ C'_{\text{seq}} = C[R \leftarrow R'_{\text{seq}};\ D \leftarrow D'_{\text{seq}};\ S \leftarrow ()]\end{array}}{\delta_{\text{seq}}(C,P) = \begin{cases}(C, \varepsilon) & \text{if } T = \text{T} \\ (C'_{\text{seq}}, \varepsilon) & \text{otherwise}\end{cases} \qquad \boldsymbol{\delta_{\text{t}}(C,P) = \emptyset}}$$

3.4.6. Other Instructions. JMP *sequentially* jumps to PC+d+1. $\texttt{OP}_o$ represents a class of arithmetic operations (e.g., MOV, ADD), parameterized by function $o : \mathcal{V}^n \to \mathcal{V}$. $\vec{r_s}$ is a list of input registers; r is the output register. Its *sequential* transition assigns $r \leftarrow o_{\mathcal{L}}(R(r_{s,1}), \ldots, R(r_{s,n}))$. Neither JMP nor OP have transient transitions. We omit transition rules here for brevity, but provide them in §A.5.

4. Characterizing Spectre Leakage in Static Constant-Time Programs

4.1. Speculative Constant-Time

We formalize Spectre leakage of secrets in CT programs as a violation of the *speculative constant-time (SCT)* security property from prior work [34], [41], [42]. A program satisfies SCT on ASP if there does not exist a trace that exposes secret-dependent observations. An *SCT violation* is a trace that exposes some secretly-labeled observation O. Cauligi et al. [42] show that all Spectre attacks manifest as SCT violations.

Definition 4.1. A program $\mathcal{P}$ is SCT iff for all traces $e = C_0 \to^{O_0} \cdots \to^{O_n} C_{n+1}$, no observation O_i is labeled secret for any $0 \leq i \leq n$.

4.2. Limitations of Traditional CT

Definition 4.2. A program $\mathcal{P}$ is *constant-time (CT)* iff for all *sequential* traces $e = C_0 \to^{O_0}_{\text{seq}} \cdots \to^{O_n}_{\text{seq}} C_{n+1}$, no observation O_i is labeled secret for any $0 \leq i \leq n$.

We observe that two limitations of Def. 4.2 prevent efficient mitigation of SCT violations in CT programs.

First, a CT program may contain *latent CT violations*, such as "`if (0) x = A[secret]`," that do not manifest in any sequential trace of the program. Such patterns exhibit compile-time *security-type violations* by assigning

a secret value (e.g., `secret`) to a public variable or supplying it to a public operand (e.g., the index operand in `A[secret]`). Existing CT compilers [13], [72] detect such security-type violations during compile-time typechecking. We introduce a *security-typeability* requirement (§4.3.2) for CTS programs, which formalizes the security-type guarantees of a CT program that has passed such typechecks.

Second, CT programs use a *Spectre-unaware calling convention* that permits passing secrets by value during calls and returns. This is *inherently unsafe* in the presence of BTB or RSB mispredictions, which can easily leak secret arguments: the call (resp. return) need only transiently jump to a procedure (resp. callsite) that expects a *public* argument in a register, which it subsequently leaks by supplying it to a transmitter. Secret argument leakage is difficult to mitigate efficiently: it forces a mitigation to conservatively assume *all arguments and return values* may be transiently secret. No combination of currently deployed mitigations for the CT leakage model can fully protect against this kind of leakage.[12]

4.3. Static Constant-Time Programming

Definition 4.3. A program $\mathcal{P}$ is *static constant-time (CTS)* iff it satisfies CT (Def. 4.2) as well as **WF** (*well-formed, §4.3.1*) and **TYP** (*security-typeable, §4.3.2*).

We propose *static constant-time (CTS) programming*, a strengthening of CT programming which overcomes the limitations in §4.2. We find that existing CT programs generally satisfy CTS when compiled with `-O3` optimizations and a carefully selected set of compiler flags. §A.8 provides a full list of these flags for LLVM, which disable optimizations that are incompatible with CTS, like stack slot sharing and argument promotion. Using these flags, *all* of the cryptographic primitives we benchmark in §6 satisfy CTS without requiring *any* source modifications.

4.3.1. Well-Formed. *Well-formedness* captures the important structural and behavioral properties and metadata of compiled code that can be leveraged by a compiler-based software mitigation like SERBERUS (§5).

The *intraprocedural successors* $\mathrm{succs}(I) \subseteq \mathcal{M}_I$ of an instruction address $I \in \mathcal{M}_I$ are defined as:

$$\mathrm{succs}(I) = \begin{cases} \emptyset & \text{if } I \mapsto \texttt{RET} \\ \{I+1, I+1+d\} & \text{if } I \mapsto \texttt{BNZ}\, r, d \\ \{I+1+d\} & \text{if } I \mapsto \texttt{JMP}\, d \\ \{I+1\} & \text{otherwise.} \end{cases}$$

A *procedure* $F \subseteq \mathcal{M}_I$ is a set of instruction addresses that (i) contains exactly one `ENDBR` instruction, denoting the entrypoint; (ii) is closed under intraprocedural succession; (iii) has a prologue on entry allocating a stack frame of fixed size k_F; (iv) has an epilogue deallocating the frame on return; and (v) has only in-bounds stack accesses into its frame. Fig. 3 shows an example procedure. F_i denotes the procedure containing the instruction I_i executing in ith step of a trace.

Definition 4.4 (Well-formed programs **WF**)**.** A program $\mathcal{P}$ is *well-formed* if it satisfies the following:

1) *Procedures only*: $\mathcal{P}$ consists of only procedures.
2) *Calling convention:* $\mathcal{P}$ defines a *calling convention* $\mathcal{A}$: $\mathcal{M}_I \rightharpoonup \mathcal{R}_{\mathrm{gpr}}$, a map from `CALL`/`RET` instructions to the set of registers used to pass arguments.
3) *Data stack*: $\mathcal{P}$ has a *data stack* $DS \subseteq \mathcal{M}_D$, represented as a set of data addresses, such that (a) the stack pointer `SP` is always public and always points into DS in all traces; (b) no global variables are in stack memory;[13] and (c) DS is zero-initialized in all initial configurations.
4) *No callee-saved registers:* No general-purpose registers are preserved across procedure calls.
5) *No segfaults*: No sequential traces access an unmapped data address outside the stack DS.

4.3.2. Security-Typeable. Intuitively, a well-formed program $\mathcal{P}$ is security-typeable if each variable can be statically typed with a security label that is never violated at runtime.[14] Note that SERBERUS does not require such a security labeling to be explicitly provided as input; SERBERUS *does not require any program annotations whatsoever*. Instead, SERBERUS relies on the following properties afforded by security-typeable CTS programs to infer an upper bound on what program variables may hold secret values (i.e., secretly-labeled values, §3.1.1) at runtime.

Definition 4.5 (**TYP**)**.** A program $\mathcal{P}$ is *security-typeable* if there exists (i) a *global variable typing* $\tau_{\mathrm{glob}} : \mathcal{M}_G \to \mathcal{L}$ where $\mathcal{M}_G \subseteq \mathcal{M}_D$ is the set of data addresses accessed by CA global accesses (Def. 3.1); (ii) a *stack variable typing* $\tau^F_{\mathrm{stk}} : [0, k_F) \to \mathcal{L}$ for each procedure F (recall k_F is F's frame size); and (iii) a *register variable typing* $\tau^F_{\mathrm{reg}} : \mathcal{R} \times F \to \mathcal{L}$ for each procedure F. The typings must also pass the following typechecking rules (where I is an instruction in some procedure F):

1) *Conservative:* No security type is violated in any sequential trace—i.e., no publicly-typed global, stack, or register variable ever holds a secret value, and no secret values are read from or written to a publicly-typed stack variable.
2) *Transmitters:* All sensitive register operands of transmitters (§3.2) are publicly-typed in τ^F_{reg}.
3) *Load consistency:* The destination register of a CA load $I \mapsto \texttt{LD}\,[\texttt{ZR}/\texttt{SP}+d], r$ from a secretly-typed global/stack variable is secretly-typed in τ^F_{reg}.
4) *Store consistency:* The source register of a CA store $I \mapsto \texttt{ST}\,[\texttt{ZR}/\texttt{SP}+d], r$ to a publicly-typed global/stack variable is publicly-typed in τ^F_{reg}.
5) *Policy consistency:* Publicly-typed global variables in τ_{glob} contain public values in initial configurations of $\mathcal{P}$.

12. Even if LFENCE/SLH, RETPOLINE, and SSBD are simultaneously enabled, return values may *still* leak via RSB speculation.

13. I.e., no CA global loads/stores access the stack DS.

14. *Security-typeable* is implicitly defined with respect to $\mathcal{P}$'s initial configuration set $\mathbf{C}_0$, which defines $\mathcal{P}$'s security policy (§3.1.3).

6) *Always-public registers:* The stack pointer SP and program counter PC are always publicly-typed in τ^F_{reg}.
7) *Register deps:* The output of an OP instruction is secretly-typed in τ^F_{reg} iff any input is secretly-typed in τ^F_{reg}.
8) *No-op:* Types of registers unmodified by instruction I do not change in τ^F_{reg} from I to $J \in \text{succs}(I)$.
9) *Public arguments:* All argument registers $\mathcal{A}(I)$ at each $I \mapsto \texttt{CALL} \mid \texttt{RET}$ are publicly-typed in τ^F_{reg}.

TYP.1–2 imply a CT program in the traditional sense (per Def. 4.2, proven in §A.1). TYP.1–8 codify the static security properties of CT programs that prior software-based mitigations implicitly assume of their input [34] and CT compilers guarantee of their output [13], resolving the first limitation in §4.2. TYP.9 resolves the second limitation in §4.2 by requiring that *all secrets arguments be passed by reference* rather than by value; that is, one must pass a public pointer to a secret rather than the secret value itself.

4.4. Taint Primitives in a CTS Program

We now provide a complete characterization of Spectre leakage in CTS programs, which gives rise to a powerful Spectre mitigation approach (§4.4.3).

4.4.1. Dynamic DFG. The *dynamic data-flow graph (DFG)* for a trace e is a directed acyclic graph where nodes are register-step pairs (r, i) and edges $(r, i) \to^e_{\text{dep}} (r', j)$ encode direct *dynamic* register or memory dependencies in the trace as follows:

- *No-op:* $(r, i) \to^e_{\text{dep}} (r, i+1)$ if r is not modified by I_i.
- *Register dependency:* $(r, i) \to^e_{\text{dep}} (r', i+1)$ if $I_i \mapsto \texttt{OP}\, r', \vec{r_s}$ and $r \in \vec{r_s}$.
- *Memory dependency:* $(r, i) \to^e_{\text{dep}} (r', j{+}1)$ if a store $I_i \mapsto \texttt{ST}\,[r_a + d], r$ sources a later load $I_j \mapsto \texttt{LD}\,[r'_a + d'], r'$. Unlike the prior register dependencies, memory dependencies can span many steps in the trace, since the store/load may not execute consecutively.

We say (r', j) is *dynamic-dependent* on (r, i) if $(r, i) \to^e_{\text{dep}*} (r', j)$, i.e., $(r, i) \to^e_{\text{dep}} \cdots \to^e_{\text{dep}} (r', j)$. We say (r', j) is *dynamic-dependent on a load* $I_i \mapsto \texttt{LD}\,[r_a{+}d], r$ if $(r, i+1) \to^e_{\text{dep}*} (r', j)$.

We use a one-step delay $(r, i{+}1)$ to reference the output of an instruction I_i updating register r (e.g., a load $I_i \mapsto \texttt{LD}\,[r_a + d], r$). This is because the updated register does not hold its new value until the *next* configuration, C_{i+1}.

4.4.2. Taint Primitives.

Definition 4.6. A *security-type violation* is a register-step pair (r, i) where r is publicly-typed at I_i (i.e., $\tau^{F_i}_{\text{reg}}(r, I_i) = \text{PUB}$, Def. 4.5) but r holds a secret value at step i (i.e., $R_i(r) = v_{\text{SEC}}$, §3.1.1).

Definition 4.7 (Taint primitives)**.** Instruction I_i executing at step i in trace e is a *taint primitive* if there is a security-type violation at step $i+1$ (Def. 4.6) that is not dynamic-dependent (§4.4.1) on any prior security-type violations.

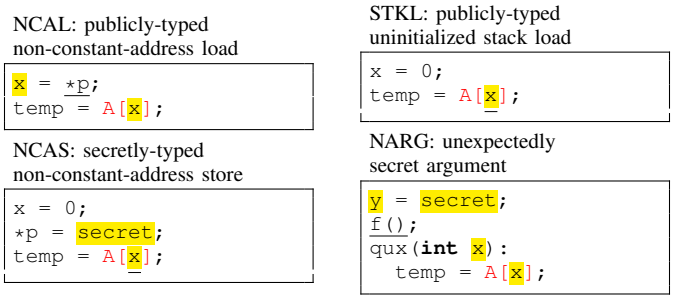

Figure 1: Exactly four kinds of *taint primitives* introduce transient security type violations in CTS programs, given our hardware model (§2.3.2). Taint primitives can be enabled by *any* speculation primitive. Taint primitives are underlined; secrets are highlighted; transmitters are red.

Intuitively, Def. 4.7 says a *taint primitive* is an instruction whose execution introduced a *new* security-type violation into the computation, since no inputs to I_i violated their security types but the *output* of I_i did. Now, we prove that CTS programs in ASP contain exactly *four classes* of taint primitives (see Fig. 1).

Theorem 4.1 (Taint primitives)**.** Every taint primitive I_i in any trace e can be classified as one of the following *transient instructions* (with the register violating its security type at step $i+1$ in parentheses):

- **NCAL:** a transient NCA load (output register).
- **NCAS:** a transient CA load that reads from a transient NCA store (output register).
- **STKL:** a transient CA stack load (output register).
- **NARG:** a transient CALL/RET (non-argument register).

Proof. We will prove the claim directly. Let I_i be a taint primitive in a trace e. By Def. 4.7, I_i introduces a new security-type violation in some register r' at step $i+1$.

Suppose I_i does not modify r'. Then $(r', i) \to^e_{\text{dep}} (r', i+1)$ and r' is secretly-typed at i but publicly-typed at $i+1$ (Def. 4.7). Since the security type of r' cannot change intraprocedurally if r' is unmodified (TYP.8), $I_i \mapsto \texttt{CALL} \mid \texttt{RET}$. Secretly-typed r' cannot be an argument register (TYP.9), so I_i satisfies **NARG**.

Suppose I_i modifies r'. First, note that $r' \neq \texttt{PC}$ since ASP's declassification of transmitter operands (§3.2) ensures that PC always holds a public value. Only two instructions modify non-PC registers: OP and LD.

If $I_i \mapsto \texttt{OP}_o\, r', \vec{r_s}$, then some input $r \in \vec{r_s}$ violated its security type at i (TYP.7). But $(r, i) \to^e_{\text{dep}} (r', i+1)$, so I_i is not a taint primitive (Def. 4.7), a contradiction.

If $I_i \mapsto \texttt{LD}\,[r_a{+}d], r'$, then I_i may be an NCA, CA stack, or CA global load. If I_i is an NCA load (resp. CA stack load), it satisfies **NCAL** (resp. **STKL**). If I_i is a CA global load, we know it did not read from a CA stack store (WF.3) or initial memory (TYP.1), so it read from an NCA store or CA global store. In the former case, the store was not sequential (TYP.1), satisfying **NCAS**. In the latter, I_i read from a CA global store of a prior security type violation (TYP.4), a contradiction (Def. 4.7). □

4.4.3. Mitigating Spectre in CTS Programs.

Corollary 4.1.1. If a CTS program $\mathcal{P}$ violates SCT (Def. 4.1), then there exists a trace in which a transmitter's sensitive operand is dynamic-dependent on an NCAL, NCAS, STKL, or NARG taint primitive.

Proof. If $\mathcal{P}$ violates SCT, some trace e of $\mathcal{P}$ executes a transmitter I_j at step j with a secretly-labeled sensitive operand r (Def. 4.1). By TYP.2, r is publicly-typed at I_j, so (r, j) is a security-type violation (Def. 4.6). Thus, I_j is dynamic-dependent on some taint primitive I_i (Def. 4.7). By Thm. 4.1, I_i is a NCAL, NCAS, STKL, or NARG. □

5. SERBERUS: A Compiler Approach for Enforcing Speculative Constant Time

Cor. 4.1.1 is powerful: to eliminate *all Spectre leakage* in a CTS program, a mitigation must simply break all dynamic dependencies from four classes of taint primitives (Thm. 4.1) to subsequent transmitters. SERBERUS consists of three intraprocedural passes (Fig. 2) that do just this.

5.1. SERBERUS's Fence Insertion Pass

SERBERUS runs the *Fence Insertion Pass* first to mitigate *all* SCT violations due to NCAS and *some* due to NCAL taint primitives *in each procedure* F. We formulate optimal fence insertion as a graph cut problem over F's weighted transient CFG (§5.1.1): we must eliminate all transient control-flow paths from *sources* to *sinks* (defined in §5.1.3).

5.1.1. Transient CFG. First, SERBERUS constructs a weighted *transient CFG* (T-CFG) for F which captures the set of all transient control-flow paths through F. Unlike a traditional procedural CFG, the T-CFG captures transient execution through F *and* across invocations of F or spurious RSB-mispredicted returns to F. Nodes are instructions in F. Edges are defined by the *transient successor* function:

$$\mathbf{I}_{\text{enter}} = \{J \in F \mid J \mapsto \texttt{ENDBR} \text{ or } J-1 \mapsto \texttt{CALL}\}$$
$$\mathbf{I}_{\text{exit}} = \{I \in F \mid I \mapsto \texttt{CALL} \mid \texttt{RET}\}$$
$$\text{tsuccs}(I) = \begin{cases} \mathbf{I}_{\text{enter}} & \text{if } I \in \mathbf{I}_{\text{exit}} \\ \emptyset & \text{if } I \mapsto \texttt{LFENCE} \\ \text{succs}(I) & \text{otherwise (§4.3.1)} \end{cases}$$

The T-CFG has the edge $I \to^F_{\text{tcfg}} J$ iff $J \in \text{tsuccs}(I)$. We write $I \to^F_{\text{tcfg}*} J$ to indicate there is a path from I to J in F's T-CFG. Notably, LFENCEs have no transient successors since they block transient execution (§3.4.5) and are thus dead-ends in the T-CFG. We compute the *weight* of an edge $I \to^*_{\text{tdfg}} J$ as $w = L/D$, where L is the loop nest depth and D is the depth in the dominator tree of instruction J. This estimates the relative execution frequency of J and thus the cost of placing a fence before it. Weights affect optimality (performance) but not correctness of the min-cut (§5.1.4).

Theorem 5.1 (T-CFG complete)**.** If same-procedure instructions $I_i, I_j \in F$ transiently execute at steps $i < j$ of some trace of a program $\mathcal{P}$, then $I_i \to^F_{\text{tcfg}*} I_j$. (*Proof.* See §A.2.)

5.1.2. Static DFG. Next, SERBERUS constructs a *static DFG* for F, which models syntactic intraprocedural dependencies through CA stack accesses and registers. Nodes are register-instruction pairs $(r, I) \in \mathcal{R} \times F$. Edges $(r, I) \to^F_{\text{dep}} (r', J)$ encode direct register or stack dependencies as follows:

- *No-op:* $(r, I) \to^F_{\text{dep}} (r, J)$ if $J \in \text{tsuccs}(I)$ and I does not modify r.
- *Register dep:* $(r, I) \to^F_{\text{dep}} (r', I+1)$ for each $r \in \vec{r_s}$ if $I \mapsto \texttt{OP}_o r', \vec{r_s}$.
- *Stack dep:* $(r, I) \to^F_{\text{dep}} (r', J+1)$ if $I \mapsto \texttt{ST}\,[\texttt{SP}+d], r$ and $J \mapsto \texttt{LD}\,[\texttt{SP}+d], r'$; i.e., if I and J are a same-offset CA stack store and load, then J may read from I.

We say (r', J) is *static-dependent* on (r, I) if $(r, I) \to^F_{\text{dep}*} (r', J)$. Furthermore, we say (r', J) is *static-dependent on a load* $I \mapsto \texttt{LD}\,[r_a + d], r$ if $(r, I+1) \to^F_{\text{dep}*} (r', J)$.

The static DFG enables SERBERUS to precisely track how candidate NCAL taint primitives may propagate security-type violations through intraprocedural dependencies. Note the similarities with dynamic-dependencies (§4.4.1), like the one-step delay for static DFG nodes referencing an instruction's *output*. We use these similarities in our final proof of SERBERUS's correctness (Thm. 5.7).

5.1.3. Source-Sink Pair Generation. SERBERUS identifies *five types* of intraprocedural *source-sink* instruction pairs that can produce SCT violations involving an NCAL or NCAS primitive *when both source and sink execute transiently*. The source of each pair is an NCA load/store, which we conservatively assume may read/write a secret at an *arbitrary* data address when executed transiently. SERBERUS accumulates a set S of source-sink pairs as follows.

A source-sink pair (I, J) is added to S for each static dependency $(r, I) \to^F_{\text{dep}*} (r', J)$, where $I \mapsto \texttt{LD}\,[r_a + d], r$ (I is an NCA load) and J is (i) a transmitter with sensitive operand r' (NCAL-XMIT); (ii) a CALL/RET with argument $r' \in \mathcal{A}(J)$ (NCAL-ARG); or (iii) a CA global store of r' (NCAL-GLOB). These pairs help SERBERUS prevent NCAL taint primitives from passing secrets (i) intraprocedurally to transmitters or (ii) interprocedurally in arguments, or (iii) enabling stores of secrets to publicly-typed global variables.

A source-sink pair (I, J) is added to S for each NCA store I paired with (iv) each CA load J (NCAS-CAL) and (v) each CALL/RET J (NCAS-CTRL). These pairs help SERBERUS prevent all candidate NCAS instructions (CA loads) from (iv) intraprocedurally or (v) interprocedurally reading from a transient NCA store I.

5.1.4. Fence Insertion. Given the set of source-sink pairs S (§5.1.3) and the T-CFG for F (§5.1.1), SERBERUS runs a heuristic minimum directed multicut algorithm (§A.7) to obtain a near-optimal edge cutset C that prevents (transient) sources from passing data to sinks. That is, SERBERUS inserts an LFENCE along each edge $(u, v) \in C$.

5.1.5. Guarantees. SERBERUS's Fence Insertion Pass produces a partially mitigated CTS program $\mathcal{P}_{\text{fence}}$ that contains *no* SCT violations caused by NCAS taint primitives

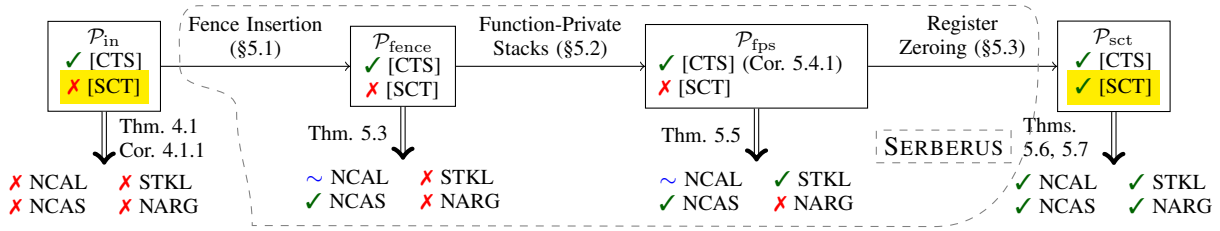

Figure 2: Given a CTS program $\mathcal{P}_{\text{in}}$ (§4.3), SERBERUS runs three passes, each of which provably eliminate a class of taint primitives (§4.4.2). Thm. 5.7 proves all passes together eliminate the final primitive, NCAL and thus the output CTS program $\mathcal{P}_{\text{sct}}$ satisfies SCT (Def. 4.1).

(✓ in Fig. 2). Some NCAL primitives remain (∼), which subsequent passes will eliminate.

Theorem 5.2 (Sources execute sequentially)**.** If (I_i, I_k) is a source-sink pair in a trace of $\mathcal{P}_{\text{fence}}$, I_i did not trap, and $i < k$, then some $I_j \mapsto$ `LFENCE` executes between I_i and I_k, so I_i executed sequentially. (*Proof.* See §A.3.)

Theorem 5.3 (No NCAS in $\mathcal{P}_{\text{fence}}$)**.** No instructions belong to NCAS (§4.4.2) in any trace of $\mathcal{P}_{\text{fence}}$.

Proof. Suppose for contradiction some I_k satisfies NCAS in trace e of $\mathcal{P}_{\text{fence}}$, i.e., I_k is a CA load that read from a prior transient NCA store I_i. If an instruction $I_j \mapsto$ `CALL`/`RET` executes for $i<j<k$, then (I_i, I_j) is a NCAS-CTRL source-sink pair. Else, (I_i, I_k) is a NCAS-CAL pair. Either way, I_i executes sequentially (Thm. 5.2), a contradiction. □

5.2. SERBERUS's Function-Private Stacks Pass

SERBERUS runs the *Function-Private Stacks (FPS) Pass* second to eliminate all STKL taint primitives in $\mathcal{P}_{\text{fence}}$, producing a functionally-equivalent program $\mathcal{P}_{\text{fps}}$. Our insight is that SCT violations due to STKL arise due to procedures reallocating frames on the same data stack. Thus, SERBERUS statically assigns a *private stack* to each procedure F on which only F can allocate stack frames.

5.2.1. Motivation. If procedures share the same data stack, it is difficult to ensure that a publicly-typed stack access will not read a secret value transiently. SERBERUS's solution is to *assign each procedure its own data stack*, which is never reused by another procedure. Thus, a publicly-typed CA stack load will *never* transiently read a value from a different-offset or different-procedure CA stack store.

5.2.2. Implementation. Let F be a procedure with stack frame size k_F. Fig. 3 depicts the code transformation performed by the FPS Pass. Formally, given the input program $\mathcal{P}_{\text{fence}}$ with data stack DS_{in}, we define the output program $\mathcal{P}_{\text{fps}} = (\mathcal{M}_I^{\text{fps}}, \mathcal{M}_D^{\text{fps}}, P_{\text{fps}}, \mathbf{C}_0^{\text{fps}})$ with data stack DS_{fps}. To start, we initialize $(\mathcal{P}_{\text{fps}}, DS_{\text{fps}}) \leftarrow (\mathcal{P}_{\text{fence}}, DS_{\text{in}})$.

Private stack assignment. SERBERUS allocates and assigns a *private stack* to F, denoted as PS_F. To construct PS_F, we choose a stack base and stack end $B_F, E_F \in \mathcal{V}$ (where $E_F - B_F$ is a positive multiple of k_F) and set

$$PS_F \leftarrow \underbrace{[B_F, E_F)}_{\text{usable region}} \cup \underbrace{[E_F, E_F + k_F)}_{\text{underflow region}}$$

```
1  foo: ENDBR
2     + LD [ZR+PSP_F],SP // load private SP
3       SUB SP,SP,k_F    // frame allocation
4     + MAX SP,SP,B_F    // prevent overflow
5     + ST [ZR+PSP_F],SP // store private SP
6       ...
7       CALL r1
8     + LD [ZR+PSP_F],SP // load private SP
9       ...
10      ADD SP,SP,k_F     // frame deallocation
11    + MIN SP,SP,E_F    // prevent underflow
12    + ST [ZR+PSP_F],SP // store private SP
13      RET
```

Figure 3: Instructions inserted by SERBERUS's FPS Pass (indicated with "+"). Lines 2–5 are the prologue; lines 11–12 are the epilogue. `MAX`/`MIN` are instances of ASP's `OP` instruction (§3.4.6).

such that $PS_F \cap \mathcal{M}_D^{\text{fps}} = \emptyset$. We then map PS_F's usable region into data memory ($\mathcal{M}_D^{\text{fps}} \leftarrow \mathcal{M}_D^{\text{fps}} \cup [B_F, E_F)$) and zero-initialize it in all initial configurations $C_0 \in \mathbf{C}_0^{\text{fps}}$; however, we leave the underflow region unmapped, since well-formed procedures never sequentially underflow their stack. We also add the private stack PS_F to the program's stack metadata ($DS_{\text{fps}} \leftarrow DS_{\text{fps}} \cup PS_F$, Def. 4.4). Finally, SERBERUS allocates a global variable $\texttt{PSP}_F \in \mathcal{M}_D$ to hold the private stack pointer and initializes it to the stack end $D_0[\texttt{PSP}_F \leftarrow E_F]$ in all initial configurations $C_0 \in \mathbf{C}_0^{\text{fps}}$.

Switching private stacks. SERBERUS inserts instructions into the prologue and epilogue of F to ensure it uses its private stack PS_F rather than the shared stack. *Prologue*: F loads its private stack pointer $\texttt{PSP}_F$ (L2 in Fig. 3) before frame allocation (L3); a lower bounds clip (L4) prevents stack overflow; and F's updated stack pointer is saved (L5). *Post-call*: after each `CALL`, we switch from the callee's stack pointer back to F's stack pointer (L8). *Epilogue*: after frame deallocation (L10), we insert an upper bounds clip (L11) to restrict any subsequent transient stack underflows to the underflow region; and F restores its private stack pointer at procedure entry (L12). Since E_F–B_F is a multiple of k_F, the bounds clips produce mutually k_F-aligned stack pointers, preventing STKL taint primitives.

5.2.3. Guarantees. SERBERUS's FPS Pass produces a CTS program $\mathcal{P}_{\text{fps}}$ (Cor. 5.4.1) that has no SCT violations due to NCAS (Thm. 5.3) or STKL taint primitives (Thm. 5.5).

Theorem 5.4 (Private and aligned stacks)**.** At each CA stack access I_i in any trace of $\mathcal{P}_{\text{fps}}$, the stack pointer points

inside the current procedure F's private stack PS_F and is aligned to F's frame size k_F. Formally, $R_i(\texttt{SP}) \in PS_F$ and $R_i(\texttt{SP}) = E_F - m \cdot k_F$ for some $m \geq 0$. (*Proof.* See §A.4.)

Corollary 5.4.1. $\mathcal{P}_{\text{fps}}$ satisfies CTS.

Proof. Thm. 5.4 implies the stack pointer SP is always in-bounds of DS_{fps} and so $\mathcal{P}_{\text{fps}}$ satisfies WF.3. All other CTS properties trivially continue to hold for $\mathcal{P}_{\text{fps}}$. □

Theorem 5.5 (No STKL in $\mathcal{P}_{\text{fps}}$)**.** No taint primitives (§4.4.2) belong to STKL in any trace of $\mathcal{P}_{\text{fps}}$.

Proof. Suppose for contradiction some taint primitive $I_j \mapsto \texttt{LD}\,[\texttt{SP}{+}d], r$ satisfies STKL in a trace of $\mathcal{P}_{\text{fps}}$. I_j read from a prior secretly-typed same-address store $I_i \mapsto \texttt{ST}\,[r'_a{+}d'], r'$, which is neither a transient NCA store (Thm. 5.3), sequential NCA store (violates TYP.4), nor CA global store (WF.3).

Thus, I_i is a CA stack store. Let (eq. 1) $A = R_i(\texttt{SP}) + d' = R_j(\texttt{SP}) + d$ be the effective address of I_i and I_j. I_i and I_j must be in the same procedure F since they are both in-bounds accesses to the same private stack (Thm. 5.4). Also by Thm. 5.4, (eq. 2) $R_i(\texttt{SP}) = E_F - m' \cdot k_F$ and (eq. 3) $R_j(\texttt{SP}) = E_F - m \cdot k_F$ for some $m, m' \geq 0$. Eqs. 1–3 imply $d' - d = k_F \cdot (m' - m)$ and thus $d' - d = 0 \pmod{k_F}$. Since frame offsets are less than the frame size (WF.3), $d = d'$, so I_i and I_j access the same stack variable d. Thus d (TYP.3) and I_i (TYP.4) are publicly-typed like I_j. We conclude (r', i) is a security-type violation but $(r', i) \rightarrow^{e}_{\text{dep}} (r, j+1)$, thus I_j is not a taint primitive by Def. 4.7, a contradiction. □

5.3. SERBERUS's Register Cleaning Pass

SERBERUS's final pass, the Register Cleaning Pass, eliminates all NARG taint primitives from $\mathcal{P}_{\text{fps}}$ to produce the fully mitigated output program $\mathcal{P}_{\text{sct}}$. It inserts instructions to zero out all non-argument registers (defined by the calling convention $\mathcal{A}$, Def. 4.4) before each call and return. Formally, let I be a CALL/RET. If $I \mapsto \texttt{CALL}\,r$, set $\mathbf{r_{zero}} = \mathcal{R}_{\text{gpr}} \setminus \mathcal{A}(I) \setminus r$; if $I \mapsto \texttt{RET}$, set $\mathbf{r_{zero}} = \mathcal{R}_{\text{gpr}} \setminus \mathcal{A}(I)$. For each $r_{\text{zero}} \in \mathbf{r_{zero}}$, insert $\texttt{MOV}\,r_{\text{zero}}, 0$ directly before I.

Theorem 5.6 (No NARG in $\mathcal{P}_{\text{sct}}$)**.** No trace of $\mathcal{P}_{\text{sct}}$ features any instruction satisfying NARG.

Proof. We publicly zero all non-argument registers before each CALL/RET, so NARG is never satisfied. □

5.4. Proof of SERBERUS's Correctness

In §5.1–5.3, we proved that SERBERUS's three passes eliminate all NCAS, STKL, and NARG taint primitives. Now, we prove in Thm. 5.7 that the fully mitigated program $\mathcal{P}_{\text{sct}}$ has no NCAL taint primitives and is thus SCT (Def. 4.1), i.e., does not transiently leak secrets.

Theorem 5.7. $\mathcal{P}_{\text{sct}}$ satisfies speculative constant-time.

Proof. Suppose for contradiction $\mathcal{P}_{\text{sct}}$ is not SCT. By Cor. 4.1.1, a transmitter I_l with sensitive operand r_{xmit} is dynamic-dependent (§4.4.1) on an NCAL taint primitive (i.e., a transient NCA load, §4.4.2) in some trace e of $\mathcal{P}_{\text{sct}}$. Let $I_i \mapsto \texttt{LD}\,[r_a + d], r_{\text{ncal}}$ be the *most recent* transient NCA load to execute on which (r_{xmit}, l) is dynamic-dependent, and let F be the procedure containing I_i. That is, $(r_{\text{ncal}}, i+1) \rightarrow^{e}_{\text{dep}*} (r_{\text{xmit}}, l)$, and for all steps k with $i < k < l$, $I_k \mapsto \texttt{LD}\,[r'_a + d'], r' \implies (r', k+1) \not\rightarrow^{e}_{\text{dep}*} (r_{\text{xmit}}, l)$. Recall that we use a one-step delay $(r_{\text{ncal}}, i+1)$ to describe dependencies to/from the output of a load I_i (§4.4.1).

We will show that there exists a source-sink pair (I_i, I_j) $(i < j \leq l)$ which guarantees that I_i executed *sequentially* via Thm. 5.2, yielding a contradiction (since I_i is *transient* by assumption). There are *two cases* we consider: either (r_{xmit}, I_l) *is* or *is not* static-dependent on I_i.

If (r_{xmit}, I_l) *is* static-dependent on I_i (§5.1.2)—i.e., $(r_{\text{ncal}}, I_{i+1}) \rightarrow^{F}_{\text{dep}*} (r_{\text{xmit}}, I_l)$—then (I_i, I_l) is a NCAL-XMIT source-sink pair and we are done.

Otherwise, there is some dynamic dependency from I_i to I_l not mirrored in a static dependency:

$$(r_{\text{ncal}}, i{+}1) \rightarrow^{e}_{\text{dep}*} (r, j) \;\rightarrow^{e}_{\text{dep}} (r', k{+}1) \rightarrow^{e}_{\text{dep}*} (r_{\text{xmit}}, l)$$
$$(r_{\text{ncal}}, I_{i+1}) \rightarrow^{F}_{\text{dep}*} (r, I_j) \not\rightarrow^{F}_{\text{dep}} (r', I_{k+1}) \tag{1}$$

We will show that (I_i, I_j) forms a source-sink pair. Clearly, $(r, j) \rightarrow^{e}_{\text{dep}} (r', k+1)$ is not an intraprocedural dynamic register dependency (§4.4.1), or else $(r, I_j) \rightarrow^{F}_{\text{dep}} (r', I_{k+1})$ would hold. Thus, it must be a dynamic interprocedural register dependency or memory dependency, implying (a) $I_j \mapsto$ CALL/RET or (b)–(d) I_j is a store.

(a) Suppose $I_j \mapsto$ CALL/RET. The Register Cleaning Pass (§5.3) breaks all dependencies through *non*-arguments, so r must be an argument. Thus, (I_i, I_j) forms a NCAL-ARG source-sink pair and we are done.

(b) If I_j is an NCA store, I_k is a load. I_k must be a CA load, since we already assumed I_i was the most recent NCA load. By Thm. 5.3, I_j executed sequentially, thus so did I_i, contradicting that I_i is a transient NCA load.

(c) If I_j is a CA global store, then (I_i, I_j) is a NCAL-GLOB source-sink pair and we are done.

(d) Else, $I_j \mapsto \texttt{ST}\,[\texttt{SP}{+}d], r$ is a CA stack store. I_k is not an NCA load (by assumption, I_i is the most recent NCA load on which I_l depends) or CA global (WF.3). Thus, $I_k \mapsto \texttt{LD}\,[\texttt{SP}{+}d], r'$ is a same-offset CA stack load in F (Thm. 5.4), so $(r, I_j) \rightarrow^{F}_{\text{dep}} (r', I_k + 1) = (r', I_{k+1})$ by the definition of static stack dependencies (§4.4.1), contradicting Eq. (1). □

6. Hardening Software Against Spectre

We produce a code artifact LLSCT, an implementation of SERBERUS for LLVM 14. We empirically evaluate LLSCT on a suite of cryptographic primitives to assess its performance overhead. For comparison, we evaluate *three variants* of LLSCT alongside *two baseline mitigations* and an *insecure baseline* NONE. See §A.9 for additional LLVM-specific LLSCT implementation details.

Baseline Mitigations. Our baseline mitigations, F+RETP+SSBD and S+RETP+SSBD, layer Spectre mitigations discussed in §2.2.2. Tab. 1 compares their security

guarantees to SERBERUS. The LLSCT repository contains evaluations of two additional SLH-based mitigations based on BladeSLH [34] and UltimateSLH [35].

SERBERUS Variants. To justify our hardware model (§2.3.2), we implement three LLSCT variants: LLSCT (our proposal), LLSCT-PSF, and LLSCT-NOSTL.

LLSCT-PSF implements a SERBERUS extension that mitigates Spectre-PSF in software (§A.6.1). In general, PSF implicates *all transient loads* as taint primitives (§4.4.2), i.e., all loads may transiently return secrets, as long as the program has stored a secret since the last `LFENCE`. Thus, we expand SERBERUS's NCAL-XMIT and NCAL-ARG source-sink pairs (§5.1.3) to encompass *all* loads (not just NCA loads) as sources and omit the three other pair types. The FPS Pass (§5.2) is not needed, since with PSF all stack loads may transiently return secrets.

LLSCT-NOSTL implements a SERBERUS extension where STL is disabled, e.g., via SSBD [32]. LLSCT-NOSTL omits the FPS Pass, since its core motivation is mitigating Spectre-STL. It is replaced with a lighter-weight *Stack Initialization Pass*, which zero-initializes the newly-allocated stack frame in a procedure's prologue. A new kind of source-sink pair CALL-XMIT is added to the Fence Insertion Pass (§5.1), where sources are `CALL`s and sinks are transmitters that are dependent on CA stack loads. This pair captures that RSB can cause a procedure to execute with the wrong stack frame featuring mismatching security types.

Transmitters. LLSCT assumes five transmitters: conditional branches, indirect branches (except returns), loads, stores, and division. ASP already models the first four (§3.2); §A.6 shows how to extend ASP to capture `DIV`.

Compilation Setup. All mitigations compile code with LLVM with `-O3` optimizations. All LLSCT variants pass the `-mllvm -sct` flag to LLVM to enable relevant SERBERUS mitigation passes. We link all code using LLD [111], which supports the `-z retpolineplt` flag (required for baseline mitigations using RETPOLINE). When compiling with LLSCT variants, we disable the following compiler optimizations which violate CTS requirements (§4.3): `-mllvm -no-stack-slot-sharing`, `-mno-red-zone`, `-mllvm -no-argument-promotion`, `-fno-jump-tables`. See §A.8 for details.

Hardware Modes. Each mitigation assumes a specific hardware model to uphold its assumptions (§2.3.2), which we realize with a set of runtime flags called the *hardware mode*. We set the hardware mode before executing the program. NONE assumes mode $\text{HWNONE} = \emptyset$, i.e., it does not restrict behavior. F+RETP+SSBD and S+RETP+SSBD assume mode $\text{SSBD} = \{\texttt{ssbd}\}$ which disables STL. LLSCT assumes mode $\text{ASP} = \{\texttt{doitm}, \texttt{rrsbad}, \texttt{psfd}, \texttt{ibt}, \texttt{shstk}\}$ to ensure the processor refines ASP (§3). LLSCT-NOSTL assumes mode $\text{ASP-NOSTL} = \text{ASP} \cup \text{SSBD}$, which disables STL in ASP. LLSCT-PSF assumes mode $\text{ASP+PSF} = \text{ASP} \setminus \{\texttt{psfd}\}$, which enables PSF in ASP.

System and Workloads. We run all experiments on a 24-core Alder Lake[15] Intel® Core™ i9-12900KS processor with 640 KiB L1d, 768 KiB L1i, 14 MiB L2, and 30 MiB L3 caches and 128 GB RAM, running a fork of Linux v5.9.0 that adds usermode IBT and SHSTK support [112].

We evaluate crypto primitives from Libsodium (Salsa20, SHA-256) [16]; the verified CT crypto library HACL* (ChaCha20, Poly1305, ECDH Curve25519) [17]; and OpenSSL (SHA-256, ChaCha20, ECDH Curve25519) [15].

15. While Alder Lake implements weak CET-IBT, we expect comparable performance on Alder Lake N, which implements strong CET-IBT (§A.10).

7. Results

Fig. 4 compares the performance of our cryptographic benchmarks when mitigated with LLSCT, its two variants, and our two baseline mitigations. Performance is normalized to NONE. On average, LLSCT incurs *lower overhead* (21.3%) than both baseline mitigations (66.7%/24.9%) for F/S+RETP+SSBD). Fig. 4 also decomposes LLSCT's overhead into four components: Fence Insertion ("f," §5.1), Function-Private Stacks (§5.2), Register Cleaning (§5.3), and SERBERUS's ASP hardware mode (§6). Unsurprisingly, the overhead of fence insertion dominates overhead in all cases, whereas the overhead of FPS, register cleaning, and the ASP hardware model are negligible (and are thus unlabeled).

Large Buffers. Cryptographic code features arithmetic-heavy loops for common operations like encryption, decryption, hashing, and message authentication. Thus, loop performance is crucial for overall cryptographic code performance. LLSCT introduces *half the overhead* (7.1%) of the next best mitigation, S+RETP+SSBD (14.1%), for large buffer sizes (8 KB). This is because SERBERUS's loop-aware transient CFG construction (§5.1.1) prioritizes placing `LFENCE`s *outside* of loops in its Fence Insertion Pass; both baselines *require* placing mitigations *inside* of loops.

LLSCT Variants. Both LLSCT-NOSTL and LLSCT-PSF perform *much worse* than LLSCT, in the worst case (OpenSSL SHA-256 64B) incurring 599.7% and 646.3% overhead, respectively. Since neither variant uses FPS, they add new source-sink pairs (§6) which require inserting more fences, *tripling* (63.1%/65.8%) the overhead relative to LLSCT (21.3%). Clearly, FPS and SERBERUS's default selection of source-sink pairs are essential to LLSCT's efficiency. Secondly, there is no clear way to adapt SERBERUS to mitigate Spectre-PSF fully in software while maintaining SERBERUS's low overhead. We conclude that the ASP hardware model is the *best fit* for the SERBERUS approach, striking the ideal balance between enabling speculation primitives that can be efficiently mitigated in software (PHT, BTB, RSB, STL) and disabling those that cannot (PSF).

Baseline Mitigations. As expected, F+RETP+SSBD performs worse than S+RETP+SSBD and LLSCT overall, since inserting `LFENCE`s after every conditional branch is expensive [106]. However, S+RETP+SSBD *sometimes performs worse* than F+RETP+SSBD. For Libsodium's SHA-256 (8KB), S+RETP+SSBD's overhead is over 10%/35% more than F+RETP+SSBD's/LLSCT's. Fig. 4 shows that in this case, enabling SSBD on top of SLH+RETPOLINE (which exhibits 12.7% overhead) incurs *significant additional overhead* (26.1%). In contrast, enabling SSBD for the insecure

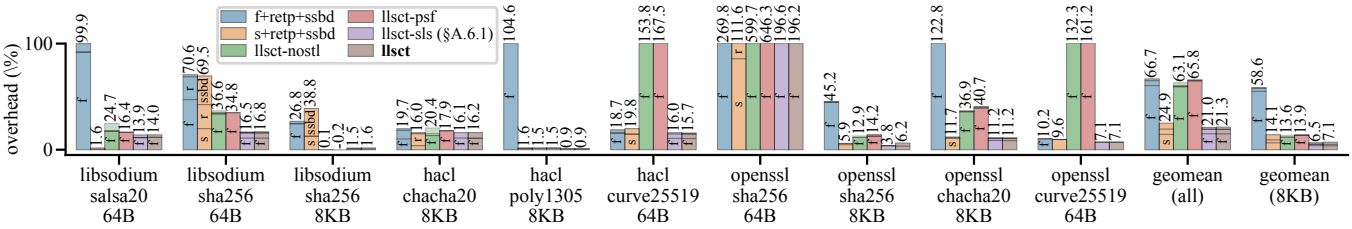

Figure 4: Runtime overheads of mitigations for crypto primitives in Libsodium, HACL*, and OpenSSL relative to code compiled with no mitigations. Segments within the same bar indicate the additional overhead incurred by layering on the mitigation component. We label components with $\geq 15\%$ overhead ("f" = LFENCE, "s" = SLH, "r" = RETPOLINE, "ssbd" = SSBD speculation control). Total percent overhead is at the top of each bar. Overheads $> 100\%$ are cut off.

baseline NONE incurs $< 1\%$ overhead. This difference is likely due to the complexity that SLH's masking operations introduce into the address calculation of stores; with SSBD, stores must wait for longer to perform.

Hardware Modes. The top segment of each bar in Fig. 4 indicates the overhead of the mitigation's hardware mode (§6) when layered on top of its software mitigations. The average overheads across all/8KB benchmarks for each are the following: 1.7%/0.9% for F+RETP+SSBD with SSBD; 5.5%/4.9% for S+RETP+SSBD with SSBD; 2.9%/1.6% for LLSCT-NOSTL with ASP-NOSTL; 0.8%/0.3% for LLSCT-PSF with ASP+PSF; and 1.9%/2.3% for LLSCT with ASP.

8. Related Work and Conclusions

Several works study detection, formal foundations, and mitigation of Spectre attacks [25], [113].

Software Detection. Symbolic execution [41], [42], [48], [102], [114], [115] is the most widely used technique to detect Spectre vulnerabilities in programs. However, existing detection tools do not scale well to large programs or to new speculation primitives. For example, none can detect Spectre-BTB vulnerabilities due to limitations of *always-mispredict* semantics [25], [41], [102]. Other approaches include *fuzzing* [116]–[118] or *static analysis* [119], [120].

Formal Foundations. Recent work deploys formal techniques to model and mitigate Spectre attacks [26], [34], [42], [101]–[103], [103], [104], [107]. Patrignani and Guarnieri [103] and Shivakumar et al. [107] both use formal models to demonstrate that LLVM's SLH mitigation is incomplete for Spectre-PHT. Fabian et al. [48] define an extensible framework for composing semantics for individual speculation primitives to model and detect Spectre leakage due to their combinations. Mosier et al. [120] and Ponce-de-León and Kinder [27] take an axiomatic approach inspired by memory consistency models to model and detect program instructions that may transiently access or leak secrets.

Software Mitigation. Few compiler-based Spectre mitigation proposals [25] are formally grounded [34] or protect against multiple Spectre variants [36], [121]. Some detection tools [105], [120] can mitigate detected Spectre vulnerabilities but do not evaluate the performance of the resulting program. None of these mitigations are readily deployable in an existing toolchain, in contrast with LLVM's LFENCE and SLH mitigations (§6).

Conclusions. We present SERBERUS, the first comprehensive mitigation for existing hardware that prevents all Spectre-PHT/BTB/RSB/STL/PSF leakage in CTS programs. We prove SERBERUS's correctness using our operational semantics, ASP, and implement it as a code artifact, LLSCT, in the LLVM compiler infrastructure. We evaluate LLSCT on a suite of cryptographic primitives from Libsodium, HACL*, and OpenSSL, and demonstrate significant performance and security improvements over the state-of-the-art.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) under award numbers CNS-2153936 and CAREER CCF-2236855; by a gift from Intel; and by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). We thank the anonymous reviewers for their valuable feedback during the review process. We also thank Carlos Rozas and Jason Brandt from Intel for clarifying the technical implementation details of Intel CET-IBT.

References

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

[2] ——, "The poly1305-aes message-authentication code," in *Fast Software Encryption*, H. Gilbert and H. Handschuh, Eds., 2005.

[3] 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*, D. H. Won and S. Kim, Eds., 2006.

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

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

[6] 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*, 2017.

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

[8] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *IEEE Symposium on Security and Privacy*, 2018.

[9] 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.

[10] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," in *Network and Distributed Systems Security*, 2018.

[11] 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.

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

[13] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *IEEE Cybersecurity Development*, 2017.

[14] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "SynthCT: Towards portable constant-time code," in *NDSS*, 2022.

[15] "OpenSSL: Cryptography and SSL/TLS toolkit," 2021, https://www.openssl.org/.

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

[17] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[18] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

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

[20] 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 *28th USENIX Security Symposium*, 2019.

[21] M. Miller, "Mitigating speculative execution side channel hardware vulnerabilities," 2018, https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[22] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy*, 2019.

[23] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," *12th USENIX Workshop on Offensive Technologies*, 2018.

[24] J. Horn, "Speculative execution, variant 4: Speculative store bypass," 2018, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[25] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *IEEE Symposium on Security and Privacy*, 2022.

[26] R. Guanciale, M. Balliu, and M. Dam, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[27] H. P. de León and J. Kinder, "Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022.

[28] 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.

[29] 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.

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

[31] Intel, "Branch History Injection and Intra-mode Branch Target Injection / CVE-2022-0001, CVE-2022-0002 / INTEL-SA-00598," 2022, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[32] Intel, "Speculative store bypass," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html, 2018.

[33] Intel, "Fast store forwarding predictor," 2022, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html.

[34] 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," *Proceedings of the ACM on Programming Languages*, 2021.

[35] 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*, 2023.

[36] 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*, 2021.

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

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

[39] 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*, 2019.

[40] 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.

[41] 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.

[42] 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.

[43] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[44] "Security analysis of amd predictive store forwarding," https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2021, accessed: 2022-10-18.

[45] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations." in *USENIX Security Symposium*, 2016.

[46] Intel, "Cpuid enumeration and architectural msrs," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/cpuid-enumeration-and-architectural-msrs.html, 2022.

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

[48] X. Fabian, M. Guarnieri, and M. Patrignani, "Automatic detection of speculative execution combinations," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[49] D. J. Bernstein, "Cache-timing attacks on aes," The University of Illinois at Chicago, Tech. Rep., 2005.

[50] 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.

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

[52] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," *2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[53] 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.

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

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

[56] 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.

[57] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*, 2006.

[58] 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*, 2018.

[59] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*, 2009.

[60] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," IACR'18, 2018.

[61] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, 2019.

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

[63] 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.

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

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

[66] 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*, 2021.

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

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

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

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

[71] 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*, 2019.

[72] 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*, 2019.

[73] "Mbed TLS," 2023, https://www.trustedfirmware.org/projects/mbed-tls/.

[74] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet *et al.*, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *IEEE Symposium on Security and Privacy*, 2020.

[75] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[76] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving c compiler," *Cryptology ePrint Archive*, 2019.

[77] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[78] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[79] 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)*, 2021.

[80] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[81] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," *27th USENIX Security Symposium*, 2018.

[82] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *IEEE Symposium on Security and Privacy*, 2020.

[83] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[84] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy*, 2019.

[85] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[86] "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, accessed: 2023-03-29.

[87] 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)*, 2018.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[94] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[95] 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 Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[96] 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 *ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020.

[97] 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)*, 2019.

[98] 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.

[99] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *CoRR*, vol. abs/1807.03757, 2018, http://arxiv.org/abs/1807.03757.

[100] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020.

[101] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, "High-assurance cryptography in the spectre era," in *IEEE Symposium on Security and Privacy*, 2021.

[102] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[103] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

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

[105] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[106] 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

[107] 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," in *IEEE Symposium on Security and Privacy*, 2023.

[108] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium*, 2022.

[109] Intel, "Retpoline: A branch target injection mitigation." https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html, 2018.

[110] K. Philips, A. Arcangeli, T. Chen, and L. Bulwahn, "Spectre side channels," 2023, https://www.kernel.org/doc/html/latest/_sources/admin-guide/hw-vuln/spectre.rst.txt.

[111] "Lld - the llvm linker," https://lld.llvm.org, accessed: 2022-10-17.

[112] Y. Yu, "linux_cet," https://github.com/yyu168/linux_cet.git, 2021.

[113] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Computing Surveys (CSUR)*, 2021.

[114] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

[115] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "Specusym: Speculative symbolic execution for cache timing leak detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[116] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium*, 2020.

[117] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[118] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *Computer Aided Verification*, 2020.

[119] 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*, 2019.

[120] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[121] Z. Shen, J. Zhou, D. Ojha, and J. Criswell, "Restricting control flow during speculative execution," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[122] ARM, "Straight-Line Speculation," 2020, https://developer.arm.com/documentation/102825/0100/?lang=en.

[123] M.-C. Costa, L. Létocart, and F. Roupin, "Minimal multicut and maximal integer multiflow: a survey," *European Journal of Operational Research*, vol. 162, no. 1, pp. 55–69, 2005.

[124] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, pp. 399–404, 1956.

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

Appendix A. Supplemental Material

A.1. Proof that CTS is a Strengthening of CT

Theorem A.1. If a program $\mathcal{P}$ for ASP satisfies CTS (Def. 4.3), then it also satisfies CT (Def. 4.2).

Proof. We prove the contrapositive. Suppose a *sequential* trace e contains a CT violation, i.e., it exposes a secretly-labeled sensitive operand r_{xmit} of some transmitter I_i via observation O_i. By CTS rule TYP.2, r_{xmit} must be publicly-typed at I_i (i.e., $\tau_{\text{reg}}^{F_i}(r_{\text{xmit}}, I_i) = \text{PUB}$). Thus, e violates the public type of r_{xmit} by sequentially assigning it a secret value, violating CTS rule TYP.1. Thus, $\mathcal{P}$ is not CTS. □

A.2. Proof of Theorem 5.1

Proof. Let $I_i \in F$, and let $I_j \in F$ ($i < j$) be the *next* executed instruction belonging to F in some trace e (i.e., for all l where $i<l<j$, $I_l \notin F$). We will show that $I_i = I_j$ or $I_j \in \text{tsuccs}(I_i)$ and thus $I_i \rightarrow^F_{\text{tcfg}*} I_j$. The claim follows due to the transitivity of $\rightarrow^F_{\text{tcfg}*}$.

If I_i halts execution (i.e., $C_i = C_{i+1}$) due to an LFENCE or trap, then clearly $I_j = I_i$, so we are done.

Now, suppose $I_i \mapsto \texttt{CALL} \mid \texttt{RET}$. Then $I_i \in \mathbf{I}_{\text{exit}}$ and the instruction that executed before I_j must have been a $I_{j-1} \mapsto \texttt{CALL} \mid \texttt{RET}$ instruction in order to have re-entered F at step j. Thus, I_j is a ENDBR or post-CALL instruction (due to ASP's control-flow restrictions, §3.4.2–3.4.3), and so $I_j \in \mathbf{I}_{\text{enter}}$. Thus, $I_j \in \text{tsuccs}(I_i)$, so we are done.

Else, the instruction executing after I_i is in the same procedure, or $I_{i+1} \in F$, so $I_{i+1} = I_j$. Since $I_{i+1} \in \text{succs}(I_i)$, $I_{i+1} = I_j \in \text{tsuccs}(I_i)$ by definition (§5.1.1). □

A.3. Proof of Theorem 5.2

Proof. Let (I_i, I_k) be a source-sink pair, and suppose for contradiction that I_i executed transiently. By Thm. 5.1, there is a path $I_i \rightarrow^F_{\text{tcfg}*} I_k$ in the T-CFG for F. Furthermore, since I_i did not trap, the path is non-trivial (i.e., for some $i \leq j < j' \leq k$, $I_j \rightarrow^F_{\text{tcfg}} I_{j'}$). SERBERUS's Fence Insertion Pass inserted an LFENCE along all paths in the T-CFG from I_i to I_k (§5.1.4), thus $I_i \rightarrow^F_{\text{tcfg}*} I_j \rightarrow^F_{\text{tcfg}*} I_k$ for some $I_j \mapsto \texttt{LFENCE}$ ($i < j < k$). Recall that LFENCEs halt transient execution (§3.4.5), so $I_j = I_k$. Thus, I_k is not a sink (i.e., the real sink did not execute), a contradiction. □

A.4. Proof of Theorem 5.4

Proof. It suffices to show that each private stack pointer (PSP) load $I_k \mapsto \texttt{LD}\,[\texttt{ZR}{+}\texttt{PSP}_F], \texttt{SP}$ inserted by the FPS Pass at procedure (re-)entrypoints (L2 or L8 in Fig. 3) returns an *in-bounds* and *aligned* value for the current procedure F.

We use the following predicate to capture whether SP is in-bounds and aligned at step i, where I_i is in procedure F (recall from §5.2 that B_F, E_F, and k_F denote F's stack base, stack end, and frame size, respectively):

$$Q(i) : \exists m \geq 0, B_F \leq R_i(\texttt{SP}) = E_F - m \cdot k_F.$$

We show that the result of each PSP load I_k satisfies $Q(k+1)$ by induction on the number of such loads.

Base case. If I_k is the first PSP load to execute, it executed in the prologue (L2) of the first procedure, before any stores have executed. Thus, I_k read from initial memory, which the FPS Pass initialized to $D_0(\texttt{PSP}_F) = E_F$. Therefore, $Q(k+1)$ is satisfied for $m = 0$.

Inductive step. Suppose that for all PSP loads I_i executing before I_k ($i < k$), $Q(i{+}1)$ is satisfied (i.e., they return an in-bounds and aligned PSP for their respective procedures).

If I_k read from initial memory, $Q(k+1)$ holds for the same reason as in the base case. Otherwise, I_k read from a prior same-address store I_j.

Suppose for contradiction $\boldsymbol{I_j}$ **is an NCA store**. The FPS Pass only inserts a PSP load I_k after a procedure (re)entrypoint, so some CALL/RET executed between I_j and I_k, forming a NCAS-CTRL source-sink pair. Thus, I_j executed sequentially (Thm. 5.2). This implies I_j in $\mathcal{P}_{\text{in}}$ sequentially wrote to global address $\texttt{PSP}_F$, which was unmapped prior to the FPS Pass, contradicting WF.5.

Suppose for contradiction $\boldsymbol{I_j}$ **is a CA stack store** in some procedure G. Since I_k is a CA global load, the PSP load I_i directly preceding I_j read an out-of-bounds PSP that pointed into global memory. So, $R_{i+1}(\texttt{SP}) \notin PS_G$, a contradiction of the inductive hypothesis that $Q(i{+}1)$ holds.

Thus, $\boldsymbol{I_j}$ **is a CA global store**. Furthermore, it is the PSP store $I_j \mapsto \texttt{ST}\,[\texttt{ZR} + \texttt{PSP}_F], \texttt{SP}$ (L5, L12), since only instructions inserted by the FPS Pass access PSP global variables which were newly allocated in $\mathcal{P}_{\text{fps}}$.

Let I_i be the PSP load that directly preceded I_j in F. If I_j is in the prologue, then $I_{j-3} = I_i$, I_{j-2}, I_{j-1}, and I_j correspond to L2–5 in Fig. 3. Applying the inductive hypothesis that $Q(i{+}1)$ holds, $R_{i+1}(\texttt{SP}) = R_{j-2}(\texttt{SP}) = E_F - m \cdot k_F$ for some $m{\geq}0$. After frame allocation I_{j-2}, $R_{j-1}(\texttt{SP}){=}E_F{-}(m{+}1){\cdot}k_F$. The bounds clip I_{j-1} ensures $B_F \leq R_j(\texttt{SP}){=}E_F - (m+d) \cdot k_F$ for a $d \in \{0,1\}$. Thus, $Q(j)$ holds. Since the PSP load I_k read from the PSP store I_j, $Q(k+1)$ holds as well, and we are done. If I_j is in the epilogue (L12), the argument is analogous. □

A.5. Additional Instruction Semantics

$$\textsc{Arithmetic Op}\quad I \mapsto \texttt{OP}_o\, r, \vec{r_s} \quad v_l = o_{\mathcal{L}}(R(r_{s,1}), \ldots, R(r_{s,n}))$$
$$R'_{\text{seq}} = R[\texttt{PC++}; r \leftarrow v_l] \quad C'_{\text{seq}} = C[R \leftarrow R'_{\text{seq}}]$$
$$\delta_{\text{seq}}(C,P) = (C'_{\text{seq}}, \varepsilon) \qquad \boldsymbol{\delta_{\text{t}}(C,P) = \emptyset}$$

$$\textsc{Uncond. Jump}\quad I \mapsto \texttt{JMP}\, d \quad R'_{\text{seq}} = R[\texttt{PC} \leftarrow R(\texttt{PC}) + 1 + d_{\text{PUB}}]$$
$$\delta_{\text{seq}}(C,P) = (C[R \leftarrow R'_{\text{seq}}], \varepsilon) \qquad \boldsymbol{\delta_{\text{t}}(C,P) = \emptyset}$$

A.6. Extending ASP and SERBERUS

A.6.1. Extending the Execution Model. SERBERUS reasons directly about taint primitives. Thus, when a new speculation primitive is discovered, one must determine if/how it changes the set of *taint primitives*.

Adding taint primitives. Consider straight-line speculation (SLS) [122], which adds the following *transient* transition to `CALL`/`RET`/`JMP`: $(C[\texttt{PC++}], O) \in \delta_{\text{t}}(C,P)$, where O is the exposed observation. One can prove that these transitions add a fifth taint primitive to CTS programs (§4.4.2): **LINE**: a transient `CALL`/`RET`/`JMP` that falls through to the next linear address (any register). SERBERUS can be extended to mitigate LINE with a new pass that inserts an `LFENCE` following each `JMP`. `CALL`s/`RET`s require no additional mitigations due to TYP.9 (§4.3.2) and Register Cleaning (§5.3). Fig. 4 evaluates this extension, LLSCT-SLS.

Replacing taint primitives. Consider PSF (§2.2.1), which redefines the *transient* transitions of `LD` as follows:

$$\textsc{Load (with PSF)}\quad \texttt{LD}\,[r_a + d], r_v \qquad A_l = R(r_a) + d_{\text{PUB}}$$
$$\mathbf{v}_{\text{t}} = \{D(A)\} \cup \{v_{l'} \mid \exists A', (A', v_{l'}) \in S\} \setminus v_{\text{seq}} \quad O = \texttt{ld}\, A_l$$
$$\boldsymbol{\delta_{\text{t}}(C,P) = \{(C[R \leftarrow R[\texttt{PC++}; r \leftarrow v_t];\, T \leftarrow \texttt{T}], O) \mid v_t \in \mathbf{v}_{\text{t}}\}}$$

One can prove that these new transitions replace NCAL, NCAS, and STKL (§4.4.2) with one new taint primitive: **LOAD**: a transient `LD` (output register). We evaluate an extension of SERBERUS which mitigates Spectre-PSF in software using the prior rule in §6 and §7.

A.6.2. Extending the Leakage Model. On many processors, `DIV` is a transmitter. To model this, we can add "`div` a, b" to ASP's observation set $\mathcal{O}$ (§3.2) and define the sequential transition for `DIV` r_a, r_b to expose the observation "`div` $R(r_a), R(r_b)$." Adding new observations does not change SERBERUS's correctness proof.

A.7. SERBERUS's Heuristic Directed Multicut

Computing the minimum directed multicut (i.e., the best global cut for multiple source-sink pairs) of a directed graph is NP-hard [123]. We approximate the optimal multicut by iteratively computing the optimal cuts for each source-sink pair individually (computed using Ford-Fulkerson [124]) while *fixing* the cuts of other source-sink pairs until convergence or loop. After that, we validate the cutset by verifying no source can reach its corresponding sink.

A.8. CTS-Unsafe Compiler Optimizations

When compiling with LLSCT, we disable these optimizations which violate CTS (§4.3). `-mllvm -no-stack-slot-sharing`: *Stack slot sharing* may assign two stack variables of different security types to the same frame index if their lifetimes do not overlap, violating CTS's stack typing requirements (§4.3.2). `-mno-red-zone`: A leaf procedure can use a limited amount of memory directly below the stack pointer (the "red zone") for its stack frame without needing to allocate it, violating WF.3 (§4.3.1). `-mllvm -no-argument-promotion`: LLVM may promote (possibly secret) pass-by-reference arguments to pass-by-value during interprocedural optimization, violating TYP.9 (§4.3.2). `-fno-jump-tables`: CTS procedures contain exactly one `ENDBR`, marking the procedure entrypoint (§4.3.1); jump tables require inserting `ENDBR`s elsewhere.

A.9. LLSCT Implementation Details

We implement Fence Insertion as a post-optimization IR pass, Function-Private Stacks as a post-register-allocation machine IR (MIR) pass that runs during frame lowering, and Register Cleaning as a post-register-allocation MIR pass that runs after call lowering.

It is safe to implement these passes at the identified points in the LLVM pipeline, given that LLVM upholds the following: (1) no NCA loads/stores are inserted when lowering IR to MIR or machine code (MC); and (2) MIR/MC optimizations are not allowed to reorder loads/stores past fences. The LLVM documentation [125] and our analysis of output machine code corroborate these assumptions.

A.10. Intel CET-IBT Implementations

Intel processors implement two variants of speculative semantics for CET-IBT: *strong* and *weak* (our terminology).

Strong CET-IBT completely blocks speculation following a missing `ENDBRANCH` instruction. Like prior work [36], ASP and SERBERUS assume a processor with strong CET-IBT. Intel processors implementing strong CET-IBT include Alder Lake N and Arizona Beach. Intel has shared with us that the long-term direction of CET-IBT implementations is towards these strong speculative semantics.

Weak CET-IBT allows some fixed, nonzero number of instructions to speculatively execute following a missing `ENDBRANCH`. Weak CET-IBT implementations are characterized by the number of instructions, and specifically loads, which may speculatively execute following a missing `ENDBRANCH`. *Older* weak CET-IBT implementations—found in Tiger Lake—allow up to 7 speculative instructions containing up to 5 speculative loads. *Newer* weak CET-IBT implementations—found in Alder Lake (our workstation), Sapphire Rapids, Raptor Lake, and some future processors—only allow up to 2 speculative instructions containing up to 1 speculative load. Our preliminary investigation shows that it is possible to restore SERBERUS's security guarantees on older and newer CET-IBT implementations with moderate and negligible runtime cost, respectively.

Appendix B. Meta-Review

B.1. Summary

The paper proposes SERBERUS, a set of compiler passes aimed at hardening cryptographic code against Spectre-type attack. The paper introduces a stronger notion of constant time programming, which code must adhere to, and relies on hardware support for preventing leaks of secret data from transient execution.

B.2. Scientific Contributions

- Creates a new tool to enable future science.
- Addresses a long-known issue.
- Provides a valuable step forward in an established field.

B.3. Reasons for Acceptance

1) The paper addresses a long-standing problem. It proposes a new approach for a solution. The solution is comprehensive, proven secure, and only incurs a modest performance overhead.

B.4. Noteworthy Concerns

1) The proposed execution model that underlies the security proof assumes mostly in-order execution. There is a gap in semantics between the model and out-of-order execution.
2) SERBERUS does not handle declassification.
3) SERBERUS relies on partially documented processor features and LLVM properties. SERBERUS is fragile due to possible future changes that break its assumptions.

Appendix C. Response to the Meta-Review

Response to Concern 1. While ASP has an in-order semantics like many prior speculative processor models [48], [102], [103], [107], it is intended to capture *all transient control- and data-flows* that may be exhibited by a program running on a speculative out-of-order processor. Specifically, we believe our transient load and store semantics capture all Spectre-relevant memory instruction reorderings that can occur on a speculative out-of-order processor.

Response to Concern 2. As presented, SERBERUS, like prior work [34], does not handle secret declassification in its formalization or proof. We expect that secret declassification can be achieved without compromising SERBERUS' guarantees by calling an auxiliary function that (1) copies a secret input buffer to a public output buffer and then (2) executes a speculation fence. Proving this would require changes to CTS and ASP, so we leave this to future work.

Response to Concern 3. Indeed, SERBERUS relies on processor and compiler properties that are not *guaranteed* to hold in the future. However, we believe academic work like SERBERUS is critical towards incentivizing hardware vendors and compiler writers to design processors and compilers that can support efficient Spectre mitigations.