

Article

From RISC to Risk: Exception Handling as a Gateway to Exploitation

Mina Soltani Siapoush  and Jim Alves-Foss * 

Center for Secure and Dependable Systems, University of Idaho, Moscow, ID 83844, USA;
solt5189@vandals.uidaho.edu

* Correspondence: jimaf@uidaho.edu; Tel.: +1-208-885-5196

Abstract

RISC-V's open and extensible design improves flexibility, but it also permits vendors to customize trap behavior and reserved opcode space in ways that can affect exception handling. This paper presents a scoping review of RISC-V exception-handling security, focusing on how traps, privilege transitions, control registers, and handler routines can become attack surfaces when they are not correctly protected. We organize the literature by a five-stage exception lifecycle and map documented attacks and defenses to the architectural state they affect. Our review shows that exception-handling vulnerabilities arise from exposed control registers, insufficiently protected trap entry and return paths, and flawed handler execution. Recent attacks, such as GhostWrite, halt-and-catch-fire conditions, and side-channel exploits, illustrate how weaknesses in exception management can lead to privilege escalation, denial of service, or data leakage. In contrast, defenses are unevenly distributed across the lifecycle: backward-edge return-time integrity is addressed by multiple dedicated mechanisms, while dispatch-time and privilege-transition protection remain largely indirect and are often embedded within broader trusted-execution designs. Because the primary studies report heterogeneous cost metrics, the overhead values we extract are presented as reported rather than normalized across a common benchmark. The literature suggests that lightweight mechanisms, such as Physical Memory Protection and software-based control-flow enforcement, generally incur modest overhead, whereas stronger approaches, such as CHERI-RISC-V and Trusted-Execution Environments, may impose substantially higher costs on exception-heavy workloads. Overall, our findings indicate that exception handling should be treated as a security-critical boundary and that future work should emphasize stage-specific defenses, formal verification, and tighter integration between hardware and software protections.

Keywords: RISC-V; ISA security; vulnerability; threat models; side-channel attacks; secure execution; privilege levels; memory protection; speculative execution



Academic Editor: Paolo Bellavista

Received: 29 May 2026

Revised: 3 July 2026

Accepted: 3 July 2026

Published: 10 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction and Motivation

The Instruction Set Architecture (ISA) defines the operations a processor can perform and serves as a critical interface between hardware and software [1]. By specifying how high-level programs translate into machine-executable instructions, the ISA enables programmability and portability across platforms [2]. RISC-V [3], an open and extensible ISA developed at the University of California, Berkeley, has gained wide adoption in academia and industry. Its modular design permits vendor-specific extensions to occupy reserved encoding space and customize trap behavior beyond the base specification, which makes

exception handling a security-critical boundary. Because traps govern state preservation, privilege transitions, and return control, weaknesses in the exception path can expose the system to privilege escalation, state corruption, and control-flow hijacking. As detailed in Section 9, this structural flexibility has already introduced critical exception-handling vulnerabilities, such as the GhostWrite exploit and various halt-and-catch-fire conditions. This paper therefore examines RISC-V exception handling as a security boundary, with particular attention to how trap entry, CSR state, privilege transitions, and handler execution can be exploited or defended.

Researchers have identified multiple risks in RISC-V, including side-channel attacks that exploit speculative execution and other microarchitectural features [4,5]. Exception handling is central to system stability because it redirects control after abnormal events, such as invalid memory access, division by zero, and external interrupts. At the same time, it creates a security boundary whose correctness affects privilege transitions, context preservation, and handler execution. If exception handling is implemented incorrectly, it can expose control and status registers (CSRs), corrupt the trap state, or enable privilege escalation, data leakage, and denial-of-service attacks. These concerns make exception handling a particularly important attack surface in RISC-V systems.

This paper examines RISC-V exception handling as a security boundary. Although exceptions are essential for correct system behavior, they can also become attack vectors when trap entry, CSR preservation, privilege transitions, or handler execution are not properly protected. Accordingly, we present a lifecycle-based analysis of how exception handling can be exploited and defended in RISC-V systems.

2. Contribution

Prior surveys of RISC-V security organize the field primarily by defense family or mechanism type. Lu's survey [6] groups the literature into hardware security, ISA extensions, and memory protection, treating exception handling only incidentally rather than as an organizing axis. Gerlach et al. [7] is a primary study demonstrating concrete microarchitectural attacks on hardware RISC-V CPUs; while it advances the attack surface, it does not synthesize the defensive landscape. Broader integrity-focused taxonomies similarly bucket defenses by mechanism. Of the twelve attack vectors mapped in Section 9, seven affect S1 or S3, yet only two of the associated defense families in Section 9 report overhead figures isolated to S1/S3 protection specifically—an asymmetry that a mechanism-family taxonomy does not surface.

This paper takes a different organizing principle. We structure both attacks and defenses by the stage of the RISC-V exception lifecycle they affect: privileged-mode transition (S1), CSR state logging (S2), trap-vector dispatch (S3), exception processing (S4), and return from exception (S5). Our contribution is the mapping of attacks, defenses, and reported costs onto these stages, so it reveals a gap that mechanism-family surveys do not make visible: backward-edge, return-time integrity (S5) is addressed by several dedicated and independently benchmarked mechanisms, whereas dispatch-time and privilege-transition integrity (S1/S3) are protected only indirectly, usually as part of larger Trusted-Execution Environment designs. Beyond locating each attack and defense within the stage it affects, this perspective identifies S1 and S3 as the principal defensive gap in current research and supports the analysis with a concise formal model of the trap and CSR-preservation path (Section 8). As a scoping review, this study does not introduce new hardware experiments or simulation results; all quantitative findings are drawn from cited primary sources. The remainder of this paper is organized around four research questions that guide the lifecycle-based analysis of attacks, defenses, and their reported costs:

- Q1. Which components of the RISC-V exception lifecycle are most exposed to attack?
- Q2. Which documented and plausible attacks specifically exploit the trap and interrupt paths, as opposed to interacting with them only indirectly?
- Q3. Which defenses protect which stage(s) of the exception path, and where are the gaps?
- Q4. What are the reported costs of these defenses, and how comparable are those figures across the primary studies?

These questions are addressed throughout this paper: by the lifecycle taxonomy and the attack-vector $\times$ affected-mechanism matrix (Section 9), by the relationship and evidence labels introduced in Section 7 and applied in Section 9, and by the comparative discussion of reported costs in Section 12 and the limitations noted in Section 13.

3. Methodology

This paper presents a scoping review following the Arksey and O'Malley framework, with reporting informed by the PRISMA-ScR guidelines. Rather than attempting an exhaustive review, this study is designed as a conceptual synthesis focusing on the representative and technically relevant literature. Accordingly, the search strategy was iterative and refined throughout this study to prioritize authoritative, RISC-V-specific sources directly addressing exception, trap, and interrupt behaviors. Due to this defined scope, detailed screening counts and a full PRISMA flow diagram are not reported.

The literature search was conducted across IEEE Xplore, the ACM Digital Library, USENIX, Scopus, and the RISC-V International specification repository for records published between January 2020 and May 2026. The search utilized Boolean strings adapted to the syntax of each database, structured as: RISC-V or RISCV, exception, trap, interrupt, CSR, security, vulnerability, exploit, and attack. The selection included peer-reviewed papers, ratified specifications, and vendor documentation, while duplicate or non-RISC-V-specific records were excluded. From the finalized sources, attacks and defenses were mapped to five lifecycle stages (S1–S5), and reported metrics for execution, area, and memory costs were extracted. No new hardware implementations or simulation experiments were conducted.

The remainder of this paper is organized as follows. Section 4 provides background on the RISC-V architecture, including its operating modes, privilege levels, and relevant control and status registers. Section 5 details the RISC-V exception-handling mechanism and its five-stage lifecycle. Section 6 reviews the related work on exception- and interrupt-handling strategies. Section 7 surveys reported RISC-V security vulnerabilities. Section 8 presents the threat model and the formal model of the trap and CSR-preservation path. Section 9 analyzes attacks that directly or indirectly target the exception-handling lifecycle. Section 10 introduces the attack–defense–lifecycle taxonomy and the affected-mechanism matrix. Section 11 reviews security extensions and mitigations, organized by lifecycle stage. Section 12 discusses the findings, and finally Section 13 concludes with limitations and directions for future work.

4. RISC-V

The RISC-V architecture [3], developed at the University of California, Berkeley, re-defines processor design through its emphasis on simplicity, extensibility, and openness. Based on the RISC principle, RISC-V offers a modular instruction set with a lean base complemented by optional extensions. Its open-source and royalty-free nature has accelerated adoption across both academia and industry, enabling tailored processor implementations for a broad spectrum of applications, from embedded controllers to high-performance computing systems. Figure 1 illustrates the base instruction formats, delineating how the

architecture partitions opcodes, registers, and immediates across standard 32-bit encodings and compressed 16-bit variants. These layouts serve as the hardware-level baseline for instruction validation within the pipeline. The decoding logic strictly enforces these field boundaries; any fetched bit pattern that fails to map to these valid formats, or that attempts to exploit unmapped vendor-defined opcode space—a recurring vector in the exploits detailed in Section 9—fails decoding and immediately triggers an illegal instruction exception (Table 1). Furthermore, these fixed-width formats inherently dictate the system’s spatial execution bounds. While standard 32-bit instructions mandate strict 4-byte alignment, the compressed variants relax this boundary to 2 bytes, directly governing the hardware thresholds under which an instruction address misaligned trap is raised.

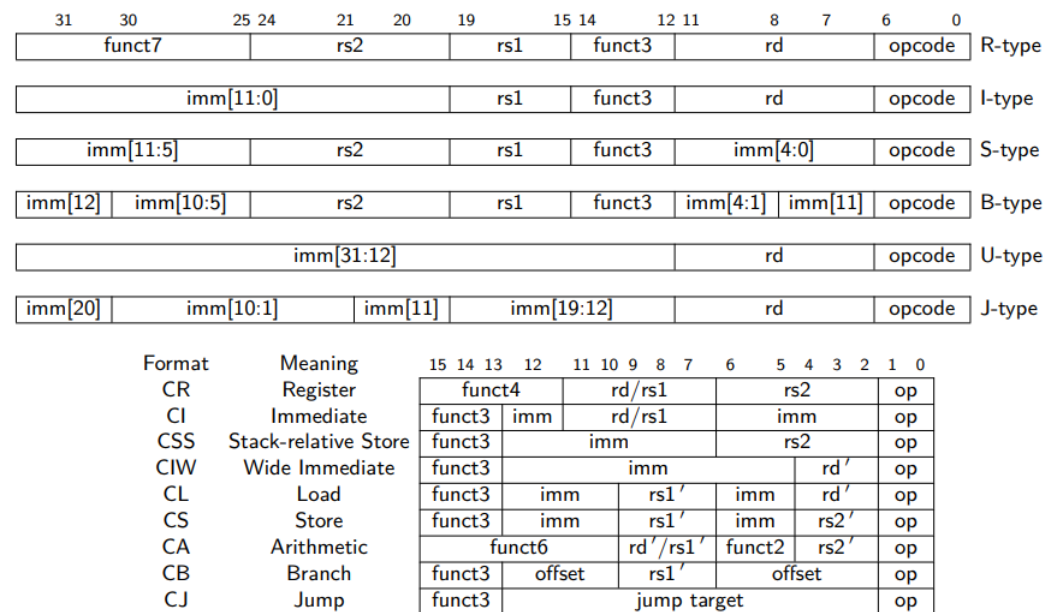


Figure 1. RISC-V instruction formats and compressed instruction encodings, adapted from the RISC-V ISA specification [3].

Table 1. List of RISC-V bases.

Name of Base or Extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers.
RV32E	Base 32-bit instruction set with only 16 registers; intended for very low-end embedded applications.
RV64I	Base 64-bit instruction set; all registers are 64-bit, with instructions to move 64-bit from/to the registers.
M	Adds integer multiply and divide instructions.
A	Adds atomic instructions needed for concurrent processing.
F	Adds single-precision (32-bit) IEEE floating-point support with 32 registers and load/store instructions.
D	Extends floating-point to double precision (64-bit) with 64-bit registers and load/store instructions.
Q	Extends floating-point to quad precision (128-bit), adding 128-bit operations.
L	Adds support for 64- and 128-bit decimal floating-point operations under IEEE standards.
C	Defines a compressed version of the instruction set for small-memory-sized embedded applications.
V	Future extension to support vector operations.
B	Future extension to support operations on bit fields.
T	Future extension to support transactional memory.
P	Extension to support packed SIMD instructions.
RV128I	Future base instruction set providing a 128-bit address space.

4.1. Why RISC-V?

RISC-V is relevant to this study due to its open and extensible Instruction Set Architecture, which allows implementers to define custom extensions and modify privileged

behavior. This flexibility introduces variability in how exceptions, traps, and interrupts are implemented across platforms.

4.2. The Base ISA

At the heart of RISC-V is its minimalist Base Integer ISA, which defines a straightforward set of instructions and 32 general-purpose registers. Its simplicity facilitates efficient implementation and high performance across platforms. The ISA's modularity allows developers to incorporate optional extensions, such as floating-point arithmetic, vector processing, and cryptographic operations, tailoring functionality to specific needs while maintaining a consistent and lean core. An overview of available base instruction sets and optional extensions is provided in Table 1, demonstrating the modularity and scalability of RISC-V.

4.3. Privilege Levels

RISC-V employs a hierarchical privilege model consisting of machine mode (M-mode), supervisor mode (S-mode), and user mode (U-mode), enabling secure and efficient system design. The specification includes Physical Memory Protection (PMP), which allows M-mode to enforce fine-grained memory access controls. Support for virtual memory via a Memory Management Unit (MMU), multi-level page tables, and a Translation Lookaside Buffer (TLB) further enhances system robustness and scalability. The architecture's exception-handling mechanism, which is an essential aspect of its privilege and memory protection model, is summarized in Table 2. It lists the standard synchronous exceptions with their *mcause* exception codes and summarizes whether each is resumable and whether it can be masked.

Table 2. Standard synchronous exceptions in RISC-V.

Exception Type	Code	Description	Res.	Mask.
Instruction address misaligned	0	Occurs when the PC is not aligned	No	No
Instruction access fault	1	Fault during instruction fetch	No	No
Illegal instruction	2	Undefined or illegal opcode	Yes	No
Breakpoint	3	Debugging breakpoint	Yes	No
Load address misaligned	4	Unaligned memory access (load)	No	No
Load access fault	5	Fault during data load	No	No
Store/AMO address misaligned	6	Unaligned memory access (store)	No	No
Store/AMO access fault	7	Fault during data store	No	No
Environment call from U-mode	8	System call from user mode	Yes	No
Environment call from M-mode	11	System call from machine mode	Yes	No
Instruction page fault	12	Page fault during instruction fetch	No	Config.
Load page fault	13	Page fault during data load	No	Config.
Store/AMO page fault	15	Page fault during data store	No	Config.

Note: Code = *mcause* exception codes per the RISC-V Privileged Specification. Res. (resumable) indicates whether execution can resume at *mepc* after handling. Mask. (maskable) indicates whether the condition can be configurably suppressed. Synchronous exceptions are generally not maskable; delegation via *medeleg* routes a trap to a different handler but does not suppress it, and *mstatus*. MIE masks interrupts rather than synchronous exceptions. Page faults are marked Config. because their trapping behavior is configurable through exception delegation. These semantics reflect common implementation behavior. However, developers should check the official RISC-V Privileged Specification for the exact requirements of their specific setup.

4.4. Registers

Beyond its base register set, RISC-V supports optional specialized registers to accommodate diverse workloads. These include registers for floating-point operations, vector computation, and bit-level manipulation. This extensible design supports application-

specific optimizations, making RISC-V suitable for domains such as machine learning, scientific computing, and real-time systems.

5. Exception Handling of RISC-V

In the RISC-V architecture, an *exception* signifies an unusual condition encountered during the execution of an instruction within a thread. These exceptions can arise from various events, such as the attempted execution of an undefined instruction or experiencing a hardware malfunction. A *trap* is the transfer of control to a handler caused by either a synchronous exception (arising from instruction execution) or an asynchronous *interrupt* (an external or timer event). Exceptions and interrupts are the two causes; the trap is the transfer they induce. *Vectored interrupts* represent a mechanism where the target address for the control transfer is determined by the specific cause of the interrupt, enabling more efficient and direct dispatch to the appropriate handler.

5.1. Types of Exceptions

RISC-V defines a range of exceptions categorized into *synchronous exceptions* arising from instruction execution and *asynchronous interrupts* triggered by external events.

Synchronous exceptions include instruction address misaligned, instruction access fault, illegal instructions, breakpoints, load/store address misaligned, load/store access faults, environment calls (*ECALL*), and instruction page faults. Asynchronous interrupts consist of software, timer, and external interrupts. While the specific set of exceptions might vary across different RISC-V implementations, these fundamental types are universally managed. When an exception occurs, control is transferred to a trap handler, which usually functions in a privileged mode. The handler retrieves information about the exception through control and status registers (*CSRs*), such as *mcause* (indicating the exception cause) and *mepc* (containing the address of the interrupted instruction or the next instruction following an interrupt).

5.2. Exception Triggering Procedure

RISC-V handles exceptions through a structured procedure involving control and status registers (*CSRs*) and transitions between privilege levels. When an exception occurs, the processor performs the following sequence of operations:

1. **Privileged-Mode Transition:** The trap is taken in the privilege mode selected by the delegation configuration, not in machine mode by default. By default, a trap is taken in M-mode; however, if the corresponding bit is set in *medeleg* (for exceptions) or *mideleg* (for interrupts) and the trap originated in a less privileged mode, the trap is instead delegated to supervisor mode. The exact target, therefore, depends on the originating privilege level, the delegation registers, and the implementation's supported modes; it is not always a transition to a higher level or to machine mode. RISC-V facilitates trap delegation through the *medeleg* (for exceptions) and *mideleg* (for interrupts) control and status registers (*CSRs*).
2. **State Logging via *CSRs*:** Critical information about the exception, which is recorded in specific *CSRs*:
 - *mcause* (or *scause*): Indicates the cause of the exception, distinguishing between interrupts and other exceptions, and specifying the type (e.g., illegal instruction, timer interrupt).
 - *mepc* (or *sepc*): Stores the address of the instruction that was interrupted or that triggered the exception, enabling the system to resume execution after the exception is handled.

- *mtval* (or *stval*): Provides additional fault-specific data, such as the memory address that triggered an access fault.
 - *mstatus* (or *sstatus*): Captures the processor's status at the time of the exception, including privilege level and interrupt enable bits.
3. Handler Dispatch via Trap Vector: The PC is redirected to the address stored in the *mtvec* (or *stvec*) register, which points to the base of the exception handler. This dispatch can occur in two modes:
 - *Direct mode*: All exceptions and interrupts jump to a common handler routine. The routine inspects the *mcause* register to identify the specific cause and determine the appropriate handling procedure.
 - *Vectored mode*: Synchronous exceptions always transfer control to the BASE address held in *mtvec* (or *stvec*), exactly as in direct mode; only asynchronous interrupts are vectored, with the target computed as $\text{BASE} + 4 \times \text{cause}$, where *cause* is the interrupt cause number. For example, an illegal instruction exception (a synchronous cause, *mcause* = 2) always dispatches to the single address *BASE* (*mtvec*), regardless of whether direct or vectored mode is configured. By contrast, a Machine Timer Interrupt (an asynchronous cause with the interrupt bit set in *mcause*) dispatches in vectored mode to $\text{BASE} (\text{mtvec}) + 4 \times 7$, placing it at a distinct entry in the handler table rather than at the shared base address used by all synchronous exceptions. No exception is ever dispatched by an offset; the offset rule applies to interrupts only. The handler still inspects *mcause* to determine the specific cause. This allows for faster dispatch to dedicated interrupt handlers without changing synchronous exception handling.
 4. Exception Processing: The trap handler performs the necessary corrective actions, such as emulating unsupported instructions, logging errors, recovering from faults, or initiating context switching.
 5. Return from Exception: After handling the exception, control is returned to the interrupted instruction using a trap-return instruction:
 - *mret*, *sret*, or *uret* restores the PC from *mepc*, resets status bits, and transitions back to the previous privilege level.

To maintain a secure and consistent environment, the exception handler must preserve the processor context, typically by saving registers either in the *mscratch* register or on the stack. Insufficient context saving can lead to information leakage or corrupted system states.

6. Related Work

RISC-V provides several mechanisms for handling exceptions and interrupts, each with different trade-offs in performance, flexibility, and implementation complexity. Trap-and-emulate handles unsupported or privileged operations by raising an *ECALL* and transferring control to a higher privilege level, where a handler emulates the operation. Exception and interrupt delegation improves efficiency by allowing machine mode to delegate selected traps to lower privilege levels, such as supervisor mode, while vectored trap handling using *mtvec* and *stvec* improves dispatch speed by directing synchronous exceptions to a fixed base address and interrupts to $\text{BASE} + 4 \times \text{cause}$ [5,8]. Software recovery restores execution by inspecting CSRs, such as *mcause* and *mtval*, then resuming via *mepc* and *MRET*, and nested exceptions with prioritization allow higher-priority traps to preempt lower-priority handlers [9].

In addition to these standard strategies, recent work has introduced techniques that improve the performance, modularity, and security of exception handling in RISC-V systems. Hardware Stacking [10] reduces context-switch latency by offloading register saving

and restoring to dedicated hardware, making it suitable for real-time embedded systems. However, this approach increases hardware resource usage and may expose predictable memory access patterns, raising security concerns [6].

By contrast, the MSCRATCH Swap method, commonly used in RISC-V Linux kernels [11], provides a software-based mechanism that saves context in the *mscratch* CSR before trap handling. While efficient and resource-light, this technique requires careful implementation to avoid state corruption or data loss. Similarly, Separate Vector Tables [12] improve modularity by decoupling synchronous exceptions and asynchronous interrupts through distinct base addresses in *mtvec*. This approach improves debugging and system maintainability, though it may increase code size and complexity, particularly in memory-constrained devices. Table 3 summarizes these strategies and recent advancements in exception handling, highlighting their primary use cases, advantages, and potential limitations.

Table 3. Recent developments in exception-handling strategies.

Strategy	Description
Trap-and-Emulate [9,13]	<i>Use Case:</i> System calls, unsupported instructions + Ensures compatibility, flexibility – Performance overhead (2–4× slower)
Interrupt Delegation [14]	<i>Use Case:</i> Multi-privilege systems, virtualization + Reduces M-mode overhead – Requires secure configuration
Vectorized Dispatch	<i>Use Case:</i> Real-time systems, high-frequency traps + Faster response, reduced latency – Requires careful handler table design
Software Recovery [15]	<i>Use Case:</i> Embedded systems, fault-tolerant designs + Maintains stability, enables recovery – Complexity in handler implementation
Nested Exceptions	<i>Use Case:</i> Real-time systems, critical event handling + Ensures timely response to priorities – Stack space constraints for nesting
Hardware Stacking [10]	<i>Use Case:</i> Systems requiring low latency + Reduces context-switch overhead – Increases hardware resource consumption
MSCRATCH Swap [11]	<i>Use Case:</i> Secure context saving + Safe register storage during traps – Depends on software handler correctness
Separate Vector Tables [12]	<i>Use Case:</i> Microcontroller profiles + Isolates exception/interrupt logic – Requires dual handler implementations
Mode-Specific Delegation [16]	<i>Use Case:</i> Multi-privilege environments + S-mode handling of user exceptions – Complex delegation register management
Asynch/Synch Split [17]	<i>Use Case:</i> Systems with mixed event types + Optimizes handling paths for interrupts – Requires precise <i>mcause</i> interpretation

Other strategies focus on optimizing control flow and privilege separation. Mode-Specific Delegation leverages the RISC-V trap delegation CSRs (*medeleg*, *mideleg*) to offload certain traps to lower privilege levels, such as supervisor mode, thereby reducing the overhead of machine mode transitions in systems with operating systems or hypervi-

sors. While this can improve performance and isolation, improper configuration may introduce serious vulnerabilities, including privilege escalation [6]. Additionally, the Asynchronous/Synchronous Split approach [17] tailors trap dispatch paths by separating external interrupts from internal exceptions using vectored *mtvec* configurations. This method streamlines mixed-event processing and reduces average interrupt latency in multitasking environments, though it requires careful interpretation of the *mcause* register to ensure events are routed to the correct handler. Collectively, these techniques illustrate ongoing advances in RISC-V exception handling aimed at balancing latency, modularity, and security.

7. Security Vulnerabilities in RISC-V

Despite being a relatively new ISA, RISC-V has already seen a range of reported security vulnerabilities in deployed and experimental implementations. Most of these issues arise from implementation choices or early design errors rather than from inherent flaws in the ISA itself. This section reviews the main classes of reported attacks and vulnerabilities in RISC-V-based processors, organized by mechanism. To aid readability, we use compact labels throughout this section: the first label indicates the relation to exception handling, where *D* denotes a direct attack on the trap path, handler state, or privilege transition, *I* denotes an indirect relationship, and *C* denotes broader contextual relevance; the second label indicates the evidence status, where *RV* means demonstrated on RISC-V, *AP* means demonstrated on another architecture but applicable to RISC-V, *H* means hypothetical or plausible, and *adj* marks an adjacent or closely related variant.

7.1. Architectural Vulnerabilities

Architectural vulnerabilities arise from flaws in processor design rather than from software bugs or isolated implementation errors. One prominent example is *GhostWrite* [18] [D-adj, RV], reported in T-Head's XuanTie C910 and C920 processors [19]. Due to an incorrect implementation of the vector extension, these CPUs allow vector instructions to access physical memory directly, bypassing virtual memory translation. This flaw may allow an unprivileged attacker to read or write physical memory and access peripheral devices, undermining memory protection mechanisms.

Another class of architectural flaws includes *halt-and-catch-fire* vulnerabilities [20] [D, RV]. In these cases, malformed instructions may cause the processor to enter an undefined or unresponsive state that requires a reboot, resulting in a denial of service. This behavior has been observed in T-Head's XuanTie C906 and C908 [19]. Unlike *GhostWrite*, these vulnerabilities primarily affect availability and are especially concerning for real-time and embedded systems.

7.2. Side-Channel Attacks

Side-channel attacks exploit unintended leakage in system behavior, such as timing variation, cache activity, power consumption, performance-counter exposure, or speculative execution effects. RISC-V processors are susceptible to several such techniques:

- Timing-based attacks [I, RV]: These attacks exploit timing variation in processor operations. In the RISC-V BOOM front-end [21], researchers demonstrated attacks that use differences in branch prediction and instruction fetching to extract sensitive information, including control-flow data and cryptographic material.
- Cache attacks [I, RV]: Cache side channels infer information from memory access patterns. On RISC-V systems, they are effective against cryptographic implementations [22]. For example, AES T-table implementations can leak key-dependent cache

accesses [23], and RSA implementations using square-and-multiply can be vulnerable to key-recovery attacks, including *Flush+Fault* on the U74 [24].

Several cache attack variants have been adapted to RISC-V, including: *Flush+Reload* [25], demonstrated on the T-Head C906 using the non-standard *dcache.civa* instruction; *Prime+Probe* [26], which exploits the deterministic replacement policy of the C906 to monitor victim cache set accesses; and *Cache+Time* [7], a RISC-V-specific technique that leverages speculative instruction prefetching and the *fence.i* instruction to infer cache activity without requiring shared memory.

- Power analysis attacks [C, AP]: Although not yet widely demonstrated on RISC-V processors, power analysis remains a relevant attack vector. By monitoring fluctuations in power consumption during execution, attackers may infer data-dependent activity such as cryptographic key bits.
- Performance-counter-based attacks [I, RV]: Hardware performance counters can reveal execution-related information when accessible to unprivileged code. On RISC-V processors, *rdinstret* has been used to infer instruction-retirement activity and reconstruct kernel memory layouts in microarchitectural KASLR attacks [27].
- Speculative execution attacks [I, RV]: These attacks exploit speculative behavior in processors to access and leak sensitive data. Some RISC-V implementations, including the T-Head C910 and C920, have been shown to expose such behavior, although others, such as the SiFive U74-MC, avoid it.

8. Threat Model

We consider three adversary tiers that reflect realistic RISC-V threat scenarios. A1 is unprivileged U-mode software with no direct access to the privileged control state. A2 is S-mode software, including a compromised operating-system component, on systems that delegate traps to lower privilege levels. A3 is a physical adversary capable of fault injection, such as voltage/clock glitching or laser-based perturbation. For each attack vector in this review, we map the vector to the minimum adversary tier required to execute it.

These tiers are not mutually exclusive across an attack chain. An adversary may begin at A1 and use a privilege-escalation vector—for example, Privilege Escalation via Exception Handling (Table 4, S1/S5)—to obtain A2 capabilities, after which S-mode-level attacks such as Rowhammer-Induced Exceptions or Exception Handler Hijacking become available that were not reachable from A1 alone. Similarly, an A3 physical adversary capable of fault injection (Section 9, Fault Injection Attacks) may use a single glitching event to skip a privilege check and land directly at A2-equivalent capability without first possessing S-mode code execution. Our per-vector minimum-tier labeling in Tables 4 and 5 identifies the lowest tier sufficient to realize each vector in isolation. While useful for single-stage analysis, the model does not extend to composite, multi-stage escalation chains under the current framework, which limits its use in informing future threat-modeling directions.

For each attack vector, we consider the minimum attacker tier, the trigger mechanism, the affected handler or CSR state, the security impact, and candidate mitigations; these attributes are captured across the lifecycle taxonomy (Table 5) and the affected-state matrix (Table 4), together with the relationship and evidence labels introduced in Section 7. We adopt this lifecycle-aligned structure because it maps directly onto the architectural mechanisms under study and supports finer-grained analysis than generic taxonomies such as STRIDE or DREAD. Specifically, STRIDE categorizes threats by type (e.g., tampering, Elevation of Privilege) independent of where in the execution flow they occur, so two STRIDE-equivalent threats—for instance, tampering with the CSR state at S2 versus tampering with return-path data at S5—are treated identically despite requiring differ-

ent mitigations (context sanitization versus shadow-stack enforcement, respectively); the lifecycle-aligned model preserves this distinction, which STRIDE and DREAD do not.

Formal Model of the Trap and CSR Path

We present a simplified formal model without attempting mechanized proof or formal verification. We model the architectural state as a tuple

$$s = (priv, pc, CSR),$$

where $priv \in \{U, S, M\}$, and CSR includes the control and exception state relevant to trap handling, such as *mepc*, *mcause*, *mstatus*, *mtvec*, *medeleg*, and *mideleg*.

A trap with cause c induces a transition

$$s \xrightarrow{\text{trap}} s' = (priv', pc', CSR').$$

For the simplified model used in this paper, trap entry is expected to preserve the following properties:

- (I1) *mepc* records the address of the faulting or interrupted instruction.
- (I2) *mcause* records the cause value c , with the interrupt bit set for asynchronous interrupts.
- (I3) *mstatus* records the prior privilege level in the appropriate saved privilege field.
- (I4) The trap is handled in M-mode by default; it is delegated to S-mode when

$$\text{delegate}(c, priv) \equiv (priv < M) \wedge \text{bit}_c(\text{medeleg} \mid \text{mideleg}) = 1,$$

That is, when the originating privilege level is below M-mode and the delegation bit corresponding to cause c is set.

- (I5) The next program counter is determined by the trap vector and the cause type:

$$pc' = \begin{cases} \text{BASE}(\text{mtvec}), & \text{for a synchronous exception,} \\ \text{BASE}(\text{mtvec}) + 4 \cdot \text{index}(c), & \text{for an asynchronous interrupt in vectored mode.} \end{cases}$$

No synchronous exception is dispatched using an offset from the trap-vector base address.

On trap return, execution resumes according to the saved return state, with privilege restored from the saved privilege field and the program counter restored from *mepc*.

Each documented attack class in this review can be interpreted as violating one or more of these invariants. Privilege-escalation attacks violate (I3) or the privilege-restoration property of the return path. Trap-vector corruption—including the undefined-mtvec-reset weakness—violates (I5). Fault-injection attacks that skip state logging violate (I1) or (I2). Expressing the attacks as invariant violations is consistent with the kind of invariant properties that tools such as RISC-V-formal are designed to check; see Section 11.1.2. Framing attacks in this way connects the threat model to a set of properties that can be checked or reasoned about formally. The formal model provides the security properties used throughout the remainder of this paper. In the next section, we map each attack vector to the exception-lifecycle stage or stages it affects and use that mapping to organize the defensive landscape.

9. Attacks Targeting Exception Handling in RISC-V

Exception handling is a critical mechanism in processor architectures, enabling the system to respond to faults, interrupts, and misaligned instructions. If the trap path, handler state, or privilege-transition logic is insufficiently protected, attackers may redirect control

flow, corrupt exception metadata, or trigger denial-of-service conditions. This section focuses on attack vectors that directly or indirectly target exception-handling behavior in RISC-V processors.

To aid clarity, each attack below carries two explicit labels. The relationship label indicates how the attack interacts with exception handling: *D* denotes a direct attack on the trap path, handler state, or privilege transition; *I* denotes an indirect interaction with traps; and *C* denotes broader contextual relevance. The evidence label indicates the status of the attack: *RV* means demonstrated on RISC-V, *AP* means demonstrated on another architecture but applicable to RISC-V, *H* means hypothetical or plausible, and *adj* marks an adjacent or closely related variant (used for GhostWrite, whose primitive lies in the vector unit but manifests through exception-related behavior).

- Exception Handler Hijacking [*D*, *AP*]: Attackers may manipulate exception vector entries or related control data to redirect control flow to attacker-controlled code. Such hijacking can arise from memory corruption vulnerabilities or insufficient protection of control registers and may result in privilege escalation or unauthorized access to system memory.
- Rowhammer-Induced Exceptions [*I*, *AP*]: Bit flips induced through repeated memory access patterns can corrupt page tables or exception-related metadata. In RISC-V systems, such corruption may lead to illegal memory accesses, exception-triggered disclosure of kernel state, or weaker isolation between protection domains.
- Interrupt-Based Timing Attacks [*D*, *AP*]: Timing differences during interrupt handling can leak sensitive information through side channels. Attackers may measure variations in interrupt latency during exception delivery and return to infer control-flow patterns or other data-dependent behavior [28].
- Privilege Escalation via Exception Handling [*D*, *H*]: Flaws in exception-handling routines may allow an attacker to transition from user mode to supervisor or machine mode. Incorrect handling of *mret*, *sret*, or *uret*, or improperly sanitized trap frames, can enable privilege escalation.
- Return-Oriented Programming on Exception Handlers [*D*, *RV*]: Exception handlers often have predictable layouts and stack behavior, making them attractive targets for return-oriented programming (ROP) [29]. By manipulating the exception stack, an attacker may chain instruction gadgets to construct code-reuse payloads.
- Fault Injection Attacks [*D*, *AP*]: Physical or electrical fault injection during exception handling may cause the processor to skip security-critical instructions or misinterpret exception types [30]. This can bypass checks, enable unauthorized access, or lead to unintended code execution.
- Misaligned Instruction Exceptions [*D*, *H*]: Inputs that generate misaligned instruction fetches can expose inconsistencies in exception processing, especially in the presence of custom ISA extensions. If the handler fails to validate or recover correctly, control flow or protected memory regions may be affected.
- Side-Channel Attacks on Exception Handlers [*I*, *AP*]: Power, timing, or cache behavior during exception handling may reveal a sensitive processor state. An attacker could exploit these variations to infer system-level secrets, including cryptographic material.

Table 4. Attack vectors mapped to the architectural state they read or write, extracted from the cited primary studies.

Attack Vector	<i>mtvec/stvec</i>	<i>mcause/scause</i>	<i>mepc/sepc</i>	<i>mstatus/sstatus</i>	<i>medeleg/mideleg</i>	Trap Frame	Handler Mem.	PMP Region	Stage(s)
Exception handler hijacking	✓						✓		S3
Rowhammer-induced exceptions		✓				✓			S2, S3
Interrupt-based timing	✓	✓							S3, S5
Privilege escalation via handler			✓	✓	✓		✓		S1, S5
ROP on handlers			✓			✓	✓		S4, S5
Fault injection		✓	✓			✓			S2, S4
Misaligned instruction exceptions		✓	✓			✓			S1, S4
Side-channel on handlers		✓				✓	✓		S2, S4
GhostWrite								✓	S1
Halt-and-catch-fire (C906/C908)	✓	✓							S3, S4
Speculative execution		✓	✓						S1, S4

Table 5. Attack–defense–lifecycle taxonomy for RISC-V exception handling. Extracted lifecycle-stage mappings and defense associations.

Attack Vector	Stage	Representative Defense(s) at Same Stage(s)
Exception Handler Hijacking	S3	PUFs [31], RoT, ASLR [32]
Rowhammer-Induced Exceptions	S2, S3	PMP [33], Dynamic PMP Churn
Interrupt-Based Timing Attacks [28]	S3, S5	Context Sanitization; Speculation Barriers (cf. KeyDrown [28], NemesisGuard [34] on other ISAs)
Privilege Escalation via Exception Handling	S1, S5	Formal verification [35–38]; PMP [33]; CHERI-RISC-V [39–42]
ROP on Exception Handlers [29]	S4, S5	CFI family (SOFIA [43], HAFIX [44], FIXER [45], LiFi-CFI [46], Bratter [47]); Zicfiss [48]; PAC [49]; RetTag [50]
Fault Injection Attacks [30]	S2, S4	Formal verification of Single-Event Upset (SEU) propagation [37]; Context Sanitization
Misaligned Instruction Exceptions	S1, S4	CFI-TE [51]; Formal verification [35]
Side-Channel Attacks on Exception Handlers	S2, S4	AES/SHA context encryption [52,53]; Context Sanitization
GhostWrite [18,19]	S1	PMP (intended to prevent bypass) [33]; CHERI capability bounds [39–42]
CPU Halting Bug, C910 [20]	S1	Formal verification of trap-generation logic [35–37]
Halt-and-Catch-Fire, C906/C908 [19,20]	S3, S4	RISCVuzz-style fuzzing [19] (used for detection rather than repair)
Speculative Execution Attacks	S1, S4	Speculation Barriers; CHERI domain tagging [39–42]; Morpheus [54,55]

Lifecycle stages: S1 = privileged-mode transition; S2 = State Logging via CSRs; S3 = Handler Dispatch via Trap Vector; S4 = exception processing; S5 = return from exception. .

Representative Vulnerabilities

Several exception-related vulnerabilities have already been observed in RISC-V CPUs. As noted in Section 7, architectural flaws such as GhostWrite [D-adj, RV] and halt-and-catch-fire conditions may also manifest through exception-related behavior, but they are discussed there as broader security vulnerabilities rather than as primary exception-handling attacks:

- CPU Halting Bug (C910 Core) [D, RV]: Triggered by reading virtual address zero, which maps to physical RAM, this bug can freeze the CPU and require a full power cycle. It acts as a denial-of-service vulnerability with roots in exception-related behavior, similar to the historical Intel F00F bug [20].

- Halt-and-Catch-Fire Bug (C906/C908 Cores) [D, RV]: Triggered by executing a vendor-specific instruction that causes an unhandled exception, this flaw can crash the CPU. It results in a denial-of-service condition and highlights the importance of robust exception handling.

10. Lifecycle Taxonomy and Defense Mapping

While Table 3 in Section 6 catalogs standard, security-agnostic exception-handling strategies (e.g., trap-and-emulate, nested exceptions), Table 5 builds on this foundation by mapping the corresponding attack vectors and defenses across lifecycle stages, linking architectural mechanisms to their security implications.

To connect the threat model with the attack surface, we map each exception-related attack vector to the lifecycle stage(s) it affects and associate it with representative defenses, highlighting where protections are well established, indirect, or absent. Table 4 further complements this view by identifying, for each attack vector, the specific architectural state it targets (e.g., trap vector, cause, EPC, status, delegation registers, trap frame, handler memory, or PMP region). This lifecycle-based perspective constitutes this paper's primary analytical contribution, revealing stage-specific asymmetries often overlooked in conventional defense-centric surveys.

11. RISC-V Security Extensions and Mitigation

This section reviews mitigation mechanisms that strengthen RISC-V exception handling. In particular, we examine defenses that protect trap entry, handler memory, CSR state, dispatch-time control flow, and exception return, and we relate each mechanism to the lifecycle stages introduced in the previous section. The subsection structure follows the major implementation families used in the literature: software-based techniques, hardware-assisted extensions, and cryptographic extensions.

11.1. Software-Based Security Techniques

Software-based techniques provide an important first layer of defense for RISC-V exception handling because they can protect control transfers, handler state, and return-path integrity without requiring major architectural changes. In this subsection, we focus on control-flow integrity and formal verification, since both are directly relevant to the lifecycle stages used in our taxonomy.

11.1.1. Control-Flow Integrity

Control-flow integrity (CFI) protects against control-flow hijacking by constraining indirect branches and return edges. In the RISC-V setting, CFI mechanisms are relevant because exception handlers and trap returns are frequent control-transfer targets. Early proposals such as SOFIA [43], HAPEI [56], HAFIX [44], SCFP [57], and FIXER [45] show that CFI can be enforced in hardware or coprocessor logic without requiring extensive changes to application code. More recent work extends these ideas with forward-edge and backward-edge protection, including LiFi-CFI [46], Bratter [47], and CFI-TE [51]. These techniques are most relevant to exception return and handler dispatch, where control-flow integrity failures can lead to code-reuse attacks or corrupted trap handling.

11.1.2. Formal Verification

Formal verification complements CFI by checking that trap logic, CSR updates, and privilege transitions behave according to specification. RISC-V-formal [35] and related work use assertion-based and model-checking techniques to detect bugs in exception-triggering behavior, including incorrect CSR handling and incomplete privilege transi-

tions [36–38,58]. For exception handling, the value of formal verification is that it can expose errors in trap entry, state preservation, and return semantics before deployment.

11.2. Hardware-Assisted Extensions

The modular architecture of RISC-V facilitates the integration of hardware-assisted security mechanisms for exception handling. These extensions strengthen security by enforcing memory isolation, protecting control-flow state, and anchoring trust in hardware. This section reviews representative hardware-based approaches that mitigate attacks such as unauthorized memory access, control-flow hijacking, privilege escalation, and state corruption. Specifically, we analyze three categories of mechanisms—memory protection architectures, runtime defense frameworks, and hardware-rooted trust mechanisms. As summarized in Figure 2, these are organized into three tiers: *core security* (e.g., memory protection, Zicfiss), *memory security* (e.g., Morpheus, CHERI-RISC-V), and *trust and isolation* (e.g., hardware-rooted trust and Trusted-Execution Environments). Together, these mechanisms provide the basis for the exception-lifecycle analysis presented in the following sections:

- Memory-protection mechanisms and control-flow defenses primarily secure handler memory and exception-return operations (S3–S5).
- Capability-based protection and memory-randomization techniques strengthen the integrity of exception state and trap-frame data (S2–S5).
- Hardware-rooted trust and trusted-execution mechanisms provide system-wide isolation and trust guarantees, indirectly reinforcing privilege transitions and trap dispatch (S1–S3).

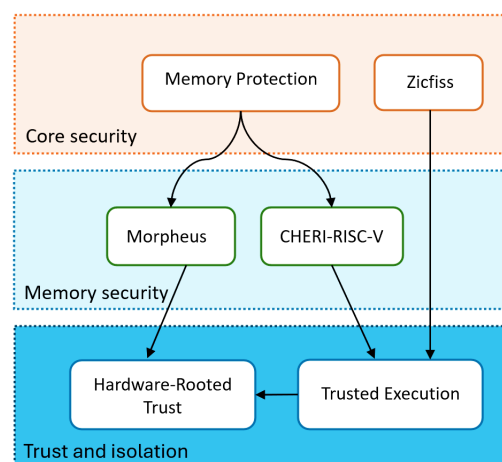


Figure 2. RISC-V hardware-assisted extensions.

11.2.1. Memory Protection Architectures

Memory protection mechanisms provide the first hardware boundary for securing exception handling by isolating handler code, trap frames, and related control data. In the RISC-V setting, Physical Memory Protection (PMP) is the most direct mechanism for enforcing region-based access control on trap-handler memory and other privileged data. Because PMP operates in machine mode through *pmpcfg* and *pmpaddr* CSRs, it can be used to prevent unauthorized access to handler stacks, trap metadata, and other sensitive regions involved in exception entry and return.

Related isolation mechanisms, such as Memory Protection Units (MPUs), extend this idea to embedded systems by enforcing hardware-access boundaries between software domains. When applied to exception handling, MPUs can reduce the risk of trap-frame corruption and handler-state tampering. Similarly, ASLR can be used to randomize handler

base addresses, making it harder for attackers to target exception dispatch or return paths. Memory tagging further strengthens these protections by attaching metadata to stack and frame objects, helping detect corrupted exception frames before an invalid state is restored.

Taken together, these mechanisms primarily support the S2, S3, and S5 stages of the exception lifecycle by restricting access to CSR-related state, trap-handler memory, and return-path data.

11.2.2. Zicfiss and Zicfilp

Zicfiss [48] and Zicfilp [59] provide ISA-level control-flow protection that is directly relevant to exception handling. Zicfiss adds shadow-stack support for return-address integrity, helping protect the S5 return stage from ROP-style attacks and corrupted trap returns. Zicfilp complements this by constraining indirect control transfers to valid landing pads (LPADs), which strengthens dispatch-time control-flow integrity at S3.

Together, these extensions provide deterministic hardware enforcement without requiring binary instrumentation or major runtime support. In our lifecycle taxonomy, they are among the most direct defenses for return-path integrity and indirect branch protection in exception-heavy code.

11.2.3. Morpheus

Morpheus [54,55] is a hardware-assisted runtime defense that strengthens control-flow integrity by combining pointer encryption, domain tagging, and periodic churn. For exception handling, its main relevance is that it makes control-flow reuse and trap-state probing significantly harder, especially when attackers attempt to predict or reuse handler entry points, saved pointers, or return-path state.

In the lifecycle taxonomy used in this paper, Morpheus most directly supports the S2 and S5 stages by protecting sensitive state during trap-related context changes and by reducing the predictability of exception-return targets. It is best viewed as a runtime hardening layer rather than a narrowly scoped trap-handler primitive, but it still provides meaningful protection for exception-heavy code paths.

11.2.4. CHERI-RISC-V

CHERI-RISC-V extends the ISA with capability-based memory protection, giving pointers explicit bounds, permissions, and integrity tags. This is directly relevant to exception handling because capabilities can constrain access to trap-handler memory, trap-frame data, and other privileged state that must remain protected during exception entry and return.

In our lifecycle taxonomy, CHERI-RISC-V most clearly supports the S2, S3, and S5 stages. It can help protect CSR-related state and handler memory during trap entry, limit unauthorized access to dispatch-time control data, and reduce the risk of corrupted return-path state being reused after an exception. CHERI is therefore best viewed as a broad hardware foundation for exception-handling security, whose relevance here lies specifically in constraining the memory and control data that exception handlers depend on, rather than as a narrowly scoped trap primitive.

11.2.5. Trusted-Execution Environments (TEEs)

Trusted-Execution Environments isolate sensitive code and data from the rest of the system, which matters for exception handling because trap handlers, the return-path state, and CSR-related data are exposed during privilege transitions and must remain protected from untrusted software. RISC-V TEE designs such as Keystone [60], Samsung's mTower, and Alibaba's XuanTie VirtualZone [61] use PMP and privilege separation to confine trusted execution and reduce the risk of trap-frame tampering, handler-state leakage, and priv-

ilege escalation during exception entry and return. Dual-core architectures [62] extend this by dedicating one core to trusted tasks, using inter-processor PMP, cryptographically verified mailboxes, and AES/SHA-accelerated shared memory to preserve confidentiality and integrity [63], and by validating interrupt sources and sanitizing trap frames before transitions into privileged modes. A key challenge in TEE exception handling is that every trap—not just explicit enclave transitions—must cross the trusted/untrusted boundary when originating from the trusted domain, since most traps are serviced via untrusted firmware or OS-level handlers; this makes trap-heavy workloads disproportionately expensive, as reflected in Table 6. Additionally, validating the trap frame is non-trivial: handlers must distinguish a genuine state from an OS-manipulated state during delegation, a limitation not addressed by PMP-based isolation and a motivation for cryptographically verified mailboxes. These mechanisms are best viewed as a broad isolation framework that indirectly strengthens exception handling rather than a dedicated trap primitive; their value lies in limiting who can observe or modify the state that exception handlers depend on. The trade-off is a performance penalty in exception-heavy workloads and the difficulty of verifying complex isolation logic, which motivates formal verification of the exception path. In our lifecycle taxonomy, TEEs are most relevant to the S1 and S3 stages, and to S5 when the protected state is restored after an exception.

Table 6. Reported costs and coverage of exception-handling defenses, separated by execution overhead, area overhead, lifecycle stage, and source. Figures are extracted from the cited primary studies and are not normalized across studies.

Mechanism	Exec. Overhead	Area Overhead	Stage	Source
PMP	low (region checks)	n/r	S3, S4	[33,64,65]
CFI (FIXER)	1.5% exec	2.9% area	S4, S5	[45]
CFI (LiFi-CFI)	n/r	<1% FPGA res.	S5 (+fwd edge)	[46]
CFI (Bratter)	5.99% exec	n/r	S4, S5	[47]
Zicfiss (shadow stack)	<1.5% IPC	n/r	S5	[48,49]
PAC/RetTag	<4% exec	n/r	S5	[49,50]
Morpheus	sub-5% exec	2% area	S2, S5 (churn)	[54,55]
CHERI-RISC-V	up to 40%	significant	S2, S3, S5	[39–42]
TEE (PMP-based)	workload-dep.	n/r	S1, S3 (indirect)	[60,61]
PUF/RoT	boot-time latency	ECC/extractor area	pre-S1	[31]
Dynamic PMP churn	5–8% (unpredict.)	n/r	S3 (layout)	[65,66]
Formal verification	none (design-time)	none	S1, S2 (logic)	[35–38]

Note: n/r = not reported by the cited source. Lifecycle stages: S1, privileged-mode transition; S2, CSR logging; S3, dispatch; S4, processing; and S5, return. This table is intended for order-of-magnitude pattern identification, not controlled quantitative comparison.

11.2.6. Hardware-Rooted Trust

Hardware-rooted trust provides a trusted foundation for protecting exception-handling infrastructure before traps are even entered. By anchoring security in device-specific secrets or hardware-generated identity, roots of trust support secure boot, measurement, and attestation of handler code and trap-state initialization. In practice, this helps prevent unauthorized replacement of exception handlers and reduces the risk that *mtvec* or other trap-control state is initialized from an untrusted image.

In our lifecycle taxonomy, hardware-rooted trust is best understood as a pre-S1 mechanism: it establishes trust in the platform before privileged-mode transitions occur and thereby strengthens the integrity of later exception-handling stages. Its value is indirect but important because exception security depends on entering the trap path from a trustworthy boot and attestation state.

11.3. Cryptographic Extensions

Cryptographic extensions strengthen exception-handling security by protecting saved context and authenticating control-flow state during trap entry and return. In this subsection, we focus on mechanisms that either secure saved context or authenticate control-flow state, since those are the cryptographic defenses most directly relevant to our lifecycle taxonomy.

11.3.1. AES and SHA

RISC-V cryptographic extensions can support the protection of exception context by accelerating the encryption and authentication of saved state during trap handling. In particular, constant-time AES- and SHA-based routines help reduce timing variability when sensitive data such as *mepc*, *mcause*, and trap-frame contents are stored or restored across privilege transitions [52,53]. This makes it harder for an attacker to infer an exception state through timing or memory-probing side channels, especially when the handler context must be written to memory before control returns to less privileged code. In our taxonomy, these mechanisms are most relevant to the S2 and S5 stages, where the sensitive state is logged and later restored.

11.3.2. Pointer Authentication Codes

Pointer Authentication Codes (PACs) and related mechanisms, such as RetTag, protect the integrity of return addresses and other control-flow pointers by attaching cryptographic tags to them. For exception handling, this is especially relevant at the S5 return stage, where corrupted *mepc* values or return-path hijacking can lead to control-flow redirection. PAC-style mechanisms therefore complement CFI and shadow-stack defenses by adding cryptographic verification to exception return.

12. Discussion

Our study shows that RISC-V's openness and extensibility create both opportunities and risks: they enable rapid innovation, but they also produce vendor-specific variation in exception-handling behavior and security hardening. This variation matters because exception handling is not just a functional mechanism; it is a security boundary that governs trap entry, handler isolation, state preservation, and exception return.

A central theme of this review is that defenses are unevenly distributed across the exception lifecycle. Stronger protection is available for some return-path and memory-isolation problems, especially through mechanisms such as CFI, PMP, CHERI, and formal verification. However, protection of trap-vector integrity and dispatch-time state is less mature, and in many systems, it is provided only indirectly as part of broader security architectures rather than as an exception-specific primitive. This asymmetry is most visible in the gap between mature backward-edge defenses at return time and the comparatively weaker protection of *mtvec/stvec* and the related dispatch state. Table 7 summarizes the principal security methods, their overhead, the lifecycle stage each affects, and their impact on exception handling.

This review also shows that several attack classes do not fit neatly into a single defense family. Speculative execution, fault injection, and side-channel attacks can all violate the assumptions made by trap handlers and CSR-state preservation. As a result, exception-handling security generally requires layered protection: architectural isolation, integrity checks on the control state, context sanitization, and, where appropriate, cryptographic or hardware-rooted mechanisms. No single mechanism fully covers the attack surface.

Table 7. Comparison of security methods for RISC-V exception handling.

Security Method	Overhead	Stage	Impact on Exception Handling
CFI [45–47]	Medium	S4, S5	Protects <i>mepc</i> and handler entry points from hijacking.
CHERI-RISC-V [39–42]	High	S2, S3, S5	Prevents CSR and memory corruption during context switches and handler execution by enforcing bounds.
PMP [33,64,65]	Low	S3, S4	Isolates exception-handler memory regions and helps prevent unauthorized access to sensitive CSRs such as <i>mepc</i> and <i>mcause</i> .
TEE [60–62,67,68]	High	S1, S3	Isolates high-risk handlers from untrusted code, reducing privilege-escalation risk.
PUFs [31]	Low	pre-S1	Validates handler code integrity during exception entry to help prevent tampering.
RoT [31,65]	Low	pre-S1	Ensures exception handlers are unmodified before execution, protecting <i>mtvec</i> integrity.
Cryptography (AES/SHA) [52,53]	Medium	S2, S5	Secures sensitive data, such as saved CSRs, during exception context switches.
Formal Verification [35–38]	Low (runtime)	S1, S2	Ensures exception logic, including CSR updates and privilege transitions, adheres to the specification.
Speculation Barriers [69]	Low	S1, S4	Prevents speculative execution during handler entry from leaking sensitive data through side channels.
Context Sanitization [52,53]	Low	S2, S5	Prevents leakage via CSRs, such as <i>mtval</i> and <i>mstatus</i> , or from stack remnants during nested exceptions.
Dynamic PMP Churn [65,66]	Medium	S3	Reduces predictability of handler memory regions, hindering attacks targeting PMP configurations.

Note: low = <2% reported execution overhead or design-time only; medium = 2–10%; high = >10% or workload-dependent; and “High (qualitative)” denotes no numeric figure reported by the source.

Finally, the stage-based taxonomy used in this paper provides a clearer view of the literature than surveys organized only by defense type. By mapping attacks and defenses to lifecycle stages, we can identify where the literature is mature, where defenses are indirect, and where important gaps remain. In particular, the lack of dedicated protection for trap-vector and dispatch integrity suggests that future RISC-V security work should treat exception entry and dispatch as first-class security targets rather than as secondary implementation details. Table 6 consolidates the reported costs of the defenses surveyed above, separated by dimension (execution and area overhead), together with the lifecycle stage each mechanism protects. The figures are reproduced as reported by the primary studies and not normalized: the studies differ in platform, benchmark, and cost metric, and several report only a subset of these dimensions. The table should therefore be read as an indication of order-of-magnitude trade-offs rather than a controlled comparison. Two patterns are nonetheless visible. First, the lightweight mechanisms that act at the return stage (S5)—shadow stacks via Zicfiss, pointer authentication, and hardware CFI—report low single-digit overheads, consistent with the maturity of backward-edge protection noted throughout this review. Second, the mechanisms that provide any protection at the transition and dispatch stages (S1/S3)—chiefly CHERI-RISC-V and TEE designs—report substantially higher or workload-dependent costs, reflecting that S1/S3 protection is currently obtained only as part of broader architectures rather than from a dedicated primitive. This cost asymmetry mirrors the defensive asymmetry that is the central finding of this review.

13. Conclusions

This paper examines exception handling within the RISC-V architecture, with an emphasis on system security. As RISC-V becomes more widely deployed, exception handling must be treated as a security-critical boundary rather than merely a functional mechanism,

since it is essential for maintaining control and privilege boundaries within processors. We show that the security implications of RISC-V exception handling arise from its trap and privilege model, not from the ISA category alone. The modularity of RISC-V introduces potential risks, as exposed control registers and loosely defined handler behavior can facilitate exploitation. Through an analysis of exception triggers, management routines, and resolution strategies, we identify both strengths and weaknesses in current RISC-V implementations. Our discussion highlights effective security mechanisms, including cryptographic integrity checks, context sanitization, hardware-enforced memory protections, and control-flow enforcement techniques. While no single solution provides complete protection, a layered defense that combines hardware, cryptography, and software can mitigate various attacks, typically incurring acceptable performance overheads. Mechanisms such as Physical Memory Protection, pointer authentication, and shadow-stack enforcement (Zicfiss) are most effective when paired with lightweight control-flow integrity, each acting on a specific lifecycle stage rather than the exception path as a whole. Nevertheless, challenges persist, including performance penalties in exception-heavy workloads and the complexity of fine-grained isolation. Many existing protections remain reactive or ad hoc, underscoring the necessity for holistic, formally verified frameworks.

Limitations: This review has several limitations. The comparative tables are drawn from heterogeneous primary studies that differ in platform, benchmark, workload, cost metric, and implementation maturity; consequently, they are illustrative of order-of-magnitude trade-offs rather than the output of a controlled experiment. Some findings are time-bounded by the rapid evolution of the RISC-V specification and its extensions, and several vendor-specific details rely on the gray literature that we label but cannot independently verify.

The stage-based lifecycle taxonomy (S1–S5) used in this review is a simplification, as some attacks (e.g., fault injection or halt-and-catch-fire conditions) may span multiple stages or exhibit cross-stage effects that are not fully captured in a single-dimensional model. The literature selection may also be subject to bias, given the concentration of RISC-V security research in a limited set of venues and the focus of our search on exception-related terminology, which may underrepresent relevant work using different vocabulary. Finally, the threat model in Section 8 abstracts adversary capability into three static tiers (A1–A3) and does not explicitly capture multi-stage escalation across tiers, which could be explored in future work.

Future Work: Future research should prioritize the co-design and integration of security mechanisms to minimize cumulative overhead while maximizing coverage. In particular, hardware–software co-designs that closely couple exception logic with security-aware microarchitectural features present promising paths to enhance performance and resilience. The formal verification of exception handlers must also become a central focus to rigorously establish correctness, safety, and resistance to tampering across various privilege levels. Moreover, adaptive security models that adjust protection levels based on runtime context could provide a dynamic balance between security and performance, particularly in systems with varying workloads and threat levels. Emerging techniques, such as machine learning for anomaly detection in exception behavior, open new frontiers, especially in heterogeneous RISC-V deployments across edge and IoT environments. Expanding paradigms like CHERI-RISC-V to include exception-specific capabilities could facilitate finer-grained memory and control-flow enforcement, thereby increasing robustness against sophisticated attacks.

Author Contributions: Conceptualization, M.S.S. and J.A.-F.; methodology, M.S.S. and J.A.-F.; software, M.S.S.; validation, M.S.S. and J.A.-F.; formal analysis, M.S.S. and J.A.-F.; investigation, M.S.S.; resources, M.S.S.; data curation, M.S.S.; writing—original draft preparation, M.S.S.; writing—review

and editing, M.S.S. and J.A.-F.; visualization, M.S.S.; supervision, J.A.-F.; project administration, J.A.-F.; funding acquisition, J.A.-F. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Schweitzer Engineering Laboratories.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Acronyms

The following abbreviations are used in this manuscript:

AES	Advanced Encryption Standard
ASLR	Address Space Layout Randomization
BOOM	Berkeley Out-of-Order Machine
CFI	Control-Flow Integrity
CHERI	Capability Hardware Enhanced RISC Instructions
CSR	Control and Status Register
CVE	Common Vulnerabilities and Exposures
DoS	Denial of Service
ECALL	Environment Call
IoT	Internet of Things
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
KASLR	Kernel Address Space Layout Randomization
LPAD	Landing Pad
MMU	Memory Management Unit
MPU	Memory Protection Unit
PAC	Pointer Authentication Code
PMP	Physical Memory Protection
PUF	Physical Unclonable Function
RISC	Reduced Instruction Set Computing
ROP	Return-Oriented Programming
RoT	Root of Trust
SEU	Single-Event Upset
TEE	Trusted-Execution Environment
TLB	Translation Lookaside Buffer

References

1. Shanley, T.; Anderson, D. *ISA System Architecture*; Addison-Wesley Professional: Boston, MA, USA, 1995.
2. Patterson, D.A.; Hennessy, J.L. *Computer Organization and Design*; Morgan Kaufmann: Burlington, MA, USA, 1994.
3. RISC-V: The Open-Standard Instruction Set Architecture. Available online: <https://github.com/riscv/> (accessed on 3 February 2026).
4. Koeune, F.; Standaert, F.X. A tutorial on physical security and side-channel attacks. In *Foundations of Security Analysis and Design III*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 78–108.
5. Gabbay, F.; Mendelson, A. *Speculative Execution Based on Value Prediction*; Technical Report 1080; Technion-IIT, Department of Electrical Engineering: Haifa, Israel, 1996.
6. Lu, T. A Survey on RISC-V Security: Hardware and Architecture. *arXiv* **2021**, arXiv:2107.04175.
7. Gerlach, L.; Weber, D.; Zhang, R.; Schwarz, M. A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs. In *Proceedings of the 44th IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 21–25 May 2023*; pp. 1234–1248.
8. SiFive. Available online: <https://www.sifive.com/boards> (accessed on 1 July 2026).

9. Chen, J.; Li, D.; Mi, Z.; Liu, Y.; Zang, B.; Guan, H.; Chen, H. Security and performance in the delegated user-level virtualization. In Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, USA, 10–12 July 2023; pp. 209–226.
10. Zhang, Z.; Zhang, X.; Shang, J.; Zhao, R.; Yan, F. Design and implementation of a fast interrupt system based on RISC-V. In Proceedings of the Second International Conference on Electronic Information Technology (EIT 2023), SPIE, Wuhan, China, 31 March–2 April 2023; Volume 12719, pp. 275–282.
11. Linux Kernel. Available online: <https://github.com/torvalds/linux> (accessed on 1 July 2026).
12. The RISC-V Microcontroller Profile. Available online: <https://github.com/emb-riscv/specs-markdown/blob/develop/exceptions-and-interrupts.md> (accessed on 1 July 2026).
13. Zabrocki, A.; Matrosov, A. Glitching RISC-V Chips: MTVEC Corruption for Hardening ISA. In Proceedings of the DEF CON 29, Conference Presentation, Las Vegas, NV, USA, 9 August 2021. Available online: <https://riscv.org/blog/video-glitching-risc-v-chips-mtvec-corruption-for-hardening-isa-adam-zabrocki-and-alex-matrosov-def-con-29/> (accessed on 16 June 2026).
14. Interrupt Management High-Level Design. Available online: <https://projectacrn.github.io/0.2/developer-guides/interrupt-hld.html> (accessed on 1 July 2026).
15. Software Fault Tolerance, Carnegie Mellon University. Available online: https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/ (accessed on 1 July 2026).
16. SiFive E21 Core Complex Manual. 2021. Available online: https://sifive-china.oss-cn-zhangjiakou.aliyuncs.com/Standard%20Core%20IP/e21_core_complex_manual_21G2.pdf (accessed on 1 July 2026).
17. Wang, W.; Wang, W.; Liu, J.; Li, L.; Li, B.; Liu, Z.; Wang, X. Attacks, defenses and perspectives for the runtime security of RISC-V IoT devices: A review. *Comput. Secur.* **2026**, *163*, 104817. <https://doi.org/10.1016/j.cose.2025.104817>.
18. Ghost Write. Available online: <https://ghostwriteattack.com/> (accessed on 1 July 2026).
19. Thomas, F.; Hetterich, L.; Zhang, R.; Weber, D.; Gerlach, L.; Schwarz, M. RISCvuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing. 2024. Available online: <https://ghostwriteattack.com/riscvuzz.pdf> (accessed on 1 July 2026).
20. Collins, R.R. The Pentium F00F Bug. 1998. Available online: <http://www.rcollins.org/ddj/May98/F00FBug.html> (accessed on 1 July 2026).
21. Chavda, R.J. Side-Channel Attacks in RISC-V BOOM Front-End. Ph.D. Thesis, Virginia Tech, Blacksburg, VA, USA, 2023.
22. Ahmadi, M.M.; Khalid, F.; Shafique, M. Side-channel attacks on risc-v processors: Current progress, challenges, and opportunities. *arXiv* **2021**, arXiv:2106.08877.
23. Osvik, D.A.; Shamir, A.; Tromer, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, 13–17 February 2006*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 1–20.
24. Wardlaw, W.P. The RSA public key cryptosystem. In *Coding Theory and Cryptography: From Enigma and Geheimschreiber to Quantum Theory*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 101–123.
25. De Mulder, E.; Gummalla, S.; Hutter, M. Protecting RISC-V against side-channel attacks. In Proceedings of the 56th Annual Design Automation Conference 2019, Las Vegas, NV, USA, 2–6 June 2019; pp. 1–4.
26. Wang, Q.; Zhang, X.; Wang, H.; Gu, Y.; Tang, M. BackCache: Mitigating contention-based cache timing attacks by hiding cache line evictions. *arXiv* **2023**, arXiv:2304.10268.
27. Seuscheck, H.; Rass, S. Side-channel leakage models for RISC instruction set architectures from empirical data. In Proceedings of the Euromicro Conference on Digital System Design, Madeira, Portugal, 26–28 August 2015; pp. 423–430.
28. Schwarz, M.; Lipp, M.; Gruss, D.; Weiser, S.; Maurice, C.L.N.; Spreitzer, R.; Mangard, S. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In Proceedings of the Network and Distributed System Security Symposium 2018, San Diego, CA, USA, 18–21 February 2018; p. 15.
29. Jaloyan, G.A.; Markantonakis, K.; Akram, R.N.; Robin, D.; Mayes, K.; Naccache, D. Return-oriented programming on RISC-V. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, 5–9 October 2020; pp. 471–480.
30. Malik, A.A.; Mihir, H.; Aysu, A. Craft: Characterizing and root-causing fault injection threats at pre-silicon. *arXiv* **2025**, arXiv:2503.03877.
31. Bringer, J.; Chabanne, H.; Icart, T. On Physical Obfuscation of Cryptographic Algorithms. In *Proceedings of the International Conference on Cryptology in India (INDOCRYPT), New Delhi, India, 13–16 December 2009*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 88–103.
32. Verhoeven, H. Static Binary Rewriting for ROP Gadget Removal. Ph.D. Thesis, Dakota State University, Madison, SD, USA, 2023.
33. Waterman, A.; Lee, Y.; Avizienis, R.; Patterson, D.A.; Asanovic, K. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.7*; Technical Report UCB/EECS-2016-129; EECS Department, University of California, Berkeley, Berkeley, CA, USA, 2016.
34. Salehi, M.; Borger, G.D.; Hughes, D.; Crispo, B. NemesisGuard: Mitigating interrupt latency side channel attacks with static binary rewriting. *Comput. Netw.* **2022**, *205*, 108744. <https://doi.org/10.1016/j.comnet.2021.108744>.

35. RISC-V Formal Verification Framework. Available online: <https://github.com/YosysHQ/riscv-formal> (accessed on 1 July 2026).
36. Verifying Security of RISC-V Processors. Available online: <https://www.embedded.com/verifying-security-of-risc-v-processors/> (accessed on 1 July 2026).
37. Xue, B.; Zwolinski, M. Using Formal Verification to Evaluate Single Event Upsets in a RISC-V Core. *arXiv* **2024**, arXiv:2405.12089.
38. RISC-V Formal Verification Framework Extension for Synopsys VC Formal. Available online: <https://blog.yosyshq.com/p/risc-v-formal-verification-framework-extension-for-synopsys-vc-formal/> (accessed on 1 July 2026).
39. Watson, R.N.M.; Moore, S.W.; Sewell, P.; Neumann, P.G. *An Introduction to CHERI*; Technical Report UCAM-CL-TR-941; University of Cambridge, Computer Laboratory: Cambridge, UK, 2019.
40. CHERI Instruction-Set Architecture (Version 9). 2023. Available online: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.html> (accessed on 1 July 2026).
41. Xia, H.; Woodruff, J.; Barral, H.; Esswood, L.; Joannou, A.; Kovacsics, R.; Chisnall, D.; Roe, M.; Davis, B.; Napierala, E.; et al. CheriRTOS: A Capability Model for Embedded Devices. In Proceedings of the IEEE 36th International Conference on Computer Design (ICCD), Orlando, FL, USA, 7–10 October 2018; pp. 92–99.
42. Almatary, H.; Dodson, M.; Clarke, J.; Rugg, P.; Gomes, I.; Podhradský, M.; Neumann, P.G.; Moore, S.W.; Watson, R.N.M. CompartOS: CHERI Compartmentalization for Embedded Systems. *arXiv* **2022**, arXiv:2206.02852.
43. De Clercq, R.; Götzfried, J.; Übler, D.; Maene, P.; Verbauwhede, I. SOFIA: Software and cOntrol Flow Integrity Architecture. *Comput. Secur.* **2017**, *68*, 16–35.
44. Davi, L.; Hanreich, M.; Paul, D.; Sadeghi, A.R.; Koeberl, P.; Sullivan, D.; Arias, O.; Jin, Y. HAFIX: Hardware-assisted flow integrity extension. In Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.
45. De, A.; Basu, A.; Ghosh, S.; Jaeger, T. FIXER: Flow integrity extensions for embedded RISC-V. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 25–29 March 2019; pp. 348–353.
46. Roodsari, M.S.; Sheikhsheaei, F.; Maunero, N.; Prinetto, P.; Navabi, Z. LiFi-CFI: Light-weight Fine-grained Hardware CFI Protection for RISC-V. In Proceedings of the IEEE International Conference on Design, Test and Technology of Integrated Systems (DTTIS), Gammarth, Tunisia, 1–4 November 2023; pp. 1–6.
47. Park, S.; Kang, D.; Kang, J.; Kwon, D. Bratter: An instruction set extension for forward control-flow integrity in RISC-V. *Sensors* **2022**, *22*, 1392.
48. Shadow Stack (Zicfiss). Available online: https://github.com/riscv/riscv-cfi/blob/main/src/cfi_backward.adoc (accessed on 1 July 2026).
49. Wang, Y.; Wu, J.; Yue, T.; Ning, Z.; Zhang, F. RetTag: Hardware-assisted return address integrity on RISC-V. In Proceedings of the 15th European Workshop on Systems Security, Rennes, France, 5–8 April 2022; pp. 50–56.
50. Zhao, Y. Protecting Return Address Integrity for RISC-V via Pointer Authentication. Master’s Thesis, University of Massachusetts Amherst, Amherst, MA, USA, 2024. <https://doi.org/10.7275/33124>.
51. Zgheib, A.; Potin, O.; Rigaud, J.B.; Dutertre, J.M. A CFI verification system based on the RISC-V instruction trace encoder. In Proceedings of the 2022 25th Euromicro Conference on Digital System Design (DSD), Maspalomas, Spain, 31 August–2 September 2022; pp. 456–463.
52. Marshall, B.; Zeh, A.; Glew, A.; Spinney, B.; Page, D.; Atkins, D.; Dockser, K.; Saarinen, M.J.O.; Menhorn, N.; Deutsch, L.P.; et al. *The RISC-V Cryptography Extensions, Volume I: Scalar & Entropy Source Instructions*; RISC-V International: Zürich, Switzerland, 2022.
53. Adams, A.; Gupta, P.; Tine, B.; Kim, H. Cryptography Acceleration in a RISC-V GPGPU. In Proceedings of the Workshop on Computer Architecture Research with RISC-V (CARRV 2021), Virtual, 17 June 2021.
54. Harris, A.; Verma, T.; Wei, S.; Biernacki, L.; Kisil, A.; Aga, M.T.; Bertacco, V.; Kasikci, B.; Tiwari, M.; Austin, T. Morpheus II: A RISC-V security extension for protecting vulnerable software and hardware. In Proceedings of the International Symposium on Hardware Oriented Security and Trust (HOST), Tysons Corner, VA, USA, 12–15 December 2021; pp. 226–238.
55. Gallagher, M.; Biernacki, L.; Chen, S.; Aweke, Z.B.; Yitbarek, S.F.; Aga, M.T.; Harris, A.; Xu, Z.; Kasikci, B.; Bertacco, V.; et al. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 13–17 April 2019; pp. 469–484.
56. Lashermes, R.; Le Boudier, H.; Thomas, G. Hardware-assisted program execution integrity: Hapei. In *Proceedings of the Nordic Conference on Secure IT Systems*; Springer: Cham, Switzerland, 2018; pp. 405–420.
57. Werner, M.; Unterluggauer, T.; Schaffenrath, D.; Mangard, S. Sponge-based control-flow protection for IoT devices. In Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 214–226.
58. Faulty Instructions in Alibaba’s T-Head C910 RISC-V CPUs Blow Away All Security, Industry Press. Available online: https://www.theregister.com/2024/08/07/riscv_business_thead_c910_vulnerable/ (accessed on 1 July 2026).
59. Landing Pad (Zicfilp). Available online: https://github.com/riscv/riscv-cfi/blob/main/src/cfi_forward.adoc (accessed on 1 July 2026).

60. Lee, D.; Kohlbrenner, D.; Shinde, S.; Asanović, K.; Song, D. Keystone: An open framework for architecting trusted execution environments. In Proceedings of the Fifteenth European Conference on Computer Systems, Heraklion, Greece, 27–30 April 2020; pp. 1–16.
61. XuanTie VirtualZone: RISC-V-Based Security Extensions | Xuan Jian, Alibaba Cloud. 2022. Available online: <https://riscv.org/blog/2022/04/xuantie-virtualzone-risc-v-based-security-extensions-xuan-jian-alibaba/> (accessed on 1 July 2026).
62. Chen, Y.; Chen, H.; Chen, S.; Han, C.; Ye, W.; Liu, Y.; Zhou, H. Dites: A lightweight and flexible dual-core isolated trusted execution soc based on RISC-V. *Sensors* **2022**, *22*, 5981.
63. TEE SoC Based on RISC-V. 2023. Available online: <https://riscv.org/blog/2023/02/tee-soc-based-on-risc-v/> (accessed on 15 July 2025).
64. Barbareschi, M.; Barone, S.; Casola, V.; Montone, P.; Moriconi, A. A Memory Protection Strategy for Resource Constrained Devices in Safety Critical Applications. In Proceedings of the 6th International Conference on System Reliability and Safety (ICSRS), Venice, Italy, 23–25 November 2022; pp. 533–538.
65. Sevaldson, R. Memory Protection for Embedded RISC-V Systems. Master’s Thesis, NTNU, Trondheim, Norway, 2022.
66. Mideus, G.; Morales Chavez, A. RISC-V Thread Isolation: Using Zephyr RTOS. 2020. Available online: <https://www.diva-portal.org/smash/get/diva2:1458135/FULLTEXT01.pdf> (accessed on 1 July 2026).
67. Kuhne, M.; Volos, S.; Shinde, S. Dorami: Privilege Separating Security Monitor on RISC-V TEEs. *arXiv* **2024**, arXiv:2410.03653.
68. Bahmani, R.; Brasser, F.; Dessouky, G.; Jauernig, P.; Klimmek, M.; Sadeghi, A.R.; Stapf, E. CURE: A Security Architecture with CUsomizable and Resilient Enclaves. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*; USENIX Association: Berkeley, CA, USA, 2021; pp. 1073–1090.
69. Sabbagh, M.; Fei, Y. Secure speculative execution via RISC-V open hardware design. In Proceedings of the Fifth Workshop on Computer Architecture Research with RISC-V (CARRV), Virtual, 17 June 2021.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.