SCHERI: Provably Secure Speculation Under the Constant-Time Policy for CHERI

Shixin Song
shixins@mit.edu
Massachusetts Institute of Technology
Cambridge, United States

Davide Davoli
davide.davoli@mpi-sp.org
MPI-SP
Bochum, Germany

Elias Storme
elias.storme@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Marton Bognar
marton.bognar@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Dominique Devriese
dominique.devriese@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Frank Piessens
frank.piessens@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Tamara Rezk
tamara.rezk@inria.fr
Inria
Sophia Antipolis, France

Abstract

Capability-based architectures such as CHERI provide strong support for the architectural isolation of software components. To additionally protect against microarchitectural leakage, software can be written in a constant-time fashion. Modern processors, however, rely heavily on speculative execution, which can invalidate the constant-time guarantees and leak isolated secrets transiently.

In this work, we show that providing secure speculation for CHERI is non-trivial, and that existing proposals fail to preserve the confidentiality guarantees. We develop a formal framework for reasoning jointly about capability safety, speculative execution, and information-flow security, and use it to demonstrate potential leaks. We then present *SCHERI*, a new processor design within this framework, and formally prove that it provides end-to-end secure speculation guarantees for the constant-time policy.

Our results provide formal foundations and practical guidance for building future capability-based processors, which are resilient to Spectre attacks for constant-time programs.

CCS Concepts

• **Security and privacy → Formal security models**.

Keywords

CHERI, formal execution models, Spectre, microarchitectural security, constant-time programming, information flow

ACM Reference Format:
Shixin Song, Davide Davoli, Elias Storme, Marton Bognar, Dominique Devriese, Frank Piessens, and Tamara Rezk. 2026. SCHERI: Provably Secure Speculation Under the Constant-Time Policy for CHERI. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846718

CCS '26, The Hague, Netherlands

1 Introduction

The ongoing prevalence of memory-safety vulnerabilities has motivated capability-based hardware architectures that deliver *enforceable guarantees by construction*. The CHERI (Capability Hardware Enhanced RISC Instructions) architecture emerged in 2010 [29] as a candidate for this vision, offering fine-grained memory protection and principled reasoning about pointer provenance. At the architectural level, CHERI enforces strong spatial memory-safety guarantees and enables robust compartmentalization [28]. The CHERI architecture has gained momentum over the last decade with the development of CHERI-RISC-V [24] and the ARM Morello project [1] that implements CHERI in ARM.

However, CHERI memory safety assumes a non-speculative execution model. Modern high-performance processors rely heavily on speculation, and, since the discovery of transient-execution attacks [20], it has become clear that speculation can violate architectural guarantees by exposing microarchitectural side effects. While misspeculated instructions are rolled back at the architectural level, their effects on microarchitectural state (e.g., caches) may persist, enabling attackers to leak secrets through side channels [19].

CSC. Prior work has demonstrated that CHERI is not immune to this problem [15]. In particular, Fuchs et al. [16] introduced the *Capability Speculation Contract* (CSC), which constrains speculative execution to preserve CHERI's capability invariants. They showed that violations of these invariants in speculative implementations (e.g., CHERI-Toooba) lead to concrete attacks such as Meltdown-CF, which break CHERI's memory-safety guarantees in practice. CSC can be understood as an instance of the broader class of hardware-software contracts [19] for secure speculation, which aim to provide principled co-design between hardware mechanisms and software reasoning. CSC restores speculative sandboxing by ensuring that speculative memory accesses respect capability permissions and bounds. While effective for preserving speculative sandboxing, CSC

does not provide confidentiality guarantees when legitimately accessible secrets can still leak through timing, cache, or other microarchitectural channels. We show that achieving provably secure speculation for the constant-time policy in CHERI requires moving beyond sandboxing, toward confidentiality guarantees.

BLACKOUT. An important step toward this goal was taken by BLACKOUT [12], which extends CHERI with hardware-supported taint tracking via *blinded capabilities* to enable data-oblivious computation. BLACKOUT raises faults when secret data influences control flow or memory accesses, thereby aiming to prevent both conventional and speculative side-channel leakage. At a high level, BLACKOUT moves beyond sandboxing toward non-interference-style guarantees by tracking the flow of secret data through computation. However, as we will show, BLACKOUT does not fully achieve secure speculation for the constant-time policy.

These limitations highlight a deeper issue: existing approaches lack a unified formal framework capable of expressing and comparing memory safety, sandboxing, and secure speculation guarantees.

Our contributions. In this paper, we address this gap by developing a formal framework for reasoning about CHERI-like architectures under speculative execution and strong leakage models. Within this framework, we:

- define a speculative semantics for CHERI-like processors with a strong attacker model and a notion of speculative constant time;
- show that the CSC does not guarantee confidentiality under leakage models stronger than speculative sandboxing;
- analyze BLACKOUT under these stronger leakage models, identifying sources of speculative information leakage;
- propose *SCHERI*, eliminating leaks while preserving practicality;
- formally prove that *SCHERI* satisfies end-to-end secure speculation guarantees in our model.

Our results provide both formal foundations and practical guidance for building CHERI processors resilient to speculative side-channel attacks. An extended version of this paper with full hardware semantics and proofs is available [26].

2 Background

CHERI. CHERI is a capability architecture that extends conventional ISAs with hardware-enforced memory protection. Instead of representing pointers as plain integers, CHERI uses *capabilities*: unforgeable references that carry both an address and metadata such as bounds, permissions, and validity information. Memory accesses through capabilities are checked in hardware, preventing out-of-bounds dereferences and unauthorized use of pointers. Capabilities can only be derived from existing capabilities, ensuring monotonicity: derived capabilities cannot gain permissions or exceed the bounds of their parent capability except at controlled security boundary crossings. These mechanisms provide strong spatial memory safety and support fine-grained compartmentalization. Practical realizations include CHERI-RISC-V [24], and ARM Morello [1]. While CHERI substantially strengthens architectural memory safety, these guarantees are defined at the ISA level and therefore assume architecturally correct execution. They do not automatically extend to (speculative) microarchitectural behavior.

Speculative execution and attacks. Modern processors improve performance through out-of-order and speculative execution. Instructions may execute before prior branches, permissions checks, or data dependencies are fully resolved. If speculation is later found incorrect, architectural state is rolled back, but transient effects on microarchitectural state, such as cache contents, predictor state, or execution timing, may remain observable.

Transient-execution attacks exploit this behavior to leak secrets via persistent microarchitectural effects. Spectre-style attacks [20] mistrain predictors to transiently execute attacker-chosen instruction sequences, Meltdown-style attacks [22] exploit incorrect forwarding from faulting or unauthorized operations. These attacks demonstrate that architectural safety alone is insufficient when secrets can influence transient execution.

For CHERI, this means that even capability-safe software may leak information if speculative execution bypasses intended checks or transiently exposes secret-dependent behavior.

Security properties: sandboxing and constant-time. Mitigations for speculative execution attacks and their desired security properties can be described using a *hardware-software contract* [19]. It is useful to distinguish two types of security properties:

- *Sandboxing* or *compartmentalization*, ensuring that a potentially malicious program cannot speculatively read outside of its architecturally designated sandbox, and
- *Constant-time* or *data-obliviousness*, ensuring that a benign program does not leak secrets through microarchitectural channels.

These are very different properties: sandboxing considers code that does not architecturally compute on secrets but tries to get access speculatively, whereas constant-time considers code that does architecturally compute on secrets, and should not leak them through side channels, even speculatively. Hence, they typically require different mitigations under speculation. CSC aims to prevent leakage in the sandboxing model, while BLACKOUT targets the speculative constant-time property.

3 Threat Model

We consider an attacker with the goal of inferring secret information from a CHERI system through microarchitectural side-channel observations. The attacker cannot directly read privileged state, break cryptographic primitives, or violate the architectural memory protection guarantees of CHERI, but they can execute untrusted code on the same processor (concurrently or across time-sharing contexts) and trigger victim execution with chosen public inputs. As is standard in the Spectre literature, the attacker may mistrain branch predictors, influence microarchitectural state to cause contention, and observe timing differences caused by victim execution.

We consider a broad class of microarchitectural side channels, including the state of microarchitectural structures (e.g., caches, branch predictors, the reorder buffer), speculation success and rollback behavior, and contention for execution ports and other timing-visible shared resources. This follows prior secure speculation work that treats attacker-observable microarchitectural state abstractly rather than committing to one specific channel [9]. Our threat model is intentionally stronger than the one captured by the Capability Speculation Contract (CSC): in addition to unauthorized speculative memory accesses, we consider leakage of legitimately

accessed secrets through timing-visible microarchitectural effects. We do *not* consider physical side channels such as power analysis, electromagnetic emanations, or fault injection.

Our goal is therefore to enforce relative speculative constant time for victim code (formalized in Theorem 6.2): secrets that are not leaked architecturally do not influence the attacker's observations. In particular, we seek guarantees stronger than CSC-style speculative sandboxing: speculative execution should preserve confidentiality, not only prevent capability forgery.

4 Prior Approaches and Shortcomings

The security property we aim to achieve is that programs that are constant-time in the absence of speculation remain constant-time under speculation. In this section, we show that existing Spectre mitigations on the CHERI architecture do not achieve this goal. We discuss two recently published approaches: *Capability Speculation Contracts* (CSC) [16] and BLACKOUT [12].

4.1 Capability Speculation Contract

CHERI provides architectural support for software sandboxing [29], and recently Fuchs et al. [16] proposed an approach to preserving these architectural sandboxing properties under speculation. Their *Capability Speculation Contract* (CSC) expresses that the hardware should not perform speculative violations of security boundaries enforced by capabilities. More precisely, it states:

Contract 4.1 (Capability Speculation Contract [16]). *All instruction and data-memory accesses issued in speculation must be authorized by capabilities either (1) in the committed register file; or (2) in memory transitively reachable through (1).*

Hence, CSC has a different goal than our design: it does not aim to prevent side-channel leaks of accessible secrets through speculation. It is straightforward to construct examples of programs that are secure in the absence of speculation and that comply with the CSC contract, but that leak secrets through side channels when speculation is allowed. Consider the following code:

```
1  f_load_sec:
2    cld a1, 0(csp) # secret data is loaded into a1
3    ccall g # after executing g, speculation starts
4  ...
5  exploit_gadget: # call to g speculatively returns here
6    cinoffset cs0, csp, a1 # cs0's offset advances by secret a1
7    cld a1, 0(cs0) # secret is leaked
```

Here, `csp` is the stack capability register, `cs0` is another capability register, and `a1` is a general purpose register. If secret data is stored on the stack referenced by `csp`, the `cld` instruction at line 2 loads it in `a1`. If the called `g` function speculatively returns to `exploit_gadget`, the secret is used to set the offset of `cs0`. Then, the load at line 7 uses the address of `cs0`, leaking the secret. Note that the secret could also be leaked through other side channels.

4.2 BLACKOUT

BLACKOUT [12] extends CHERI with *blinded capabilities* for enforcing data-oblivious execution, also during speculation. BLACKOUT incorporates the ideas of oblivious instruction set architectures [13, 31] into the CHERI architecture by repurposing an unused bit in capability metadata to mark a capability as *blinded*. When data is loaded through a blinded capability into a register, the register is also marked as blinded, and the hardware implements taint-tracking to propagate blindedness. Blinded values are treated as secrets, and the hardware prevents them from affecting timing behavior by faulting on e.g., load/store instructions with blinded addresses or branches with blinded conditions. During speculation, the faults are not raised architecturally, but they still prevent the leakage of blinded data. Building on these primitives, BLACKOUT allows programmers to write data-oblivious code, relying on the hardware to prevent side-channel leakage.

The security argument of BLACKOUT relies on the maintenance of five key invariants, but in order to efficiently support the call stack and register spills, BLACKOUT allows some of these invariants to be violated in a controlled way. One invariant (I3 [12]) says that the bounds of valid blinded and non-blinded capabilities must not simultaneously overlap. However, this complicates storing both blinded and non-blinded data on the stack. Hence, BLACKOUT allows this invariant to be violated for the stack capability: a local blinded variable will architecturally be accessed through a blinded capability that overlaps with the non-blinded stack capability. This is not a problem in the absence of speculation if the software stack can enforce that blinded data is only accessed through the blinded capability. However, this opens the door to speculative execution attacks that can load and leak blinded data through the non-blinded stack capability. Another invariant (I1) says that blinded data cannot be stored into memory using non-blinded capabilities. This is a problem for compiler-induced register spills: blinded registers cannot be spilled to the stack using the non-blinded stack capability. To address this, BLACKOUT repurposes the capability validity tag to indicate blindedness of secret register spills. Specifically, when the program spills a blinded register to the stack (using a `csc` instruction that stores the whole 16 bytes in the register), the BLACKOUT processor emits a special structure named *Blinded Register Record (BRR)*, which consists of the 8-byte spilled blinded value and an 8-byte marker (see Figure 1). BLACKOUT also sets the capability validity tag corresponding to the spill slot to 1. When restoring the blinded register spill from the stack using the `clc` instruction, the validity tag and the marker indicate that the restored value is a BRR rather than a capability or other public data on the stack, so the processor can properly mark the loaded data as blinded.

4.2.1 Security Analysis. At its core, BLACKOUT maintains its security guarantees as long as the hardware correctly tracks when secret data is accessed. Data can be marked as secret in two ways: either by setting the blinded bit in its associated capability, or, in the case of spilled registers on the stack, with a Blinded Register Record (BRR), which stores a magic constant right next to the spilled register value in memory. As long as the data is accessed through the blinded capability (in the case of blinded memory) or through a capability load through the stack capability (in the case of BRRs), the processor correctly sets and propagates the taint bit. In the following, we systematically explore how blinded data could be accessed in other ways that deviate from the above, leading to the loaded secret data incorrectly not being tainted, enabling leakage.

(1) Improperly accessing a BRR.
   (a) The blinded bit is only set when BRRs are loaded via full capability load instructions (`clc`). Accessing a spilled value,

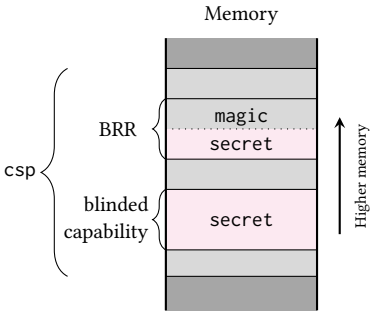

Figure 1: In BLACKOUT, memory referenced by blinded capabilities and Blinded Register Records (BRR) on the stack are considered secret (areas colored in). However, the non-blinded stack capability (csp) can also access this secret data.

potentially speculatively, through a narrower load instruction (e.g., `cld`) via the stack pointer (or any non-blinded capability) will not set the blinded bit.

(b) The blinded bit is only set when the BRR is tagged as a capability and its metadata corresponds to the magic constant. Even if the BRR is accessed via the stack pointer using a capability load, the blinded bit will not be set if the magic value has been corrupted or the capability tag cleared. This corruption could also happen transiently, e.g., as a result of incorrect store-to-load forwarding.

(2) Improperly accessing secret data through a non-blinded capability. In BLACKOUT, this is always possible for function-local blinded data, as the stack capability is not blinded (cf. Figure 1).

(a) Access through a load via the stack capability, bypassing compiler enforcement of exclusive access through the associated blinded capability. This can easily happen during speculation; e.g., if the stack capability has a different offset than the compiler expects.

(b) Stale data on the stack might be accessed by a newly derived capability in a later function call. Even if accesses to uninitialized memory are not allowed by the compiler, this could happen transiently if the store performing the initialization is speculatively skipped before a load.

(c) Blinded data could also be accessed through a non-blinded capability derived from the stack pointer during speculation. The CSC (see Section 4.1) does not curb this issue, as it allows all legal capability modifications to be performed speculatively as well.

4.2.2 Concrete Attacks. To provide evidence for the validity of the security concerns, we demonstrate proof-of-concept attacks (released publicly in Appendix A) exploiting Issues 1a and 2a on the BLACKOUT prototype. Issue 2b and Issue 2c can likely be exploited with additional engineering effort. The current prototype lacks the hardware speculation features required to exploit Issue 1b, but we believe it could be exploited on future implementations of BLACKOUT. Below, we briefly describe our exploits for Issues 1a and 2a, which rely on return address speculation [21]: the attacker causes a return instruction from a victim function to transiently jump to attacker code by poisoning the return stack buffer.

Exploit for Issue 1a. Consider the following code:

```
f_write_brr:
  # csp is the stack pointer capability
  # ca0 contains secret data
  csc ca0, -16(csp) # spill the secret to the stack as a BRR
  ccall g # after executing g, speculation starts
...
exploit_gadget: # call to g speculatively returns here
  cld a1, -16(csp) # secret data is loaded to a1 through csp
  cinoffset cs0, csp, a1 # cs0's offset advances by secret a1
  cld a1, 0(cs0) # secret is leaked
```

Here, the victim function spills a secret register (using the capability store instruction `csc`) onto the stack in a BRR. During speculation, this BRR is accessed via `cld`, a regular load instruction, which does not check whether the loaded data is a BRR, and as a result, does not set the blinded bit. Consequently, the loaded secret data in `a1` can be leaked by using it as an offset for a load.

Exploit for Issue 2a. This attack is structurally very similar to the previous one, and is exemplified below:

```
f_write_sec:
  # csp is the stack pointer capability
  # a0 contains secret data
  # cs0 is a blinded capability referencing [csp-8, csp)
  csd a0, 0(cs0) # store secret data to the stack through cs0
  ccall g # after executing g, speculation starts
...
exploit_gadget:  # call to g speculatively returns here
  cld a1, -8(csp) # secret data is loaded to a1 through csp
  cinoffset cs0, csp, a1 # cs0's offset advances by secret a1
  cld a1, 0(cs0) # secret is leaked
```

The main difference with the previous attack is that in this case, secret data is stored on the stack using a blinded capability, and the compiler enforces that this data is never accessed using the (non-blinded) stack capability during normal execution. However, this property can be broken during speculation, and the attacker can induce a transient load instruction that accesses the secret data on the stack using the stack pointer.

The leakage in CSC and BLACKOUT shows that reasoning jointly about capability safety, speculative execution, and information leakage is subtle and deserves rigorous formal treatment. Next, we present an alternative design addressing these issues, formally verified to provide strong security guarantees under speculation.

5 *SCHERI*

In this section, we introduce a formal execution model to study the security of CHERI-like architectures under speculative execution. To the best of our knowledge, our design is the first to formalize a taint-tracking mechanism where capabilities keep track of the security level of the data they reference. We treat the stack capability as always tainted and mark spills of *public* values using a distinguished encoding, which we call *Untainted Register Record* (URR). URRs set the capability tag and use a reserved marker value in the metadata field. When the capability tag is set, capability-sized loads consult the metadata field to distinguish data from capabilities: if the metadata matches the reserved marker, the value is interpreted as untainted data; otherwise, it is treated as a capability.

A key challenge of this design is that using a tainted stack capability would cause restored capabilities to be conservatively classified as secret, unnecessarily constraining speculation. This would cause a performance degradation, especially in constant-time software, where memory access patterns and control flow must be public.

To address this, we encode taint information explicitly in capability metadata. Concretely, when a capability c with address ℓ, metadata ℓ_{meta}, and taint status t is stored to memory, the pair of words $(\ell, \mathit{packTaint}(\ell_{\text{meta}}, t))$ is stored instead, where $\mathit{packTaint}(\cdot)$ is a function that embeds the taint bit into reserved metadata bits. Upon loading, the hardware reconstructs both the original metadata and the taint of the destination register.

Our design solves the issues of BLACKOUT identified in the previous section. Since the initial stack capability is tainted, it will remain tainted during execution and there will be no public aliasing capability addressing the same region if public data is always stored in URRs. This prevents speculative accesses to secrets on the stack from producing untainted values.

Throughout the remainder of this section, we illustrate *SCHERI*'s semantics and the rationale behind its security guarantees through a running example: we formalize the attack based on Issue 2a (Section 4.2) in *SCHERI*, and use it to contrast *SCHERI*'s design with alternative approaches. *SCHERI* is flexible enough to model realistic capability-specific speculation mechanisms, presented in Section 7.

5.1 Preliminaries

Capabilities. In *SCHERI*, memory is accessed via capabilities. In turn, capabilities are modeled as pairs $\mathsf{CapData} \ni c ::= (\ell, \ell_{\text{meta}})$, composed of an address $\ell \in \mathsf{Addr} \triangleq \mathbb{N}$, and metadata $\ell_{\text{meta}} = (p, b, e, t)$, where the permission $p \in \mathsf{Perm}$ constrains memory accesses using the pointer. Permissions[1] can either be O (no access), RO (read-only), RW (read-write) or RX (read+execute), with the order:

$$\text{O} \sqsubseteq \text{RO} \qquad \text{RO} \sqsubseteq \text{RW} \qquad \text{RO} \sqsubseteq \text{RX}$$

The addresses b and e express the range of memory $[b, e)$ that is allowed to be accessed by the capability; the bit $t \in \mathsf{Taint}$ is a *taint* indicating whether data in such range is considered secret (if $t = \mathsf{T}$) or not (if $t = \mathsf{U}$). Following the usual convention, we assume that Taint is a lattice where $\mathsf{U} \leq \mathsf{T}$. We use the shorthand $\mathsf{Meta} \triangleq \mathsf{Perm} \times \mathsf{Addr}^2 \times \mathsf{Taint}$ for the set of all capability metadata, and we write $\mathsf{CapData} \triangleq \mathsf{Addr} \times \mathsf{Meta}$ for the set of capability data.

Register files and memories. In *SCHERI*, *total register files* are modeled as functions $r : \mathsf{Reg} \to \mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag}$ that associate each register to:

- a sequence of *values*;
- a *taint*, which is equal to T if the register content is tainted and to U otherwise;
- a *capability tag*, which is equal to C when the register stores a capability, and to V, otherwise.

For simplicity, we leave the set Val of values abstract, and we assume $\mathbb{Z} \cup \mathbb{B} \cup \mathsf{Addr} \cup \mathsf{Meta} \cup \mathsf{Instr} \cup \mathsf{Taint} \cup \mathsf{Tag} \subseteq \mathsf{Val}$. Following most CHERI implementations, registers can hold multiple words, and their content is considered a *capability* only if the *capability tag* is C, i.e., registers holding values in CapData, but with tag set to V will not be considered capabilities.

In our semantics, due to out-of-order execution, the content of some registers may be unavailable. To model this, we introduce *partial register files* with type $r : \mathsf{Reg} \to (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\perp$. In such register files, a register can now also be mapped to $\perp$ when its value is not available.

[1]For simplicity, we only model CHERI permissions that are required for our formal analysis of speculative constant-time. In particular, we do not model sealing and unsealing.

$$\begin{array}{rcll} \mathsf{Exprs} \ni e & ::= & v \mid x \mid op(e^*) & \textit{Expression} \\ \mathsf{Instr} \ni \mathit{instr} & ::= & x \leftarrow e \mid \texttt{load}\ x, e, sz & \textit{Instruction} \\ & & \mid \texttt{store}\ x, e, sz \mid \texttt{jmp}\ e & \\ & & \mid \texttt{beqz}\ x, \ell & \end{array}$$

Figure 2: *SC*Asm syntax.

Data memories are modeled as maps from addresses $\ell \in \mathsf{Addr}$ to values. We assume that each address designates one word, i.e., one memory cell holds a single value $v \in \mathsf{Val}$. Analogously, *pc* increments by one word (corresponding to one instruction) per step. As other CHERI-like systems do, *SCHERI* keeps track of whether a certain memory region stores a capability in a dedicated *tag memory*, which we model with a mapping $m_{\mathsf{t}} : \mathsf{Addr} \to \mathsf{Tag}$. Recall that each capability $c \in \mathsf{CapData} = \mathsf{Addr} \times \mathsf{Meta}$ occupies 2 words, so $m_{\mathsf{t}}[k]$ stores the tag that indicates whether the words in the region $[2k, 2k+2)$ are considered a capability. Equivalently, whether the word at address ℓ is part of a capability is recorded in $m_{\mathsf{t}}[\lfloor \ell/2 \rfloor]$, which we abbreviate as $m_{\mathsf{t}}[\ell/2]$.

When dealing with memories, we will often use the notation $m[\ell \mapsto v]$, to indicate the memory m with address ℓ updated to v, and $m[[\ell, \ell+n) \mapsto \vec{v}]$ as a shorthand for $m[\ell \mapsto v_1][\ell+1 \mapsto v_2] \ldots [\ell+n-1 \mapsto v_n]$. Dually, the notation $m[\ell, \ell+n)$ denotes the tuple $(m(\ell), \ldots, m(\ell+n-1))$.

Running Example. The first step to model Issue 2a in *SCHERI* is to model the stack capability that is exploited by the attack, which we assume is stored in a dedicated register s:

$$r(s) \triangleq \big((42, (\text{RW}, 0, 256, \mathsf{T})), \mathsf{U}, \mathsf{C}\big). \tag{1}$$

Here r maps s to a 2-word value, which is a capability (due to tag C) with address 42, permissions RW, and bounds $[0, 256)$. Its outer taint is U, because the capability, seen as a pointer, is public. Its inner taint is instead T in our design, to avoid transient leaks of tainted data (as discussed at the start of this section)—as opposed to BLACKOUT, where it would be U.

5.2 Syntax

We model *SCHERI* using a simple ISA presented in Figure 2, called *SC*Asm, which extends μAsm [18]. An *expression* is either a value v, a register name x, or an application of an operator $op \in \mathsf{Op}$ to a sequence of expressions. Each operator has an associated arity, and we only consider expressions where the arity constraints of operators are respected. For the moment we leave the set of operators abstract; it will be further specified in Section 5.3. *Instructions* include assignments $x \leftarrow e$, which update x with the value obtained by evaluating e, and load instructions $\texttt{load}\ x, e, sz$, which evaluate e to a capability and use it to load sz consecutive memory words into x, represented as a tuple. For demonstration purposes, we only allow $sz \in \{1, 2\}$—i.e., memory operations over one or two words—but the model can be extended with arbitrary sizes. For simplicity, we assume that neither assignments nor load instructions can modify the value of a dedicated register *pc* which

holds the program counter capability. The instruction `store` x, e, sz evaluates e to a capability and uses it to store the value of x in memory. Finally, `jmp` e is an indirect jump that evaluates e to an executable capability used as the jump target, while `beqz` x, ℓ is a conditional branch that adds the offset ℓ to the program counter if the value of x is 0, and increments it by 1 otherwise.

Running Example. Attack 2a can be modeled by the following sequence of *SC*Asm instructions:

$$\mathtt{P} \triangleq \mathtt{store}\ x, t, 1; \mathtt{jmp}\ g; \quad \mathtt{Q} \triangleq \mathtt{load}\ y, s, 1; \mathtt{load}\ y, s \mathbin{✚} y, 1. \quad (2)$$

The program P models function `f_write_sec`: it writes a secret contained in register x to memory through the tainted capability in register t, and jumps to the instruction pointed to by register g via the instruction `jmp` g. The program Q models the leak gadget `exploit_gadget`. The first load instruction uses the stack capability in register s to load one word of data into register y. The second load instruction performs the secret-dependent memory access by loading one word of data from the capability that is obtained by summing the value of y to the stack pointer.

5.3 Expression Semantics

To ensure correct taint and resolution propagation, we assume that each operator is equipped with an interpretation $\overline{op}$ that is:

- *monotone with respect to taints*, i.e., the taint of the output is greater than or equal to that of the inputs;
- *eager with respect to* $\bot$, meaning that if any of the inputs is $\bot$, the output must also be $\bot$.

To enforce monotonicity with respect to capability manipulation, we assume that $\mathsf{Op} = \mathsf{OpArith} \uplus \mathsf{OpCap}$ consists of

- a set of *arithmetic* operators OpArith, modeling ordinary arithmetical operations and always returning values tagged with V;
- a set of *capability* operators OpCap, modeling capability operations, such as shrinking and address modifications. These operators are the only ones that can return capabilities, i.e., values in CapData with tag C.

We require that n-ary *arithmetic* operators $op \in \mathsf{OpArith}$ have interpretation $\overline{op} : (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})^n_\bot \to (\mathsf{Val}^* \times \mathsf{Taint} \times \{\mathsf{V}\})_\bot$, and since these operators do not output capabilities, we do not impose further constraints. To ensure that *capability* operators $op \in \mathsf{OpCap}$ are well-behaved, we assume that they have the interpretation $\overline{op} : (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\bot, (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\bot \to (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\bot$.

We also require:

(1) *Capability in first position:* the output has tag C iff only the first input has tag C.
(2) *Monotonicity with respect to capabilities*: if the output is a capability, then its permissions and bounds are no greater than those of the input capability.
(3) *Propagation of capability taint*: if the input capability has inner taint t, then the output capability must also have inner taint t.

Expression semantics. Given a *partial register file* r, the denotational semantics of expressions is a function $[\![\cdot]\!]_r : \mathsf{Exprs} \to (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\bot$ that computes the expression value together with its taint and capability tag, or $\bot$ if the value of any of the registers used in the expression is not available. Observe that when an expression evaluates to a capability, it carries two distinct taints: the *inner taint*, which describes the security level of the data referenced by the capability, and the *outer taint*, which expresses the security level of the capability itself as a value. The denotational semantics of expressions is defined by structural recursion:

$$[\![v]\!]_r \triangleq (v, \mathsf{U}, \mathsf{V}) \qquad [\![x]\!]_r \triangleq r(x) \qquad [\![op(e^*)]\!]_r \triangleq \overline{op}([\![e^*]\!]_r).$$

For convenience, we assume that the set of operators includes an operator ✚ for adding an integer offset to the pointer of a capability, with the following interpretation:

$$[\![e_1 \mathbin{✚} e_2]\!]_r \triangleq \begin{cases} \bot & \text{if } [\![e_1]\!]_r = \bot \vee [\![e_2]\!]_r = \bot \\ ((\ell + v, \ell_{\text{meta}}), t_1 \sqcup t_2, \mathsf{C}) & \text{if } [\![e_1]\!]_r = ((\ell, \ell_{\text{meta}}), t_1, \mathsf{C}) \\ & \wedge\, [\![e_2]\!]_r = (v, t_2, \mathsf{V}) \\ (0, \mathsf{U}, \mathsf{V}) & \text{otherwise.} \end{cases}$$

Running Example. We continue on Attack 2a, focusing on the capability $s \mathbin{✚} y$ used for the secret-dependent load. Here, s has outer taint U (see (1)) while y, loaded through the stack capability, is tainted (Section 5.5): $[\![y]\!]_r = (v, \mathsf{T}, \mathsf{V})$. Hence $[\![s \mathbin{✚} y]\!]_r = ((42 + v, (\textsc{rw}, 0, 256, \mathsf{T})), \mathsf{T}, \mathsf{C})$, whose outer taint is the supremum of the taints of y and s, i.e., $\mathsf{T} \sqcup \mathsf{U} = \mathsf{T}$. *SCHERI* therefore blocks the secret-dependent load under speculation. In BLACKOUT, by contrast, the inner taint of s would be U, giving $[\![y]\!]_r = (v, \mathsf{U}, \mathsf{V})$ and $[\![s \mathbin{✚} y]\!]_r = ((42 + v, (\textsc{rw}, 0, 256, \mathsf{U})), \mathsf{U}, \mathsf{C})$, so the load would not be prevented.

5.4 Architectural Semantics

In this section, we equip *SCHERI* with an architectural operational semantics, formalizing its behavior and, in particular, the interpretation of memory operations.

Configurations are tuples $((m_\mathsf{d}, m_\mathsf{t}), r)$, composed by a data memory, a tag memory and a register file, and ranged over by the meta-variable S. Following a common approach in modeling side-channel leaks [2, 4, 18, 19], the architectural semantics of *SCHERI* is defined as a small-step operational semantics, where transitions $S_0 \xrightarrow{o} S_1$ are labeled with observations o revealing the information that the program leaks to the adversaries. Observations are defined by the following BNF:

$$\mathsf{Obs} \ni o ::= () \mid \mathit{load}(c) \mid \mathit{store}(v, c) \mid \mathit{br}(c) \mid \mathit{br}(b)$$

where $c \in \mathsf{CapData}$, $b \in \mathbb{B}$, and $v \in (\mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag})_\bot$. The observation $()$ labels transitions that do not leak. The observations $\mathit{load}(c)$ and $\mathit{store}(v, c)$ are produced by load and store operations from and to capability c, respectively. In the $\mathit{store}(v, c)$ observation, v is equal to the stored value when the store targets the public region, and to $\bot$ otherwise. This ensures that the resulting trace reveals declassified values or values that are already public. The observation $\mathit{br}(c)$ is produced when an indirect jump to c executes, and $\mathit{br}(b)$ is produced by direct branches whose guard evaluates to b. The branch target does not need to be included in the observation because attackers can derive it from their knowledge of the victim program and the value of b.

The n-step transition relation is defined as follows:

$$\frac{}{S \xrightarrow{\epsilon}{}^0 S} \qquad \frac{S_0 \xrightarrow{o} S_1 \quad S_1 \xrightarrow{O}{}^n S_2}{S_0 \xrightarrow{o:O}{}^{n+1} S_2}$$

Architectural Semantics

$$\frac{\textit{readMemInst}(m, [\![pc]\!]_r) = (x \leftarrow e) \quad [\![e]\!]_r = (z,t,c) \quad x \neq pc}{(m,r) \xrightarrow{()} (m, r[pc \mapsto [\![pc \mathbin{\blacksquare} 1]\!]_r][x \mapsto [\![e]\!]_r])} \text{[ISA-ASSIGN]}$$

$$\frac{\textit{readMemInst}(m, [\![pc]\!]_r) = \texttt{jmp}\ e \quad ((\ell, \ell_{\text{meta}}), t, \mathsf{C}) = [\![e]\!]_r}{(m,r) \xrightarrow{br(\ell,\ell_{\text{meta}})} (m, r[pc \mapsto ((\ell, \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})])} \text{[ISA-JMP]}$$

$$\frac{\textit{readMemInst}(m, [\![pc]\!]_r) = \texttt{beqz}\ x, \ell' \quad (v, \_, \_) = [\![x]\!]_r \quad ((\ell, \ell_{\text{meta}}), t, \mathsf{C}) = (v = 0)\ ?\ [\![pc \mathbin{\blacksquare} \ell']\!]_r : [\![pc \mathbin{\blacksquare} 1]\!]_r}{(m,r) \xrightarrow{br(v=0)} (m, r[pc \mapsto ((\ell, \ell_{\text{meta}}), t, \mathsf{C})])} \text{[ISA-BEQZ]}$$

$$\frac{\textit{readMemInst}(m, [\![pc]\!]_r) = \texttt{load}\ x, e, sz \quad \textit{readMem}(m, [\![e]\!]_r, sz) = (v,t,c) \quad [\![e]\!]_r = ((\ell, \ell_{\text{meta}}), \_, \_)}{(m,r) \xrightarrow{load(\ell,\ell_{\text{meta}})} (m, r[pc \mapsto [\![pc \mathbin{\blacksquare} 1]\!]_r][x \mapsto (v,t,c)])} \text{[ISA-LOAD]}$$

$$\frac{\textit{writeMem}(m, [\![e]\!]_r, sz, [\![x]\!]_r) = m' \quad v = (t = \mathsf{U})\ ?\ [\![x]\!]_r : \bot \quad \textit{readMemInst}(m, [\![pc]\!]_r) = \texttt{store}\ x, e, sz \quad [\![e]\!]_r = ((\ell, (p,b,e,t)), \_, \_)}{(m,r) \xrightarrow{store(v,(\ell,(p,b,e,t)))} (m', r[pc \mapsto [\![pc \mathbin{\blacksquare} 1]\!]_r])} \text{[ISA-STORE]}$$

Core rules of memory operations

$$\frac{b \le l \quad l + sz \le e \quad p' \sqsubseteq p \quad sz = 2 \Rightarrow \ell \%2 = 0 \quad (p,b,e,t) \neq v_{\text{magic}}}{\textit{checkCap}((\ell, (p,b,e,t)), sz, p')} \text{[CHECK-CAP]}$$

$$\frac{m_{\mathsf{d}}(\ell) \in \text{Instr} \quad \textit{checkCap}((\ell, (p,b,e,\mathsf{U})), 1, \text{RX})}{\textit{readMemInst}((m_{\mathsf{d}}, m_{\mathsf{t}}), ((\ell, (p,b,e,\mathsf{U})), \mathsf{U}, \mathsf{C})) = m_{\mathsf{d}}(\ell)} \text{[READ-MEM-INST]}$$

$$\frac{\textit{checkCap}((\ell, (p,b,e,t_c)), sz, \text{RO}) \quad \textit{processRead}(m_{\mathsf{d}}[\ell, \ell + sz), t_c, m_{\mathsf{t}}[\ell/2], sz) = (v,t,c)}{\textit{readMem}((m_{\mathsf{d}}, m_{\mathsf{t}}), ((\ell, (p,b,e,t_c)), \_, \mathsf{C}), sz) = (v,t,c)} \text{[READ-MEM]}$$

$$\frac{\textit{checkCap}((\ell, (p,b,e,t_c)), sz, \text{RW}) \quad \textit{processWrite}((v,t,c), t_c, sz) = (v', c') \quad m' = (m_{\mathsf{d}}[[\ell, \ell + sz) \mapsto v'], m_{\mathsf{t}}[\ell/2 \mapsto c'])}{\textit{writeMem}((m_{\mathsf{d}}, m_{\mathsf{t}}), ((\ell, (p,b,e,t_c)), \_, \mathsf{C}), sz, (v,t,c)) = m'} \text{[WRITE-MEM]}$$

Figure 3: *SC*Asm architectural semantics and memory operation semantics.

Two ISA states S_0 and S_1 produce the same leakage (written $S_0 \equiv_{ISA} S_1$) when for every $n \in \mathbb{N}$ and observation $O \in \text{Obs}^*$, we have:

$$\exists S_0'. S_0 \xrightarrow{O}{}^{n} S_0' \iff \exists S_1'. S_1 \xrightarrow{O}{}^{n} S_1'.$$

The semantics is defined in Figure 3. Each rule evaluates the program counter in the register file r, and uses the resulting capability to fetch the current instruction. In turn, fetching is performed via the $\textit{readMemInst}(m, [\![pc]\!]_r)$ function, which is formally defined by Rule [READ-MEM-INST]. In this rule, the premise on the left ensures that the memory location pointed to by the program counter is an instruction, and the premise on the right validates the capability that is used to fetch the next instruction via the predicate $\textit{checkCap}(\cdot,\cdot,\cdot)$, defined by Rule [CHECK-CAP]. Essentially, this rule ensures that the capability offset is in bounds, and it has sufficient permissions. When the size of the load is 2, the rule also ensures that the accessed address is aligned. This requirement is necessary for correct handling of capability tags, as $m_{\mathsf{t}}(\ell/2)$ keeps track of the tag associated with the data stored in $m_{\mathsf{d}}[\ell, \ell+2)$ (see Rules [WRITE-MEM] and [READ-MEM]). Consequently, the alignment condition enforces consistency of that data and tag. Finally, the last premise ensures that the metadata field of the capability is not a dedicated value v_{magic}, which marks spills of untainted values; we discuss this mechanism in more detail when we describe the semantics of memory operations in Section 5.5.

We now turn to the semantics of instructions. Assignments $x \leftarrow e$ are evaluated via Rule [ISA-ASSIGN], which evaluates expression e and updates x accordingly. The pc register is updated to point to the next instruction, while its metadata, taint and capability tag are not changed. Rule [ISA-JMP] executes indirect jump instructions `jmp` e by updating the program counter with the jump target $[\![e]\!]_r$, which is leaked via the observation. Conditional branches `beqz` x, ℓ' are executed with Rule [ISA-BEQZ], which evaluates the branch condition x to a value v, and sums the offset ℓ' to the program counter if $v = 0$, and increments the program counter otherwise.

Load instructions `load` x, e, sz are evaluated via Rule [ISA-LOAD]. The load capability is obtained by evaluating $[\![e]\!]_r$. The destination register x is then updated with the ISA-level value produced by $\textit{readMem}(m, [\![e]\!]_r, sz)$, together with its associated taint and capability tag. The value of $\textit{readMem}(m, [\![e]\!]_r, sz)$ is provided by Rule [READ-MEM], which is also responsible for ensuring that the load capability has sufficient permissions, and for providing the correct ISA-level representation of unspills and loaded capabilities, given by the helper function $\textit{processRead}(\cdot,\cdot,\cdot,\cdot)$. Leaving the helper function unspecified allows our semantics to represent different memory designs. The one of *SCHERI* is described in detail in Section 5.5, the one of BLACKOUT in Section 7.2. Store instructions `store` x, e, sz are handled by Rule [ISA-STORE]. The store capability is obtained by evaluating $[\![e]\!]_r$, and the value is taken from register x. Memory is updated via $\textit{writeMem}(m, [\![e]\!]_r, sz, [\![x]\!]_r)$, whose value is provided by Rule [WRITE-MEM]. This rule is responsible for ensuring that the store capability has sufficient permissions to write and for computing the ISA-level representation (v', c') of the value and capability tag to store, given by the helper function $\textit{processWrite}(\cdot,\cdot,\cdot)$. Again, leaving this helper function unspecified allows our semantics to represent different designs. The updated memory is obtained by writing the value v' and tag c' at the addresses spanned by the store capability. Note that when the capability's inner taint is untainted, Rule [ISA-STORE] leaks the stored value to include declassified values in the leakage trace.

Running Example. We illustrate *SCHERI*'s architectural semantics on (2), assuming g points to a benign, skip-like target at ℓ' performing $w \leftarrow w$; this is *not* the Attack 2a behavior, where Q is reached via misspeculation (covered in Section 5.6). The architectural execution of $\mathsf{P} \triangleq$ `store` $x, t, 1$; `jmp` g is

$$\begin{aligned}(m,r) &\xrightarrow{store(\bot, [\![t]\!]_r)} (m', r[pc \mapsto [\![pc \mathbin{\blacksquare} 1]\!]_r]) \\ &\xrightarrow{br([\![g]\!]_r)} (m', r[pc \mapsto ((\ell', \ell'_{\text{meta}}), \mathsf{U}, \mathsf{C})]) \\ &\xrightarrow{()} (m', r[pc \mapsto (\ell' + 1, \ell'_{\text{meta}}, \mathsf{U}, \mathsf{C})]).\end{aligned}$$

The store executes via Rule [ISA-STORE]: since t has inner taint T, the leaked observation carries $\bot$ rather than the declassified value, while the secret $r(x) = (v, \mathsf{T}, \mathsf{V})$ is written to m, giving m'. The jump (Rule [ISA-JMP]) leaks the target $[\![g]\!]_r = ((\ell', \ell'_{\text{meta}}), \mathsf{U}, \mathsf{C})$ and transfers control to the benign ℓ'. Finally, $w \leftarrow w$ fires Rule [ISA-ASSIGN], producing the empty observation () and advancing the program counter.

5.5 Memory Operation Semantics

At a high level, our memory semantics combines three mechanisms:

(1) On reads, the capability used for the access determines the taint of the loaded value. Hence, values stored via a capability to untainted data are treated as untainted by any following load, and thereby declassified.
(2) 2-word-wide public values with high-order bits equal to zero are represented in memory by *Untainted Register Records* (URRs), i.e., pairs (v, v_{magic}) tagged as capabilities, where v_{magic} is a magic value that does not collide with any valid capability metadata. This allows URRs to be distinguished from genuine capabilities. Unlike BLACKOUT, where the BRR mechanism only applies to register spills, the URR mechanism applies to every untainted value store in *SCHERI*. To improve memory efficiency, future work could refine this mechanism.
(3) When a capability is written to memory, its *outer* taint is saved as part of its stored metadata and later recovered when the capability is read back. If this encoded taint is U, the outer taint is restored to U regardless of the taint of the loading capability. This mechanism avoids the performance degradation caused by conservatively tainting originally untainted capabilities.

The semantics of memory accesses is defined in Figure 4, via two helper functions that convert between the ISA-level representation of values and their in-memory representation. The function $\mathit{processWrite}((v, t, c), t_c, sz)$ takes as arguments the value (v, t, c) being stored—i.e., the ISA-level value v, together with its taint t and capability tag c—the inner taint t_c of the capability used to perform the store, and the store size sz; it returns the pair (v', c') to be written respectively to m_{d} and m_{t}. Dually, the function $\mathit{processRead}(v, t_c, c, sz)$ takes as arguments the raw value v read from m_{d}, the inner taint t_c of the capability used to perform the load, the tag c read from m_{t} at the corresponding address, and the load size sz; it returns the ISA-level value, taint, and capability tag.

For writes, operations of size $sz = 1$ are handled by Rule [WRITE-DATA], which tags them as non-capabilities, since capabilities have size 2. Capability writes are handled via Rule [WRITE-CAP], which encodes the taint bit into the metadata via the function $\mathit{packTaint}(\cdot)$ before storing it in memory. This way the capability's taint can be restored at load time (Rule [READ-CAP]) via function $\mathit{unpackTaint}(\cdot)$. For generality, we do not further specify these two functions: we only assume that $\mathit{unpackTaint}(\mathit{packTaint}(\cdot))$ is the identity. Finally, Rule [SPILL-VALUE] ensures that untainted values are spilled to memory using URRs, as previously discussed. The restriction to pairs whose second element is zero ensures no information loss.

For reads, Rule [READ-DATA] specifies that data of size 1, or data not tagged as a capability, is forwarded to the ISA level as a non-capability: its value is unchanged, and its taint is set to t_c, i.e., to the taint of the capability used for loading, rather than to any taint recorded in memory. If the loaded data is tagged as a capability and the access size is 2, then two cases arise, distinguished by the value v_m of the second memory word: if $v_m = v_{\text{magic}}$, i.e., the loaded data is a URR, Rule [UNSPILL-VALUE] returns the spilled value and marks it as untainted. Otherwise, Rule [READ-CAP] applies, and the value is reconstructed as a capability, using $\mathit{unpackTaint}(\cdot)$ to recover its metadata and stored taint from v_m. The two taints are combined via $\sqcap$, so the resulting capability is untainted if the stored taint is U—meaning that the capability was untainted before being stored—or if the accessing capability is untainted.

Running Example. We illustrate memory operations with two stores. First, when P stores the tainted x, Rule [WRITE-DATA] applies, storing the value as-is and tagged as a non-capability; when Q later loads it via `load` $y, s, 1$, the tag in m_{t} is V, so Rule [READ-DATA] returns $(v, \mathsf{T}, \mathsf{V})$—the observed taint is entirely determined by that of s. Second, if x holds a public 2-word value $(v, 0)$, it is spilled via Rule [SPILL-VALUE] as a URR: $\mathit{processWrite}(((v, 0), \mathsf{U}, \mathsf{V}), t, 2) = ((v, v_{\text{magic}}), \mathsf{C})$, i.e., tagged as a *capability* in m_{t} with second word v_{magic}, *independently* of the storing capability's taint. When the value is read back the tag is C and the second word is v_{magic}, so Rule [UNSPILL-VALUE] applies and the value is read back untainted, *independently* of the loading capability's taint.

5.6 Hardware Semantics

Reorder buffers. Following a common approach [4, 9, 19], out-of-order and speculative execution are modeled by using a *reorder buffer* (ROB): a partial map $\rho : \mathbb{N} \rightharpoonup \mathsf{PInstr}$, expressing the position of a partially evaluated instruction in the buffer. In turn, partially evaluated instructions are described by the following BNF:

$$\mathsf{SpecTag} \ni \tau ::= \varepsilon \mid (\ell, \ell_{\text{meta}}) \mid ((\ell, \ell_{\text{meta}}), sz, z)$$
$$\mathsf{PInstr} \ni \iota ::= x \leftarrow e@\tau \mid x \leftarrow (v, t, c)@\tau \mid \texttt{load}\ x, e, sz@\tau \mid \texttt{store}\ x, e, sz@\tau \mid \texttt{store}\ (v, t, c), (\ell, \ell_{\text{meta}}), sz@\tau$$

where $e \in \mathsf{Exprs}$ is an expression that has not yet been evaluated, $(v, t, c) \in \mathsf{Val}^* \times \mathsf{Taint} \times \mathsf{Tag}$ represents an evaluated expression for assignments, or a load/store value, and $(\ell, \ell_{\text{meta}}) \in \mathsf{CapData}$ is an evaluated capability for memory access. In order to resolve speculative choices, partially evaluated instructions are tagged with a speculation tag $\tau \in \mathsf{SpecTag}$. In turn, the speculation tag can be:

- the constant ε, if the partial execution of the corresponding instruction has so far made no unresolved speculative choice;
- the *pc* capability from which the branch instruction was fetched, in the case of control-flow speculation;
- a triple $((\ell, \ell_{\text{meta}}), sz, z) \in \mathsf{CapData} \times \{1, 2\} \times \mathbb{N}_\bot$, which is used to tag assignments resulting from loads, in the presence of speculative store-to-load forwarding. Precisely, $(\ell, \ell_{\text{meta}})$ is the load capability, sz is the load size, and z is the index in the ROB of the aliasing store that forwarded the data, or $\bot$ if the value is fetched from the memory.

Given a ROB ρ and an index i, the ROB $\rho \setminus i$ is pointwise identical to ρ, but is undefined on i. Similarly, the ROB $\rho|_i$ is pointwise identical to ρ, but is undefined at indices greater than or equal to i.

$$\frac{sz = 1}{processWrite((v,t,c),t_c,sz) = (v,\mathsf{V})}\ [\textsc{write-data}] \qquad \frac{packTaint(v_{\text{meta}},t) = v_m}{processWrite(((v,v_{\text{meta}}),t,\mathsf{C}),t_c,2) = ((v,v_m),\mathsf{C})}\ [\textsc{write-cap}]$$

$$\frac{(v'_m,c') = (v_m = 0 \wedge t = \mathsf{U})\ ?\ (v_{\text{magic}},\mathsf{C}) : (v_m,\mathsf{V})}{processWrite(((v,v_m),t,\mathsf{V}),t_c,2) = ((v,v'_m),c')}\ [\textsc{spill-value}]$$

$$\frac{sz = 1 \vee c = \mathsf{V}}{processRead(v,t_c,c,sz) = (v,t_c,\mathsf{V})}\ [\textsc{read-data}] \qquad \frac{v_m = v_{\text{magic}}}{processRead((v,v_m),t_c,\mathsf{C},2) = ((v,0),\mathsf{U},\mathsf{V})}\ [\textsc{unspill-value}]$$

$$\frac{v_m \neq v_{\text{magic}} \quad unpackTaint(v_m) = (v_{\text{meta}},t)}{processRead((v,v_m),t_c,\mathsf{C},2) = ((v,v_{\text{meta}}),t \sqcap t_c,\mathsf{C})}\ [\textsc{read-cap}]$$

Figure 4: *SCHERI*'s memory operation semantics.

More formally, their domains are:

$$\mathrm{dom}(\rho \setminus i) \triangleq \mathrm{dom}(\rho) \setminus \{i\},$$
$$\mathrm{dom}(\rho|_i) \triangleq \mathrm{dom}(\rho) \cap \{0,\ldots,i-1\},$$

and, for every j in the respective domain:

$$(\rho \setminus i)(j) \triangleq \rho(j), \qquad (\rho|_i)(j) \triangleq \rho(j).$$

Buffer application. Instructions in the ROB cannot be evaluated directly under the architectural register file, as it may be stale with respect to updates performed by earlier buffered instructions. Instead, evaluation must use a register file that reflects the updates of the preceding portion of the buffer. This is captured by the function $apl(\rho, r)$, defined as follows:

$$apl(\rho,r) \triangleq \begin{cases} r & \mathrm{dom}(\rho) = \emptyset \\ apl(\rho \setminus i, aplinst(\rho(i),r)) & i = \min \mathrm{dom}(\rho), \end{cases}$$

where:

$$aplinst(x \leftarrow (v,t,c)@\tau, r) \triangleq r[pc \mapsto [\![pc \mathbin{+} 1]\!]_r][x \mapsto (v,t,c)]$$
$$aplinst(x \leftarrow e@\tau, r) \triangleq r[pc \mapsto [\![pc \mathbin{+} 1]\!]_r][x \mapsto \bot]$$
$$aplinst(\texttt{load}\ x,e,sz@\tau, r) \triangleq r[pc \mapsto [\![pc \mathbin{+} 1]\!]_r][x \mapsto \bot]$$
$$aplinst(\texttt{store}\ x,e,sz@\tau, r) \triangleq r[pc \mapsto [\![pc \mathbin{+} 1]\!]_r].$$

This function applies the updates of ρ to r by processing instructions in program order, starting from the smallest index in the buffer. It also updates the program counter for each instruction. Note that the result of a buffer application in general is a *partial register file*. Using *partial* register files is also convenient to model how our design prevents the leak of tainted data during speculative execution. Following [9], this is done by evaluating instructions using exclusively untainted values during speculative execution. We model this via an *untainted projection*, $\cdot|_{\mathsf{U}}$, defined as follows:

$$w|_{\mathsf{U}} \triangleq \begin{cases} w & \text{if } w = (v,\mathsf{U},c) \text{ for some } v \in \mathrm{Val}^*, c \in \mathrm{Tag}, \\ \bot & \text{otherwise.} \end{cases}$$

The untainted projection is extended pointwise to register files by stipulating: $r|_{\mathsf{U}}(x) \triangleq r(x)|_{\mathsf{U}}$. The untainted projection of a register file can be used to sanitize speculatively tainted values, as follows:

$$aplsan(\rho,r) = \begin{cases} r & \text{if } \mathrm{dom}(\rho) = \emptyset \\ apl(\rho,r)|_{\mathsf{U}} & \text{otherwise.} \end{cases}$$

During out-of-order execution, when the value of an expression at the i-th entry of a ROB ρ might be leaked through side channels, that expression is evaluated with the register file $aplsan(\rho|_i, r)$. The case $\mathrm{dom}(\rho|_i) = \emptyset$ captures instructions at the very beginning of the ROB: such instructions are non-speculative, evaluated directly against the architectural register file r. All other instructions have at least one predecessor in the buffer, and are evaluated against $apl(\rho|_i, r)|_{\mathsf{U}}$, i.e., against the sanitized register file obtained by applying their predecessors' updates and discarding any value that is tainted. This prevents leaking tainted values during speculation.

For instance, take a ROB ρ where index 0 holds $x \leftarrow (secret, \mathsf{T}, c)$, and index 1 holds a subsequent instruction ι that references and leaks x. When ι is evaluated, it is evaluated in $aplsan(\rho|_1, r) = aplsan(\{0 \mapsto x \leftarrow (secret,\mathsf{T},c)\}, r)$. Since $\rho|_1$ is not empty, this reduces to $apl(\rho|_1, r)|_{\mathsf{U}}$: first, *apl* propagates the update to x; then, $\cdot|_{\mathsf{U}}$ maps the tainted value of x to $\bot$. Therefore, when ι is evaluated, x is unavailable, preventing its transient leak. Only once the first instruction retires, can ι access x's value.

Microarchitectural contexts. Having discussed how our hardware model supports speculative and out-of-order execution, we turn to the speculative semantics itself. Following [9, 19], we model the attacker by including in our configurations a *microarchitectural context*, which abstracts both the observations available to the adversary and its influence on execution.

Definition 5.1 (Microarchitectural context). A microarchitectural context is a structure $(\mathrm{Ctx}, update(\cdot,\cdot), predPc(\cdot), next(\cdot))$ with non-empty carrier Ctx, where:

(1) $update(\cdot,\cdot) : \mathrm{Ctx} \times A \to \mathrm{Ctx}$ returns a microarchitectural context that is obtained by updating the input context with information leaked during evaluation, taken from a set A which we do not further specify;
(2) $predPc(\cdot) : \mathrm{Ctx} \to \mathrm{Addr}$ takes the current microarchitectural context and returns a jump target (address) prediction;
(3) $next(\cdot) : \mathrm{Ctx} \to \mathrm{Dir}$ takes the current context and returns a directive from $\mathrm{Dir} ::= \mathrm{fetch} \mid \mathrm{exec}\ i \mid \mathrm{commit}$, where $i \in \mathbb{N}$.

The semantics interacts with the microarchitectural context through the operations $update(\cdot,\cdot)$, $predPc(\cdot)$ and $next(\cdot)$. First, all observable microarchitectural effects are reported via $update(\cdot,\cdot)$, ensuring that any information that may leak is reflected in the context. Second, speculative choices are generated using $predPc(\cdot)$, which is invoked to predict the target of jump instructions. Finally, the execution order is governed by $next(\cdot)$, whose output determines whether the processor fetches a new instruction (directive fetch), executes the i-th instruction in the ROB (directive exec i), or commits the oldest instruction (commit). In the following, we use a fixed arbitrary microarchitectural context. This ensures that all of our results hold independently of the specific adversary.

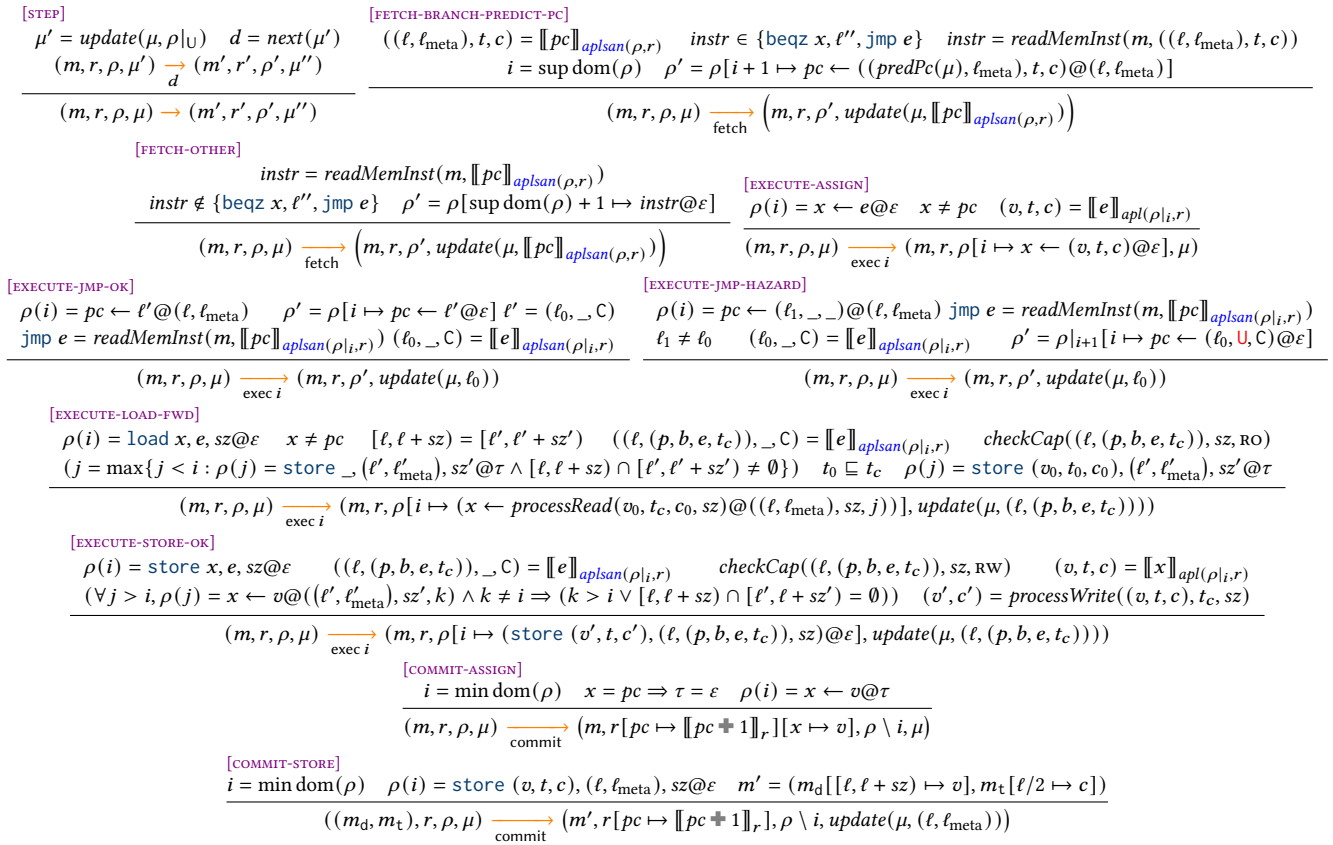

Figure 5: Hardware semantics of *SCHERI*, excerpt.

Hardware semantics. Hardware configurations are tuples of the form (m, r, ρ, μ), where m is a pair of a data and a tag memory, r is a register file, ρ is a ROB, and μ is a microarchitectural context. Our hardware-level semantics of *SCHERI* is a small-step operational semantics. An excerpt of the transition rules is in Figure 5. The complete set of rules is in [26]. The hardware semantics proceeds at two levels. Externally, the behavior is described by Rule [STEP]: at each step, the microarchitectural context is updated with the untainted projection of the ROB, ensuring that scheduling and prediction decisions depend only on untainted information. The context then produces a directive via $next(\cdot)$, which determines the next directive d, governing an internal transition.

If the directive is fetch, and the program counter points to a branch instruction ($instr \in \{\texttt{beqz}\ x, \ell'', \texttt{jmp}\ e\}$), Rule [FETCH-BRANCH-PREDICT-PC] applies. It uses $predPc(\cdot)$ to obtain a predicted target, and appends a speculative update of the program counter to the ROB by setting:

$$\rho' = \rho[i+1 \mapsto pc \leftarrow ((predPc(\mu), \ell_{\text{meta}}), t, c)@(\ell, \ell_{\text{meta}})].$$

Here, the branch instruction is tagged by its pc value, indicating the speculation on its target, which is resolved at execute time. Finally, the rule leaks the current program counter to the microarchitectural context μ via $update(\mu, [\![pc]\!]_{aplsan(\rho,r)})$. If the program counter points to a non-control-flow instruction, Rule [FETCH-OTHER] applies. It appends the fetched instruction to the ROB with an empty speculation tag, and leaks the current program counter to the context.

When the issued directive is exec i, different rules can be applied, depending on the i-th entry of the reorder buffer. Rule [EXECUTE-ASSIGN] evaluates assignment instructions $x \leftarrow e$ in the buffer by computing the value of e under the partially applied register file $apl(\rho|_i, r)$, and storing it in the buffer. Rule [EXECUTE-JMP-OK] resolves indirect jumps. The target e is evaluated under the sanitized register file $aplsan(\rho|_i, r)$. This ensures that the value of the jump target is not stale and that the leaked jump target depends only on untainted values. If the target resolves as correct, the speculative annotation is cleared and the resulting target is reported to the context. If instead the resolved target ℓ_0 does not match the predicted one, Rule [EXECUTE-JMP-HAZARD] applies: the corresponding entry of the ROB is updated with the correct target ℓ_0 (retagged as U), the speculation tag is cleared, and all younger instructions are dropped from the buffer. Conditional branches are executed analogously to indirect jumps by Rules [EXECUTE-BEQZ-OK] and [EXECUTE-BEQZ-HAZARD], whose details are in [26].

Rule [EXECUTE-LOAD-FWD] evaluates load instructions. It first checks that the target register is not the program counter, as this

can only be updated via `beqz` and `jmp` instructions, then it evaluates the load address to a capability $((\ell, \ell_{\text{meta}}), \_, \mathtt{C}) = [\![e]\!]_{aplsan(\rho|_i, r)}$. Using *aplsan* ensures that the address only depends on untainted values in speculative execution. The rule then checks whether the resulting capability has sufficient permissions for performing the load in the $checkCap((\ell, \ell_{\text{meta}}), sz, \text{RO})$ premise. Then, if an aliasing evaluated store exists at index $j < i$, the stored value is forwarded by setting the i-th entry in the ROB to an assignment of that value to x. Note that this forwarding is a form of SSB speculation, as the ROB may contain a not-yet-evaluated aliasing store at some index j' with $j < j' < i$. Thus, the speculation tag of the assignment carries the load capability $(\ell, \ell_{\text{meta}})$, the load size sz, and the index j in the ROB of the store instruction that forwarded its value.

Rule [EXECUTE-STORE-OK] executes store instructions and resolves store-to-load dependency speculation. As for the load rule, the store address is first evaluated under the sanitized register file to ensure that no tainted values are leaked during speculative execution. The resulting capability is then checked for sufficient permissions via $checkCap(\cdot, sz, \text{RW})$. The value to be stored is obtained by evaluating x under the partially applied register file $apl(\rho|_i, r)$. The value to be written is then computed via $processWrite(\cdot)$, the corresponding entry in the ROB is updated, and the accessed address is leaked to the microarchitectural context. Crucially, the rule additionally enforces consistency with prior store-to-load forwarding. Recall that Rule [EXECUTE-LOAD-FWD] may speculatively forward a value from an older store to a younger load. To ensure that such forwarding is sound, the store rule checks that no younger load has speculated on a conflicting store. This is captured by the side condition on indices $j > i$: if a younger load has been resolved by forwarding from a store at position $k \neq i$, then either that store is younger than the current one ($k > i$), or the accessed memory regions do not overlap (otherwise, the pipeline is flushed).

Rule [COMMIT-ASSIGN] commits assignments by updating the architectural register file with the computed value and removing the instruction from the buffer. This rule explicitly checks that if the target register is *pc*, the speculation tag has to be ε, to ensure that speculative jumps have been validated at execution time. Rule [COMMIT-STORE] commits store instructions by updating the data and tag memories, leaking the accessed address to the context a second time and removing the instruction from the buffer.

Running example. In Figure 6, we illustrate the execution in *SCHERI* of the vulnerable program (2), modeling Attack 2a. First, the store `store` $x, t, 1$ is fetched into entry 1 by Rule [FETCH-OTHER] and executed by Rule [EXECUTE-STORE-OK], resolving its store capability to (z, z_{meta}) and value to $(v, \mathsf{T}, \mathsf{V})$ without yet updating memory. Then, the jump `jmp` g is fetched by Rule [FETCH-BRANCH-PREDICT-PC]. Recall that we assume g points to a benign, skip-like target at ℓ' performing $w \leftarrow w$. Under the correct prediction (top), the predicted target is $predPc(\mu) = \ell'$, so entry 2 is speculatively set to $pc \leftarrow ((\ell', \ell_{\text{meta}}), \mathsf{U}, \mathtt{C})$, tagged with the branch's *pc* value $(\ell, \ell_{\text{meta}})$. Next, the benign $w \leftarrow w$ is fetched into entry 3 by Rule [FETCH-OTHER]. Finally, exec 2 resolves the jump (Rule [EXECUTE-JMP-OK]): the evaluated target matches the prediction, so entry 2's speculation tag is cleared, and the run reproduces the architectural execution.

Under the attacker's misprediction $predPc(\mu) = \mathtt{Q}$ (bottom), the store and jump execute as above except that entry 2 records the mispredicted target, steering speculation into the gadget `Q`. The first load `load` $y, s, 1$ is fetched into entry 3 by Rule [FETCH-OTHER] and executed by Rule [EXECUTE-LOAD-FWD]. The load capability is resolved to (z, z_{meta}), aliasing the store capability at entry 1, so the store value is forwarded to the load, yielding $y \leftarrow (v, \mathsf{T}, \mathsf{V})$. The instruction `load` $y, s \mathbin{+} y, 1$ is fetched into entry 4 by Rule [FETCH-OTHER], but exec 4 is *disabled*: its address evaluates to $[\![s \mathbin{+} y]\!]_{aplsan(\rho|_4, r)} = \bot$ because *aplsan* drops the tainted y.

6 Correctness and Security of *SCHERI*

In this section, we establish the two main properties of *SCHERI*, its correctness and security. *Correctness* ensures that every evaluation that is performed at the hardware level can be simulated at the ISA level. *Security* states that *SCHERI* is *relative speculative constant time*. Intuitively, this property guarantees that if a program does not leak secret data at the ISA level, it also does not leak secret data under the hardware semantics of *SCHERI*. For brevity's sake, proofs and formal definitions are deferred to [26].

Correctness of SCHERI. The following result establishes that *SCHERI* is correct with respect to its ISA-level semantics.

Theorem 6.1 (Functional Correctness). *Let $S = (m, r)$ and $C = (m, r, \rho, \mu)$ be initial ISA and HW configurations, respectively. For every $n \in \mathbb{N}$, and configuration $C' = (m', r', \rho', \mu')$ such that $C \rightarrow^n C'$, there exist $t \in \mathbb{N}$ and $O \in \text{Obs}^*$ such that $S \xrightarrow{O}{}^t (m', r')$.*

The proof of Theorem 6.1 is in [26], and is carried out by induction on the length of the execution trace.

Security of SCHERI. Our main result shows that the hardware semantics preserves ISA-level security guarantees in the presence of speculation. Formally, we require that hardware configurations that contain the same public data and produce the same ISA-level leakage also produce the same leakage at the hardware level. We first define two ISA states (m_0, r_0) and (m_1, r_1) containing the same public values, written $(m_0, r_0) \simeq_{pub} (m_1, r_1)$: untainted values coincide in the two states. Next, we define how two hardware configurations can leak the same data. Since the hardware-level semantics updates the microarchitectural contexts at each transition with leaked values, it suffices to require the identity of the target microarchitectural contexts at each transition step. More formally, we say that two hardware states $C_0 = (m_0, r_0, \rho_0, \mu), C_1 = (m_1, r_1, \rho_1, \mu)$ produce the same leakage traces (written $C_0 \equiv_{HW} C_1$) whenever for every $n \in \mathbb{N}$ and adversarial context μ', we have

$$\exists m_0', r_0', \rho_0'. C_0 \rightarrow^n (m_0', r_0', \rho_0', \mu') \;\Leftrightarrow\; \exists m_1', r_1', \rho_1'. C_1 \rightarrow^n (m_1', r_1', \rho_1', \mu').$$

Note that this condition imposes the identity of the target microarchitectural states that are updated at each step with microarchitectural leakage. With these definitions at hand, security of *SCHERI* can now be stated as follows:

Theorem 6.2 (Relative Speculative Constant Time). *Let*

$$S_0 = (m_0, r_0) \qquad C_0 = (m_0, r_0, \rho_{init}, \mu)$$
$$S_0' = (m_0', r_0') \qquad C_0' = (m_0', r_0', \rho_{init}, \mu)$$

be initial ISA and HW configurations where $\text{dom}(\rho_{init}) = \emptyset$. If $S_0 \simeq_{pub} S_0'$ and $S_0 \equiv_{ISA} S_0'$, then we have $C_0 \equiv_{HW} C_0'$.

$$(\emptyset, \mu_0) \xrightarrow[\text{fetch}]{} (\{1 \mapsto \texttt{store}\ x, t, 1@\varepsilon\}, \mu_1) \xrightarrow[\text{exec 1}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon\}, \mu_2)$$
$$\xrightarrow[\text{fetch}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\ell', \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}})\}, \mu_3)$$
$$\xrightarrow[\text{fetch}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\ell', \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}}),\ 3 \mapsto w \leftarrow w@\varepsilon\}, \mu_4)$$
$$\xrightarrow[\text{exec 2}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\ell', \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@\varepsilon,\ 3 \mapsto w \leftarrow w@\varepsilon\}, \mu_5)$$

$$(\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\mathsf{Q}, \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}})\}, \nu_3)$$
$$\xrightarrow[\text{fetch}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\mathsf{Q}, \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}}),\ 3 \mapsto \texttt{load}\ y, s, 1@\varepsilon\}, \nu_4)$$
$$\xrightarrow[\text{exec 3}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\mathsf{Q}, \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}}),\ 3 \mapsto y \leftarrow (v, \mathsf{T}, \mathsf{V})@((z, z_{\text{meta}}), 1, 1)\}, \nu_5)$$
$$\xrightarrow[\text{fetch}]{} (\{1 \mapsto \texttt{store}\ (v, \mathsf{T}, \mathsf{V}), (z, z_{\text{meta}}), 1@\varepsilon,\ 2 \mapsto pc \leftarrow ((\mathsf{Q}, \ell_{\text{meta}}), \mathsf{U}, \mathsf{C})@(\ell, \ell_{\text{meta}}),\ 3 \mapsto y \leftarrow (v, \mathsf{T}, \mathsf{V})@((z, z_{\text{meta}}), 1, 1),\ 4 \mapsto \texttt{load}\ y, s \blacksquare y, 1@\varepsilon\}, \nu_6)$$

Figure 6: Two hardware executions of (2) from the same initial configuration: execution under a correct prediction (top), which matches the architectural execution of Section 5.4; execution under the attacker's misprediction reaches Q (bottom).

The proof is carried out by induction on the length of the execution trace and relies on key invariants. The main one is the preservation of public equivalence at the hardware level, established via the functional correctness of *SCHERI* (Theorem 6.1), which helps relate leakage at the ISA level and the HW level. Another is the absence of memory regions referenced simultaneously by tainted and untainted capabilities, which prevents using untainted capabilities to access and leak tainted data.

7 Comparison with Other Designs

In this section, we compare *SCHERI* with different designs. To this end, we start in Section 7.1 by modeling CHERI-Toooba's design issues outlined in [14–16], and showing that the resulting system does not satisfy the CSC contract. We then model BLACKOUT [12] formally in Section 7.2, and show that the resulting system can leak secrets through microarchitectural side channels.

7.1 Modeling CHERI-Toooba's CSC Violations

Fuchs et al. [14–16] describe several violations of the CSC contract in CHERI-Toooba. These vulnerabilities fall into two categories: violations caused by control-flow speculation crossing the program counter capability boundaries, and violations arising from capability manipulation, where a modified capability can be used transiently before the corresponding monotonicity checks are enforced. In both cases, transient execution may perform memory accesses that violate capability bounds or permissions. In the following, we demonstrate that our formal semantics can capture these classes of vulnerabilities with only minimal modifications.

Speculative control-flow violation. This vulnerability [15] occurs when the branch target predictor predicts a jump target that is not allowed by the current program counter capability, violating the CSC contract. Such a vulnerability can be modeled by modifying Rule [FETCH-BRANCH-PREDICT-PC] as follows:

[FETCH-BRANCH-PREDICT-PC-CAP]

$$\frac{\begin{array}{l}((\ell, \ell_{\text{meta}}), t, c) = [\![pc]\!]_{aplsan(\rho,r)} \qquad instr \in \{\texttt{beqz}\ x, \ell'', \texttt{jmp}\ e\} \\ instr = readMemInst(m, ((\ell, \ell_{\text{meta}}), t, c)) \qquad i = \sup \mathrm{dom}(\rho) \\ \rho' = \rho[i+1 \mapsto pc \leftarrow \boxed{predPcCap(\mu)}\ @(\ell, \ell_{\text{meta}})]\end{array}}{(m, r, \rho, \mu) \xrightarrow[\text{fetch}]{} \left(m, r, \rho', update(\mu, [\![pc]\!]_{aplsan(\rho,r)})\right)}$$

The main difference between this rule and Rule [FETCH-BRANCH-PREDICT-PC] lies in the $predPcCap(\mu)$ function highlighted in red: while in our system, the function predicts the offset in the *pc* register, here it can predict the whole *pc* capability, potentially violating the architectural boundaries of the *pc*, and thereby the CSC contract (and also our security property).

Speculative capability manipulation. This class of vulnerability [14, 16] is caused by race conditions in the hardware between capability usage and validation that can occur with operations modifying capability offsets or, more generally, constructing capabilities from existing ones. Specifically, the constructed capability can be used during transient execution before the hardware ensures that such a capability has smaller range and permissions than an existing one, thus violating the CSC contract. To model these vulnerabilities, we extend the language of expressions with an additional operator $buildCap(\cdot, \cdot)$ corresponding to CHERI's `cbuildcap` instruction. It has the following interpretation:

$$[\![buildCap(e_1, e_2)]\!]_r \triangleq \begin{cases} \bot & \text{if } [\![e_1]\!]_r = \bot \vee [\![e_2]\!]_r = \bot \\ (v_2, t_1 \sqcup t_2, \mathsf{C}) & \text{if } [\![e_1]\!]_r = (v_1, t_1, \mathsf{C}) \\ & \wedge [\![e_2]\!]_r = (v_2, t_2, \mathsf{V}) \\ & \wedge v_2 \sqsubseteq v_1 \\ (0, \mathsf{U}, \mathsf{V}) & \text{otherwise.} \end{cases}$$

The primitive satisfies the axioms for capability operators in Section 5.3. First, the interpretation above satisfies Axiom (1) by construction. Second, Axioms (2) and (3) are satisfied given the following definition of $v_2 \sqsubseteq v_1$, for $v_1, v_2 \in \mathrm{CapData}$ [26]: it requires monotonicity of permissions and bounds, as well as equality of the inner taints of v_1 and v_2.

The primitive is sufficiently expressive to capture arbitrary monotone capability manipulations: e_1 is a valid capability derived in accordance with the capability monotonicity requirement, which we call the *guarantor* capability; e_2 expresses the arithmetic operations to build the binary representation of the new capability.

To model the race conditions that occur in CHERI-Toooba with capability manipulation, we extend the HW semantics with a rule

for executing an instruction that builds a capability.

$$\text{[EXECUTE-FORGE]}\quad \frac{\rho(i) = x \leftarrow \mathit{buildCap}(e_1, e_2)@\varepsilon \quad x \neq pc \qquad (v_1, t_1, \mathsf{C}) = [\![e_1]\!]_{apl(\rho|_i, r)} \quad (v_2, t_2, \mathsf{V}) = [\![e_2]\!]_{apl(\rho|_i, r)}}{(m, r, \rho, \mu) \xrightarrow[\text{exec } i]{} (m, r, \rho[i \mapsto x \leftarrow (v_2, t_1 \sqcup t_2, \mathsf{C})@(v_1, v_2)], \mu)}$$

In this rule, the value of e_2 is promoted to a capability without enforcing the monotonicity constraints between e_1 and e_2; these are instead deferred to commit time, as specified by the next rule:

$$\text{[COMMIT-FORGE]}\quad \frac{i = \min \mathrm{dom}(\rho) \quad x \neq pc \quad \rho(i) = x \leftarrow (v_2, t, \mathsf{C})@(v_1, v_2) \quad v_2 \sqsubseteq v_1}{(m, r, \rho, \mu) \xrightarrow[\text{commit}]{} (m, r[pc \mapsto [\![pc \dotplus 1]\!]_r][x \mapsto (v_2, t, \mathsf{C})], \rho \setminus i, \mu)}$$

Here, the rule uses the speculation tag (v_1, v_2)—corresponding to the values of the *guarantor* and the new capability—to check whether the new capability satisfies the monotonicity constraints.

By leveraging Rule [EXECUTE-FORGE], an attacker can issue a *buildCap*() instruction during speculation, and use the resulting capability to perform memory accesses that violate the CSC contract. Such problematic memory accesses would also violate our relative speculative constant time property, since the resulting capability may allow accesses to memory locations containing secrets.

These violations also illustrate how much control an attacker has over *which* secrets are leaked. Because CHERI-Toooba admits transient accesses that violate capability bounds, an attacker can steer speculation towards secrets that are never accessed architecturally, exactly as in classical Spectre attacks on non-capability architectures. A CSC-compliant design reduces this control: the attacker can only dereference capabilities reachable from the committed architectural state and leak the memory they authorize. *SCHERI* pushes this even further, by preventing leakage of *any* secret data.

7.2 Modeling BLACKOUT

To model BLACKOUT [12], there are two largely orthogonal differences to consider. First, BLACKOUT attempts to eagerly detect violations of the data-oblivious programming model and will generate faults when tainted values are stored to memory through non-blinded capabilities (except in the case of spilling tainted registers into BRRs, see below). In our formal model, this corresponds to an alternative version of the rules in Figure 7. The additional condition $t \sqsubseteq t_c$ in rule [BLACKOUT-WRITE-DATA] (highlighted) will make execution get stuck when a tainted value is written to memory through a non-blinded capability, modeling the exception that BLACKOUT would generate. Additionally, the extra conditions in rules [BLACKOUT-READ-MEM] and [BLACKOUT-WRITE-MEM] (highlighted) will make execution get stuck when memory is accessed through a tainted capability.

This eager detection of data-obliviousness violations is a consequence of the different threat model targeted by both systems. As explained in Section 4.2, BLACKOUT strives to enforce speculative constant time under five assumptions on user code. The check $t \sqsubseteq t_c$ in Figure 7 enforces the first (I1) of these assumptions, while other assumptions only hold for code generated by a trusted compiler. Our threat model is different: for assembly code that is constant time (i.e., data oblivious) in the architectural semantics,

Eager detection of data-obliviousness violations

$$\text{[BLACKOUT-WRITE-DATA]}\quad \frac{sz = 1 \quad \boxed{t \sqsubseteq t_c}}{\mathit{processWrite}((v, t, c), t_c, sz) = (v, \mathsf{V})}$$

$$\text{[BLACKOUT-READ-MEM]}\quad \frac{\mathit{checkCap}((\ell, (p, b, e, t_c)), sz, \text{RO}) \quad \mathit{processRead}(m_\mathsf{d}[\ell, \ell + sz), t_c, m_\mathsf{t}[\ell/2], sz) = (v, t, c)}{\mathit{readMem}((m_\mathsf{d}, m_\mathsf{t}), ((\ell, (p, b, e, t_c)), \boxed{\mathsf{U}}, \mathsf{C}), sz) = (v, t, c)}$$

$$\text{[BLACKOUT-WRITE-MEM]}\quad \frac{\mathit{checkCap}((\ell, (p, b, e, t_c)), sz, \text{RW}) \quad \mathit{processWrite}((v, t, c), t_c, sz) = (v', c') \quad m' = (m_\mathsf{d}[[\ell, \ell + sz) \mapsto v'], m_\mathsf{t}[\ell/2 \mapsto c'])}{\mathit{writeMem}((m_\mathsf{d}, m_\mathsf{t}), ((\ell, (p, b, e, t_c)), \boxed{\mathsf{U}}, \mathsf{C}), sz, (v, t, c)) = m'}$$

Treatment of BRRs

$$\text{[BLACKOUT-UNSPILL-VALUE]}\quad \frac{v_m = v_{\text{magic}} \quad c = \mathsf{C}}{\mathit{processRead}((v, v_m), t_c, c, 2) = ((v, v_m), \boxed{\mathsf{T}}, \mathsf{V})}$$

$$\text{[BLACKOUT-SPILL-VALUE]}\quad \frac{(v'_m, c') = (t = \boxed{\mathsf{T}}) \ ? \ (v_{\text{magic}}, \mathsf{C}) : (v_m, c)}{\mathit{processWrite}(((v, v_m), t, c), t_c, 2) = ((v, v'_m), c')}$$

Figure 7: Rules modeling eager detection of data-obliviousness violations and BRRs in BLACKOUT.

we ensure that speculation will not create new leaks and preserves constant-time behavior. As such, we make no attempt to enforce architectural security of the code and restrict ourselves to correctly propagating but not enforcing taint in the architectural semantics.

The second important change with respect to BLACKOUT is that we adopt a treatment of the stack that is *dual* to BLACKOUT's. We treat the stack capability as always tainted, and mark spills of *public* values using a distinguished encoding, URR, that sets the capability tag and uses a reserved marker value in the metadata field. Figure 7 shows modified rules that revert to BLACKOUT's original treatment of BRRs. A consequence of BLACKOUT's design is that secrets are stored in memory that is accessible through non-blinded capabilities, only guarded by the BRR's capability tag and magic value. To preserve the tainted status of such values, it is crucial to enforce atomicity of BRR writes (the magic value or tag must not be overwritten without erasing the secret value) and to ensure all reads (including partial reads) of the BRR secret value taint the result. BLACKOUT does not enforce these two properties in hardware, but instead relies on invariants of software. Section 4 shows that constructing software that satisfies these invariants is highly non-trivial, especially because the invariants must hold in transient execution as well as architecturally.

By reversing the treatment of BRRs, atomicity of URR writes becomes less important: if the magic value or the tag is overwritten, then an untainted value will be treated as tainted, which only risks degrading performance. Similarly, if a URR value is read using an instruction (e.g., a partial read) that does not correctly take into account the URR, this will also only result in unnecessarily marking the value that has been read as tainted. Moreover, URRs are unforgeable, just like CHERI's capabilities. A URR is created only

when the hardware sets the capability tag and the magic value while spilling a public value. Since software cannot set capability tags itself (a property inherited from CHERI), writing the magic value alone leaves the tag clear and yields no URR. Conversely, writing a secret over an existing URR clears its tag and thereby destroys it. Hence, *SCHERI* guarantees that URRs hold public values, and achieves sound taint tracking without any assumption on software, preventing BLACKOUT's Issue 1.

BLACKOUT narrows the attacker's control on speculative memory accesses compared to CSC-compliant systems, but less than *SCHERI*. In BLACKOUT, only secrets that are reachable through a *non-blinded* capability can leak. However, BLACKOUT keeps the stack capability non-blinded, while still allowing programs to create secret stack variables, as long as these variables are accessed only through blinded capabilities during non-speculative execution. As a result, an attacker who controls the stack capability's offset can speculatively reach a leak gadget (such as the one in Issue 2a) and leak *arbitrary* stack secrets. Such leakage is impossible in *SCHERI*, since it avoids any overlap between blinded and non-blinded capabilities, preventing BLACKOUT's Issue 2.

8 Related Work

Since CSC [16] and BLACKOUT [12] are discussed extensively throughout the paper, we focus this section on other work.

Formal models of transient execution. A large body of prior work develops formal semantics for speculative processors to detect or prove absence of transient leaks. These cover variants [3, 23] such as Spectre-PHT, Spectre-BTB, Spectre-STL, and have enabled tools for leak detection, symbolic analysis, and formal noninterference proofs [2, 4, 5, 8, 10, 18]. Our semantics builds on this line of work, but extends it with capability state, capability provenance, and capability-specific authorization checks that are central in CHERI-like systems. A foundational direction [19] formalizes security guarantees as contracts between hardware and software. Several prior works characterize the attacker-visible behavior of speculative processors and show how software guarantees must be matched to hardware behavior to obtain end-to-end confidentiality. This contract-based view is closely aligned with our approach.

Relative security. Prior works [6, 11, 17] introduce general security notions to model speculative execution vulnerabilities, which relate execution traces produced with and without speculation to determine whether speculation introduces additional leakage. We compare our security property (Theorem 6.2) against the state-of-the-art notion in [11]. First, our threat model is similar to theirs: both include control-flow and memory-access addresses in the attacker's observation. We additionally leak the value written by a store whose capability has untainted inner taint, i.e., when storing to public memory regions. We adopt a simpler setup, not modeling secret input introduced during execution or interactive attacker actions. Second, we require an additional assumption of public equivalence on initial states, which states that all secrets in the initial state are already properly tracked by taint bits on registers and capabilities. Third, the security property of [11], instantiated to our setting, reads: if the attacker observes a difference between a pair of HW traces, then there must exist a pair of ISA traces that carry the same secrets as the HW traces (respectively) and are also attacker-distinguishable. Theorem 6.2 is a stronger, less general version of this statement: rather than asserting the existence of such ISA traces, we construct them by executing the ISA semantics from the same initial ISA state as the corresponding HW trace.

Provably secure speculation for constant time. Several works study how hardware support can maintain the classic constant-time programming model during speculative execution. ProSpeCT [9] develops a processor model with secrecy tracking in the pipeline and proves that constant-time software remains secure under speculative and out-of-order execution across a range of predictors and speculation policies. Importantly, it assumes a fixed partitioning of memory into secret and public regions, which forces the stack to reside entirely in secret or public memory, with complementary efficiency-security trade-offs. *SCHERI* instead integrates secrecy into CHERI's capability system: capabilities keep track of the data's secrecy level, which is propagated through capability derivation, and preserved across spills and restores. This design choice enables secret and public objects to coexist within the same stack, but requires a new design of capability metadata, stack management (URR), and information-flow issues that do not arise in ProSpeCT.

More broadly, taint-based and delayed-transmitter processor designs demonstrate that dynamic secrecy tracking or speculative shadow structures can preserve performance while blocking Spectre-style leaks [7, 9, 25, 30, 32]. Our work shares this goal, but addresses the additional challenges introduced by capabilities, capability-derived pointers, and capability-mediated memory accesses. Relative to this line of work, our approach is closer to BLACKOUT, but BLACKOUT does not provide security proofs. To the best of our knowledge, our work is the first one to provide provably secure speculation for constant-time in CHERI.

Machine-level enforcement after compilation. A related line of work observes that source-level security guarantees may be invalidated during compilation, register allocation, and stack layout. Recent work such as SecSep [27] addresses this problem for speculative side-channel defenses by rewriting compiled assembly code to separate secret and public data after compilation, thereby accounting for register spills, stack slots, and other low-level effects that are difficult to control at the source level. SecSep improves how ProSpeCT [9] handles secrets in memory. This perspective closely aligns with our emphasis on machine-level reasoning: in capability systems, speculative leaks may similarly arise from transient interactions with spilled values, stale stack contents, or lowered memory accesses that are invisible in higher-level models. Our work differs in targeting CHERI-like capability machines and providing formal secure-speculation guarantees for capability-aware hardware semantics rather than compiler rewriting alone.

9 Conclusion

Architectural isolation in CHERI does not guarantee confidentiality under speculation. We showed that secure speculation for capability machines requires reasoning jointly about authorization and information flow, and presented SCHERI with formal guarantees for preserving constant-time behavior under speculation. Our work

provides a foundation for future capability systems that should remain secure not only architecturally, but also microarchitecturally.

Acknowledgments

This research is partially funded by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-22-1-0511, the Internal Funds KU Leuven, the Cybersecurity Research Program Flanders, the Research Foundation – Flanders (FWO) under grant number G081322N, and a European Research Council (ERC) Starting Grant (UniversalContracts; 101040088) funded by the European Union. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Research Council.

References

[1] ARM. 2020. *ARM Architecture Reference Manual Supplement Morello for A-profile Architecture.* Technical Report. ARM.

[2] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *IEEE S&P.* doi:10.1109/SP40001.2021.00046

[3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium.*

[4] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020).* doi:10.1145/3385412.3385970

[5] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE S&P.* doi:10.1109/SP46214.2022.9833707

[6] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A formal approach to secure speculation. In *Computer Security Foundations Symposium (CSF).*

[7] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *54th Annual IEEE/ACM International Symposium on Microarchitecture.*

[8] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *NDSS.* https://www.ndss-symposium.org/ndss-paper/hunting-the-haunter-efficient-relational-symbolic-execution-for-spectre-with-haunted-relse/

[9] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *USENIX Security Symposium.* https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[10] Hernán Ponce de León and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *IEEE S&P.*

[11] Brijesh Dongol, Matt Griffin, Andrei Popescu, and Jamie Wright. 2024. Relative security: Formally modeling and (dis) proving resilience against semantic optimization vulnerabilities. In *Computer Security Foundations Symposium (CSF).*

[12] Hossam ElAtali, Merve Gülmez, Thomas Nyman, and N. Asokan. 2025. BLACKOUT: Data-Oblivious Computation with Blinded Capabilities. In *ACM CCS.* doi:10.1145/3719027.3765169

[13] Hossam ElAtali, Lachlan J. Gunn, Hans Liljestrand, and N. Asokan. 2024. BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking. In *NDSS.*

[14] Franz Fuchs. 2024. *Toward transient-execution attack mitigations on CHERI.* Ph. D. Dissertation. Apollo - University of Cambridge Repository. doi:10.17863/CAM.118739

[15] Franz Anton Fuchs. 2021. *Analysis of Transient-Execution Attacks on the out-of-order CHERI-RISC-V Microprocessor Toooba.* Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[16] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Jessica Clarke, John Baldwin, Brooks Davis, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2024. Safe Speculation for Cheri. In *IEEE International Conference on Computer Design, ICCD.*

[17] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *ACM CCS.*

[18] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *IEEE S&P.*

[19] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *IEEE S&P.*

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P.* doi:10.1109/SP.2019.00002

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

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (2020).

[23] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security Symposium.*

[24] RISC-V International. 2026. RISC-V Specification for CHERI Extensions. https://github.com/riscv/riscv-cheri. GitHub repository, accessed 27 April 2026.

[25] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS.*

[26] Shixin Song, Davide Davoli, Elias Storme, Marton Bognar, Dominique Devriese, Frank Piessens, and Tamara Rezk. 2026. SCHERI: Provably Secure Speculation Under the Constant-Time Policy for CHERI (Extended Version). arXiv:2609.17399 [cs.CR] https://arxiv.org/abs/2609.17399

[27] Shixin Song, Tingzhen Dong, Kosi Nwabueze, Julian Zanders, Andres Erbsen, Adam Chlipala, and Mengjia Yan. 2025. Securing Cryptographic Software via Typed Assembly Language. In *ACM CCS.*

[28] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David F. Oswald, Frank Piessens, and Dominique Devriese. 2023. CHERI-TrEE: Flexible enclaves on capability machines. In *IEEE EuroS&P.*

[29] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE S&P.*

[30] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO.*

[31] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS.*

[32] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2020. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. *IEEE Micro* 40, 3 (2020).

A Open Science

The code of our exploits for the BLACKOUT prototype is released at https://doi.org/10.5281/zenodo.22755930. We additionally reported issues in the hardware,[2] allowing BRR writes through unblinded capabilities, and in the compiler,[3] which does not set blindedness properly for struct fields.

B Ethical Considerations

We responsibly disclosed all identified issues to the authors of BLACKOUT prior to submission. As it is a research prototype, no users are directly affected by this research. We believe that our work improves the understanding and security of speculative execution in capability systems and contributes to the design of more secure capability-based processors. These considerations motivated our decision to submit this work.

[2] https://github.com/blindedcapabilities/blinded-toooba/issues/1
[3] https://github.com/blindedcapabilities/blinded-cheri-llvm/issues/1