Cross-Core Covert Channel for RISC-V: Implementation, Countermeasures and Cross-Platform Analysis

Mahreen Khan[1], Renaud Pacalet[1], Maria Mushtaq[1], and Ludovic Apvrille[1]

Telecom Paris, Institut Polytechnique de Paris, France
firstname.secondname@telecom-paris.fr

Abstract. Cache-based covert channels exploit microarchitectural timing differences to enable unauthorized communication between processes. While extensively studied on x86 architectures, such channels remain underexplored in the emerging RISC-V ecosystem. This paper presents the design and implementation of a novel prefetcher and cache timing covert channel for RISC-V platforms that exploits the timing difference between cached and uncached memory accesses. Our implementation supports both standardized RISC-V cache management extensions (Zicbom and Zicbop) and vendor-specific instructions (T-Head C910 custom instructions), demonstrating cross-platform portability across heterogeneous RISC-V implementations. The sender encodes bits by selectively prefetching or flushing a shared cache line, while the receiver decodes information by measuring memory access latency. Through careful synchronization using POSIX shared memory and atomic operations, we achieve reliable bit transmission on both the RISC-V gem5 full-system simulator (Sifive U54 core) and physical RISC-V Beagle-V Ahead (T-Head C910 core). Our paper contributes to understanding the security implications of cache and prefetcher management instructions in RISC-V systems and provides a foundation for developing detection and mitigation strategies for this emerging architecture.

Keywords: RISC-V Architecture · gem5 Simulator · Beagle-V Ahead· Covert channels · Embedded Systems · Side-Channel Attacks · Attack Assessment · Microarchitectural Security · Security Evaluation

1 Introduction

Covert channels represent a fundamental security challenge in modern computing systems, enabling unauthorized communication that violates security policies and bypasses traditional isolation mechanisms [22]. These channels exploit shared system resources to transmit information between processes that should remain isolated by operating system security boundaries. Among the various covert channel techniques, cache-based timing channels have proven particularly effective due to their high bandwidth potential, reliability across diverse platforms, and difficulty of detection [24],[9]. These channels fundamentally exploit

the observable timing differences between accessing data present in the processor cache versus data that must be fetched from slower main memory, creating a side channel that can leak sensitive information or enable covert communication.

The RISC-V instruction set architecture has gained significant momentum as an open source alternative to proprietary architectures, with growing adoption in embedded systems, high-performance computing, and security-critical domains [41][1]. Unlike established architectures with decades of security research, RISC-V presents a relatively nascent security landscape where fundamental vulnerabilities are still being discovered and characterized [10,18,19,17]. As RISC-V deployments expand into security-sensitive applications including trusted execution environments, cryptographic accelerators, and isolated computing platforms, understanding potential microarchitectural vulnerabilities becomes essential for building secure systems. While cache timing attacks have been extensively studied on x86 and ARM architectures [29], [24], [46], [12], [23], [21], their manifestation on RISC-V platforms with different cache management instructions, diverse implementation choices, and varying microarchitectural features warrants dedicated investigation.

This paper addresses the security implications of cache and prefetcher management instructions in RISC-V architectures by implementing and analyzing a practical covert channel that exploits cross-core timing side channels. Our implementation leverages the fundamental principle that memory access latency depends critically on cache state, where accessing cached data completes in a handful of processor cycles while accessing evicted data requires fetching from lower cache levels or main memory, incurring latencies orders of magnitude larger. The sender modulates this observable timing difference by selectively prefetching data to bring it into cache or flushing data to evict it from cache at a shared memory location, while the receiver infers transmitted bits by measuring access latency and comparing against a calibrated threshold.

A key challenge in RISC-V security research is platform diversity and implementation heterogeneity. Unlike mature architectures with relatively uniform instruction sets and well-established microarchitectural features, RISC-V implementations vary significantly across vendors, market segments, and design points. The standard Zicbom extension provides cache block management operations including flush, clean, and invalidate instructions, while Zicbop adds software-controlled prefetch hints for read, write, and instruction fetch operations [34], [33]. However, existing commercial hardware implementations like the T-Head C910 core deployed in BeagleV development boards use custom cache management instructions [38], [3]. Our implementation addresses this heterogeneity through conditional compilation and platform abstraction, supporting both standardized instructions and vendor-specific instructions. Despite architectural divergence, we demonstrate a functional and successful implementation of our covert channel across diverse RISC-V systems.

Furthermore, our covert channel facilitates practical attack scenarios in real-world RISC-V deployments. For example, distinct malicious processes running on the same system could silently exchange sensitive information, such as cryp-

tographic keys or credentials, without relying on files, system calls, or network communication, thereby avoiding traditional monitoring mechanisms [31,47,24]. In containerized or virtualized environments, such microarchitectural communication can bypass software-enforced isolation boundaries, including namespaces, cgroups, and even hypervisor separation [44,35]. Moreover, the channel can act as a secondary exfiltration path in multi-stage attacks, relaying data extracted through other side-channel techniques and enabling composite attack strategies that circumvent defenses designed under isolated threat assumptions [5].

1.1 Contributions

This paper makes the following contributions to the security analysis of RISC-V microarchitecture:

1. We present a complete, novel working covert channel implementation that functions correctly across multiple RISC-V platforms using the gem5 full-system simulator configured with standard Zicbom (for cache instructions) and Zicbop (for prefetcher instructions) extensions and physical T-Head C910 hardware with custom cache instructions.
2. Our sender-receiver protocol incorporates careful timing and synchronization mechanisms necessary for reliable communication on systems. We employ POSIX shared memory with atomic operations to coordinate between processes while maintaining the covert nature of the channel.
3. We provide detailed implementation insights, including calibration procedures for threshold selection, bit encoding schemes using least-significant-bit ordering, and robustness techniques like double-flushing that improve reliability in the presence of hardware prefetchers and cache pollution.
4. We discuss countermeasures against cache and prefetcher-based covert channels, analyzing their strengths and limitations.
5. We provide the full implementation along with a step-by-step reproducibility guide, available through our public GitHub repository at https://github.com/MKIP23/RISCV_Covert_Channel.

1.2 Paper Organization

The remainder of this paper proceeds as follows. Section 2 provides necessary background on cache architectures, RISC-V cache management extensions, and surveys related work on cache timing attacks, positioning our work within the broader literature. Section 3 details the implementation across different platforms, including the threat model, sender and receiver logic, and compilation procedures. Section 4 discusses experimental methodology and evaluation results. Section 5 analyzes potential countermeasures and their tradeoff with performance. Section 6 provides a discussion and outlines directions for future research. Section 7 concludes with a summary of contributions.

2 Background and Related Work

2.1 Cache Architecture Fundamentals

Modern processors employ hierarchical memory systems where fast but expensive caches buffer frequently accessed data from slower, cheaper main memory [13]. This hierarchy exploits temporal and spatial locality, where programs tend to reuse recently accessed data and access nearby memory locations [30]. When a processor issues a memory request, it first queries the fastest cache level (typically L1), then progressively slower cache levels (L2, L3), and finally main memory if the data is not cached. A cache hit occurs when the requested data resides in the cache, completing within a few processor cycles, whereas a cache miss forces the processor to fetch data from a lower level, causing delays ranging from tens of cycles for L2 accesses to hundreds of cycles for main memory accesses. This pronounced timing disparity between hits and misses forms the foundation of microarchitectural timing side-channel attacks [46],[24].

Caches store data in fixed-size blocks called cache lines, typically 64 bytes on modern architectures. Accessing any byte of a cache line triggers the transfer of the entire line into the cache. Similarly, cache-management instructions such as flush operate at cache-line granularity, evicting the entire 64-byte block. Most commercial processors implement set-associative caches, where each memory address maps to a specific cache set, and within each set, the data may occupy any available way (e.g., 4-way or 8-way associativity). Replacement policies determine which cache line to evict when a set is full, with least-recently-used (LRU) and pseudo-LRU being dominant choices. These policies introduce observable patterns in access latencies that adversaries can exploit to construct high-resolution side-channel attacks.

2.2 Prefetcher Fundamentals

To mitigate the high latency gap between processor cores and main memory, modern CPUs incorporate prefetchers that predict future memory accesses and fetch the corresponding cache lines before the processor explicitly demands them [11]. Prefetchers leverage patterns in program behavior, such as sequential iteration, fixed-stride memory traversals, or recurring address streams, to anticipate upcoming accesses. When predictions are accurate, prefetching significantly reduces miss rates and improves effective memory bandwidth. Most commercial systems deploy a combination of spatial and stride prefetchers. Spatial prefetchers detect linear progressions and fetch the next consecutive cache line, while stride prefetchers infer regular address deltas across iterations of loops [11]. More advanced designs use correlation or Markov-based prediction tables to capture complex or irregular memory-access patterns [16].

Although prefetchers aim to improve performance, they inadvertently influence microarchitectural side channels. Because prefetching modifies cache contents independent of explicit program loads, it can introduce noise that obscures or amplifies side-channel signals [11]. Moreover, adversaries can manipulate prefetchers to bring specific cache lines into the hierarchy, enabling attack

primitives such as prefetch-based address probing, weakening address-space isolation, and bypassing privilege boundaries [11]. Consequently, prefetchers serve as both a performance optimization and an attack surface that must be carefully evaluated when designing secure microarchitectures.

2.3 Covert Channels

Lampson's seminal 1973 paper "A Note on the Confinement Problem" introduced the concept of covert channels as communication paths not intended for information transfer [22]. Lampson distinguished between legitimate channels explicitly designed for communication and covert channels that exploit system resources in unintended ways.

Cache-based covert channels specifically exploit cache state as a shared resource for unauthorized communication. Xu et al. demonstrated high-bandwidth cache covert channels in virtualized environments [45]. Maurice et al. extended this work with robust cache covert channels capable of establishing SSH sessions between isolated virtual machines in cloud environments [27].

Wu et al. explored cross-core covert channels on multicore processors, demonstrating that cache coherence protocols create exploitable timing channels between cores even without shared cache levels [43]. Inci et al. demonstrated cache covert channels on ARM processors, showing that these vulnerabilities extend beyond x86 architectures [15].

2.4 RISC-V Security Research

RISC-V security research remains relatively immature compared to established architectures, though interest has grown rapidly as RISC-V adoption accelerates. The open nature of the RISC-V specification enables transparency in security analysis, but also creates challenges due to implementation diversity.

Dessouky et al. conducted a systematic analysis of hardware security features in RISC-V processors through their HardFails framework [7]. Their study examined several RISC-V cores, including Rocket and BOOM, and uncovered multiple vulnerability classes, such as timing channels created by shared microarchitectural resources. Canella et al. provided a comprehensive survey of transient execution attacks across different architectures, including RISC-V, highlighting recurring vulnerability patterns and cross-architecture similarities [5]. Gerlach et al. [10] demonstrated microarchitectural attacks on RISC-V cores such as the C906 and U74. Khan et al. [17], [18] conducted a microarchitectural side-channel analysis on the RISC-V architecture using gem5. Schrammel et al. showed that interrupt latency can itself form an exploitable side channel on RISC-V systems, even in minimal embedded-class processors without caches [36].

More recently, Thomas et al. introduced RISCover, an automated framework for discovering user-exploitable architectural security vulnerabilities in closed-source RISC-V CPUs [39]. Their work uncovered the GhostWrite vulnerability, which enables arbitrary read and write access to memory across privilege domains on multiple commercial RISC-V cores, including the T-Head XuanTie

C910. This finding demonstrates that certain architectural design flaws can fundamentally undermine isolation guarantees, rendering traditional covert-channel mitigations ineffective. Complementing this line of work, Austa et al. performed a systematic assessment of cache timing vulnerabilities on commercial RISC-V processors [2]. Their evaluation of the SiFive U54 and T-Head C910 cores revealed that the C910 exhibits significantly more distinct timing behaviors, suggesting a broader microarchitectural attack surface. These results highlight the variability in leakage characteristics across RISC-V implementations and reinforce the need for architecture-aware vulnerability assessment. Wistoff et al. investigated both the detection and mitigation of microarchitectural covert channels on open-source RISC-V cores, demonstrating practical leakage evaluation techniques and proposing hardware-level countermeasures to prevent cross-domain information flow [42].

Building on these insights, our paper focuses on the combined behavior of the prefetcher and cache subsystems across different RISC-V extensions. By characterizing their microarchitectural interactions, we develop a novel covert channel that leverages prefetcher-driven cache state manipulation, enabling information transfer not previously demonstrated in the RISC-V architecture.

3 RISC-V Covert Channel Design

3.1 Threat Model and Assumptions

We consider a scenario where two unprivileged processes (either on the same core or different cores) executing on the same RISC-V system seek to establish covert communication despite operating system isolation policies that prohibit direct information exchange. Both processes possess standard capabilities available to typical user-space applications.

Both processes can read hardware cycle counters via rdcycle for timing measurements, a capability provided by the base RISC-V ISA for performance profiling and optimization. Both can execute cache management instructions and can use standard POSIX synchronization primitives, including atomic operations, to coordinate their actions through the shared memory region.

The attacker's goal is to establish a cache and prefetcher-based covert communication channel that transmits information reliably while remaining undetectable to standard security monitoring that does not observe microarchitectural events.

3.2 Channel Protocol Design

Our covert channel encodes information by manipulating the cache state of a single shared memory line. Two unprivileged processes, the Trojan (sender) and the Spy (receiver), map the same shared memory segment, which contains control flags and one cache-line-aligned data block used as the transmission medium. The covert channel design is illustrated in Figure 1.

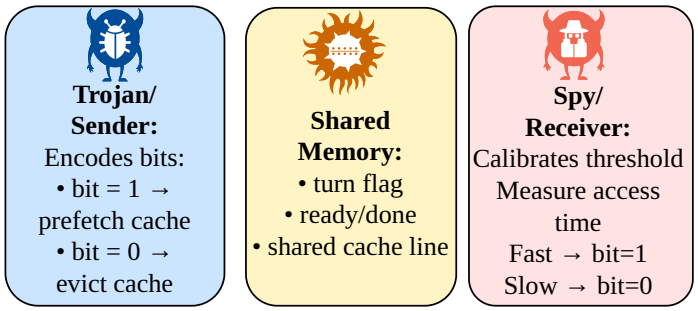

Fig. 1. A Trojan modifies the cache state of a shared line to transmit bits. Spy measures access latency to decode them. Synchronization uses atomic flags in shared memory.

Communication proceeds in alternating steps. The Trojan encodes each bit by either caching the shared line (bit 1) or evicting it (bit 0). After modifying the cache state, it signals the Spy via an atomic turn variable. The Spy measures the access latency to the shared line using rdcycle. A fast access implies the line is cached (bit 1), while a slow access indicates eviction (bit 0). It then returns control to the Trojan.

Before transmission, the Spy performs a short calibration phase to determine a timing threshold that reliably separates cached from evicted accesses. The message is transmitted bit-by-bit in LSB-first order, beginning with the message length. Synchronization, turn-taking, and termination are implemented using atomic variables within the shared memory structure.

3.3 Shared Memory Structure

The shared memory region defined in our implementation contains both control variables for protocol synchronization and the target cache line for information encoding (Listing 1.1). The ready flag indicates when the receiver has completed initialization, and the sender may begin transmission. The turn variable implements a simple turn-taking protocol where zero indicates the sender may act and one indicates the receiver may measure. The bit_value field stores the last transmitted bit for debugging purposes, though covert operations would omit this information leakage. The msg_len field communicates the message length in bytes from sender to receiver before transmission begins. The done flag indicates transmission completion, allowing graceful protocol termination. Finally, the target_line array contains 64 bytes aligned to cache line boundaries through the aligned attribute, ensuring that cache operations on this array affect exactly one cache line without inadvertently impacting adjacent data structures.

All control variables use atomic operations for safe concurrent access between the sender and receiver processes. The C11 atomic builtins like atomic_store_n and atomic_load_n with sequential consistency memory ordering ensure that updates become visible across processes in a well-defined order without race

conditions. Memory ordering is critical for correct protocol operation since both processes read and write shared variables, and the protocol relies on the specific ordering of these operations to maintain synchronization.

```
#include <stdint.h>
#define CACHELINE 64
#define SHM_NAME "/prefetch_cbo_shm_v2_fix"
#define MESSAGE_MAX_LEN 128
#define MAX_BITS (MESSAGE_MAX_LEN * 8)
struct shared_area {
    int ready;
    int turn;
    int bit_value;
    int msg_len;
    int done;
    target_line[CACHELINE] __attribute__((aligned(CACHELINE)));
};
```

Listing 1.1. Control variables for protocol synchronization

3.4 Sender/Trojan Implementation Details

The sender process begins by attempting to open the shared memory region created by the receiver (Listing 1.2). Since the receiver must initialize the region before the sender can attach, the sender implements a retry loop that repeatedly calls shm_open until successful or a maximum retry count is reached. Each iteration waits one millisecond before retrying, providing a sufficient polling interval to avoid excessive CPU consumption while maintaining reasonable responsiveness. Once the shared memory file descriptor is obtained, the sender maps it into its address space using mmap with read-write permissions and the MAP_SHARED flag to ensure changes are visible across processes.

```
       // Wait until receiver has created shared memory (ready == 1)
    printf("[sender] Opening shared memory (waiting for receiver)...\n");
    int fd = -1;
    void *map = MAP_FAILED;
    struct shared_area *sh = NULL;

    for (int retry = 0; retry < 10000; retry++) {
        fd = shm_open(SHM_NAME, O_RDWR, 0666);
        if (fd >= 0) {
            map = mmap(NULL, sizeof(struct shared_area),
                       PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
            if (map != MAP_FAILED) {
                sh = (struct shared_area*)map;
                break;
            } else {
                close(fd);
                fd = -1;
            }
        }
        usleep(1000); // wait 1ms and retry
    }

    if (!sh) {
        fprintf(stderr, "[sender] failed to open: %s\n", strerror(errno));
        return 1;
    }
```

Listing 1.2. Open the shared memory region in sender

After successfully mapping shared memory, the sender verifies receiver readiness by polling the ready flag with atomic loads until it observes the value one (Listing 1.3). This synchronization ensures the receiver has completed initialization and calibration before transmission begins. Once ready, the sender prompts for a message string through standard input, validates the length to ensure it fits within the defined maximum, then stores the length in the shared msg_len field so the receiver knows how many bits to expect. A full fence and a small delay stabilize the system state before beginning bit transmission.

```
static inline void fence_all(void) {
    asm volatile("fence rw, rw" ::: "memory");}
static inline void tiny_delay(void) {
    for (volatile int i = 0; i < 400; i++);
    asm volatile("fence rw, rw" ::: "memory");
}
// confirm receiver presence
    int ready = 0;
    for (int i = 0; i < 5000; i++) {
        ready = __atomic_load_n(&sh->ready, __ATOMIC_SEQ_CST);
        if (ready == 1) break;
        usleep(1000); }
    if (ready != 1) {
        fprintf(stderr, "[sender] receiver not ready (ready=%d)\n", ready);
        munmap(map, sizeof(struct shared_area));
        close(fd);
        return 1;}
    printf("[sender] Sending message (%zu bytes): \"%s\"\n", len, msg_buf);
   __atomic_store_n(&sh->msg_len, (int)len, __ATOMIC_SEQ_CST);
    // small stabilization before starting
    fence_all();
    tiny_delay();
```

Listing 1.3. Verify Receiver Readiness to Accept Message

The main transmission loop iterates through each byte of the message, and within each byte iterates through bits in least-significant-bit-first order (Listing 1.4). For each bit, the sender first waits for its turn by spinning on the turn variable until it reads zero, indicating the receiver has completed its measurement and the sender may proceed. The sender stores the bit value for debugging, then executes the appropriate cache operation: if the bit is one, prefetch_w_ptr brings the target line into cache and an additional volatile access touches the data to reinforce cached state; if the bit is zero, cbo_flush_ptr evicts the line and a second flush provides robustness against incomplete evictions or timing races where the line might be brought back due to hardware prefetch. A full fence ensures cache operations complete, followed by tiny_delay that inserts a busy-wait loop allowing cache state changes to propagate through all cache levels. Finally, the sender atomically sets turn to one, signaling the receiver to measure, and the loop continues with the next bit. Upon completing all bits, the sender sets the done flag to indicate transmission completion.

RISC-V defines cache management through optional standard extensions rather than mandating specific instructions in the base ISA. This modular design reflects RISC-V's philosophy of supporting diverse implementation points from minimal embedded systems without caches to high-performance servers with sophisticated cache hierarchies.

```
    // For each byte, send 8 bits LSB-first.
    for (size_t b = 0; b < len; b++) {
        unsigned char c = (unsigned char)msg_buf[b];
        for (int k = 0; k < 8; k++) {
            int bit = (c >> k) & 1; // LSB-first
            // wait until it's our turn (turn == 0)
            while (__atomic_load_n(&sh->turn, __ATOMIC_SEQ_CST) != 0) {
                // optional: check receiver alive
                usleep(50);}
            // store for debug
            __atomic_store_n(&sh->bit_value, bit, __ATOMIC_SEQ_CST);
            if (bit) {
                prefetch_w_ptr(sh->target_line);
                // ensure it's touched
                volatile uint8_t tmp = sh->target_line[0]; (void)tmp;}
            else {
                cbo_flush_ptr(sh->target_line);
                // a second flush to be extra robust
                cbo_flush_ptr(sh->target_line);}
            fence_all();
            tiny_delay();
            // signal receiver to measure
            __atomic_store_n(&sh->turn, 1, __ATOMIC_SEQ_CST);
        } }
    __atomic_store_n(&sh->done, 1, __ATOMIC_SEQ_CST);
    printf("[sender] Finished sending.\n");
```

Listing 1.4. Main transmission loop

For standard RISC-V implementations supporting the Zicbom (Cache Block Management) extension and Zicbop (Cache Block Prefetch) extensions such as gem5 configured with these features, the prefetch function uses inline assembly to emit the prefetch.w instruction (Listing 1.5). The implementation first moves the pointer argument to a temporary register, then executes prefetch.w with that register as the base address. The memory clobber informs the compiler that this assembly sequence may access memory, preventing incorrect optimizations. A memory fence follows to ensure the prefetch completes before subsequent operations. Similarly, the flush function emits cbo.flush instruction using the same pattern of moving the pointer to a temporary register then executing the cache operation with appropriate memory barriers.

```
/* PREFETCH (prefetch_w_ptr) */
// Zicbop supported: use prefetch.w
static inline void prefetch_w_ptr(void *p) {
    asm volatile(
        "mv t0, %0\n\t"
        "prefetch.w 0(t0)\n\t"
        : : "r"(p) : "t0", "memory"
    );
    asm volatile("fence rw, rw" ::: "memory");
}
// Zicbom supported: use cbo.flush
static inline void cbo_flush_ptr(void *p) {
    asm volatile(
        "mv t0, %0\n\t"
        "cbo.flush 0(t0)\n\t"
        : : "r"(p) : "t0", "memory"
    );
    asm volatile("fence rw, rw" ::: "memory");
}
```

Listing 1.5. Prefetching and Cache Flushing using Zicbom And Zicbop

The T-Head C910 processor core deployed in BeagleV development boards and other commercial products implements custom cache instructions (Listing 1.6). The DCACHE.CIVA instruction (encoded as 0x278800b) invalidates data cache lines by virtual address, providing similar functionality to cbo.flush but with vendor-specific semantics. For prefetching, the C910 lacks a dedicated instruction analogous to prefetch.w from Zicbop. Instead, software must trigger hardware prefetch mechanisms through load operations, effectively emulating prefetch hints. This approach accommodates assemblers that may not recognize the vendor-specific mnemonic.

```
#ifdef C910
/* PREFETCH (prefetch_w_ptr) */
// C910: emulate with load (triggers HW prefetch)
static inline void prefetch_w_ptr(void *p) {
    asm volatile(
        "ld t0, 0(%0)\n\t"
        : : "r"(p) : "t0", "memory"
    );
    asm volatile("fence rw, rw" ::: "memory");
}
/* Cache FLUSH */
static inline void cbo_flush_ptr(void *p) {
    asm volatile(
        "xor a7, a7, a7\n"
        "add a7, a7, %0\n"
        ".long 0x278800b\n"   // DCACHE.CIVA a7
        : : "r"(p) : "a7", "memory"
    );
    asm volatile("fence rw, rw" ::: "memory");
}
```

Listing 1.6. Prefetching and Cache Flushing for C910 core

3.5 Receiver/Spy Implementation Details

The receiver process is initialized by creating or opening the shared memory object with both create and read-write flags, ensuring the region exists regardless of whether this is the first invocation (Listing 1.7). The ftruncate call resizes the shared memory to exactly the size of the shared_area structure, allocating backing storage for all fields. The receiver maps this region into its address space identically to the sender, obtaining a pointer to the shared structure that both processes will reference.

```
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (fd < 0) { perror("shm_open"); return 1; }

    if (ftruncate(fd, sizeof(struct shared_area)) < 0) {
        perror("ftruncate"); close(fd); return 1;
    }

    void *map = mmap(NULL, sizeof(struct shared_area),
                     PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (map == MAP_FAILED) {
        perror("mmap"); close(fd); return 1;
    }
    struct shared_area *sh = (struct shared_area*)map;
```

Listing 1.7. Creating shared memory object

Initialization proceeds by atomically storing initial values to all control fields: ready is set to one to signal the sender that initialization is complete, turn is set to zero to indicate the sender may act first, and other fields are zeroed (Listing 1.8). The target_line array is filled with a pattern (0xAA) to ensure consistent memory content during calibration. A full fence ensures all initialization completes before calibration begins.

```
        // initialize control fields (use atomic stores)
    __atomic_store_n(&sh->ready, 1, __ATOMIC_SEQ_CST);
    __atomic_store_n(&sh->turn, 0, __ATOMIC_SEQ_CST);
    __atomic_store_n(&sh->bit_value, 0, __ATOMIC_SEQ_CST);
    __atomic_store_n(&sh->msg_len, 0, __ATOMIC_SEQ_CST);
    __atomic_store_n(&sh->done, 0, __ATOMIC_SEQ_CST);

    // init cache line content
    memset(sh->target_line, 0xAA, CACHELINE);
    fence_all();
```

Listing 1.8. Initialization of Atomic variables

The calibration phase establishes a threshold to distinguish cached from evicted memory accesses (Listing 1.9).

```
// rdcycle
static inline uint64_t rdcycle64(void) {
    uint64_t v;
    asm volatile("rdcycle %0" : "=r"(v));
    return v; }
// measure one access in cycles
static inline uint64_t measure_access_cycles(void *p) {
    fence_all();
    uint64_t t0 = rdcycle64();
    volatile uint8_t tmp = *(volatile uint8_t*)p;
    (void)tmp;
    uint64_t t1 = rdcycle64();
    fence_all();
    return t1 - t0;
}
    // calibration: gather cached and evicted samples
    const int trials = 80;
    uint64_t sum_cached = 0, sum_evicted = 0;

    // Ensure cached samples
    for (int i = 0; i < 5; i++) {
        volatile uint8_t tmp = sh->target_line[0]; (void)tmp;
    }
    fence_all();
    for (int i = 0; i < trials; i++)
    sum_cached += measure_access_cycles(sh->target_line);
    // Evicted samples
    for (int i = 0; i < 5; i++) {cbo_flush_ptr(sh->target_line); fence_all();}
    for (int i = 0; i < trials; i++) {
        sum_evicted += measure_access_cycles(sh->target_line);
        cbo_flush_ptr(sh->target_line);
        fence_all(); }

    uint64_t avg_c = sum_cached / trials;
    uint64_t avg_e = sum_evicted / trials;
    uint64_t threshold = (avg_c + avg_e) / 2;
    if (threshold <= avg_c) threshold = avg_c + 1;
    printf("[receiver] Calibration done: threshold=%llu\n",
           (unsigned long long)threshold);
```

Listing 1.9. Calibration of Threshold

Precise timing measurement is essential for cache timing attacks and covert channels. RISC-V provides the rdcycle instruction in the base user-level ISA to read a hardware cycle counter that increments with each processor clock cycle. The receiver repeatedly measures access latencies to a target cache line using the rdcycle instruction, first ensuring the line is cached, then after flushing it. For cache timing analysis, rdcycle provides the fine-grained resolution necessary to distinguish between cached accesses (5-15 cycles typical) and uncached accesses (100+ cycles typical). Memory fences ensure that memory operations complete in program order and are not reordered by the processor pipeline or memory subsystem. Latencies are averaged over multiple trials to reduce noise, and the threshold is set as the midpoint between cached and evicted averages. This adaptive procedure accounts for variations across cache hierarchies, system load, and different RISC-V implementations, providing a reliable decision boundary for decoding transmitted bits.

After calibration, the receiver waits for the sender to indicate message length by polling msg_len until it reads a non-zero value (Listing 1.10).

```
    printf("[receiver] Waiting for sender to message (msg_len > 0)...\n");
    int msg_len = 0;
    while (1) {
        msg_len = __atomic_load_n(&sh->msg_len, __ATOMIC_SEQ_CST);
        if (msg_len != 0) break;
        int done = __atomic_load_n(&sh->done, __ATOMIC_SEQ_CST);
        if (done) { fprintf(stderr, "[receiver] no msg_len\n"); break; }
        usleep(1000); }
    if (msg_len <= 0 || msg_len > MESSAGE_MAX_LEN) {
        fprintf(stderr, "[receiver] invalid msg_len=%d\n", msg_len);
        munmap(map, sizeof(struct shared_area));
        close(fd);
        return 1;}
    printf("[receiver] Receive %d bytes (%d bits)...\n", msg_len, msg_len*8);
    int total_bits = msg_len * 8;
    int *bits = calloc(total_bits, sizeof(int));
    if (!bits) {
    perror("calloc"); munmap(map, sizeof(struct shared_area)); close(fd);
    return 1; }
    for (int i = 0; i < total_bits; i++) {
        // wait for sender to signal (turn == 1)
        while (__atomic_load_n(&sh->turn, __ATOMIC_SEQ_CST) != 1) {
            // check if sender died
            int done = __atomic_load_n(&sh->done, __ATOMIC_SEQ_CST);
            if (done) break;
            usleep(100); }

        fence_all();
        uint64_t access = measure_access_cycles(sh->target_line);
        int bit = (access < threshold) ? 1 : 0;
        bits[i] = bit;

        __atomic_store_n(&sh->bit_value, bit, __ATOMIC_SEQ_CST);

        printf("[receiver] bit %3d: measured=%3llu => %d\n",
        i+1, (unsigned long long)access, bit);

        // give control back to sender
        __atomic_store_n(&sh->turn, 0, __ATOMIC_SEQ_CST); }
```

Listing 1.10. Synchronization in receiver

This synchronization ensures both processes agree on how many bits to transmit. The receiver allocates an array to store decoded bits, then enters the main reception loop. For each bit, the receiver waits for its turn by spinning on the turn variable until it reads one, indicating the sender has completed its cache operation. A full fence ensures memory ordering, then the receiver measures access latency by calling the measure_access_cycles function. Bit decoding compares measured latency to threshold: values below threshold decode as one (cached), values above threshold decode as zero (evicted). The decoded bit is stored in the array and optionally written to bit_value for debugging. The receiver then atomically sets turn to zero, signaling the sender to proceed with the next bit. After receiving all bits, the receiver reconstructs the message by iterating through bits in groups of eight (Listing 1.11). Each byte is assembled by shifting bits into position according to least-significant-bit first convention: bit zero becomes the LSB, bit one is shifted left once, bit two shifted twice, and so on through bit seven forming the MSB. The assembled bytes form the ASCII message string, which is null-terminated and printed. The receiver sets the done flag, frees allocated memory, and enters the outer infinite loop that reinitializes for the next message. This design enables continuous operation receiving multiple messages without restarting.

```
    // reconstruct message LSB-first per byte
    char *out = calloc(msg_len + 1, 1);
    if (!out) { perror("calloc2"); free(bits);
    munmap(map, sizeof(struct shared_area)); close(fd); return 1; }
    for (int b = 0; b < msg_len; b++) {
        unsigned char v = 0;
        for (int k = 0; k < 8; k++) {
            int bit = bits[b*8 + k];
            v |= (bit & 1) << k; }
        out[b] = (char)v;
    }
    out[msg_len] = '\0';
    printf("[receiver] Recovered message: \"%s\"\n", out);
    __atomic_store_n(&sh->done, 1, __ATOMIC_SEQ_CST);
    free(bits);
    free(out); }
    munmap(map, sizeof(struct shared_area));
    close(fd);
    printf("[receiver] Finished.\n");
```

Listing 1.11. Message reconstruction in receiver

3.6 Compilation and Deployment

Compiling the implementation requires different flags depending on the target platform. For gem5 full-system simulation with standard RISC-V extensions, the march flag specifies rv64gc_zicbom_zicbop, indicating 64-bit RISC-V with the general compressed extensions plus cache block management and prefetch extensions (Listing 1.12). The static flag produces statically-linked binaries that contain all necessary libraries, essential for execution in minimal simulated environments where dynamic linking infrastructure may not be available. The pthread flag links POSIX threading libraries required for atomic operations support.

```
/opt/riscv/bin/riscv64-unknown-linux-gnu-gcc -march=rv64gc_zicbom_zicbop \
    sender.c -o sender -static -pthread

/opt/riscv/bin/riscv64-unknown-linux-gnu-gcc -march=rv64gc_zicbom_zicbop \
    receiver.c -o receiver -static -pthread
```

Listing 1.12. Compilation using RISC-V extensions for gem5 simulator

For T-Head C910 hardware, compilation uses the DC910 preprocessor definition to activate vendor-specific code paths (Listing 1.13). The march flag specifies only rv64gc without Zicbom or Zicbop since the C910 does not support these standard extensions. Other flags remain identical, producing statically-linked binaries that execute on the physical board. The conditional compilation ensures the same source files generate correct binaries for both platforms, demonstrating the effectiveness of the abstraction layer.

```
gcc -DC910 -march=rv64gc \
    sender.c -o sender -static -pthread

gcc -DC910 -march=rv64gc \
    receiver.c -o receiver -static -pthread
```

Listing 1.13. Compilation on Beagle-V Ahead Hardware

4 Experimental Evaluation and Results

4.1 Experimental Platforms

We evaluate our covert channel implementation on two distinct RISC-V platforms that represent different points in the design space. The gem5 full-system simulator provides a controlled and reproducible environment for security analysis with detailed visibility into microarchitectural events [4,25]. We use the gem5 model of the SiFive U54 core [37]. Our configuration simulates a 64-bit RISC-V processor supporting the RV64GC ISA together with the Zicbom and Zicbop extensions. The cache hierarchy consists of separate 32 KB L1 instruction and data caches with four-way associativity and 64 byte cache lines, and a unified 256 KB L2 cache with eight-way associativity. The simulated system boots a Linux kernel with POSIX shared memory support, which provides a realistic software environment for isolation and inter-process communication.

The BeagleV-Ahead development board serves as our physical RISC-V platform. It features the T-Head C910 processor, which is a high-performance out-of-order quad-core running at 2 GHz. The C910 implements a 64 KB four-way instruction cache and a 64 KB four-way data cache, both with 64 byte lines. All cores share a 1 MB L2 cache [3]. The board runs a Linux distribution with a 5.x series kernel, which offers a complete POSIX environment. Physical hardware introduces natural variability caused by system load, operating system scheduling, interrupt handling, and other external effects. These sources of variability are absent in simulation, and evaluating our covert channel in this setting is essential for demonstrating its robustness.

4.2 Methodology and Measurements

Our evaluation focuses on the reliability and timing characteristics of the covert channel under varying conditions. For each experimental trial, we execute the receiver process to initialize shared memory and perform calibration, then launch the sender process to transmit a message of known content. The receiver decodes the transmitted bits and reconstructs the message, allowing us to compare the recovered string against the original to compute the bit error rate. We record calibration results, including average cached latency, average evicted latency, and computed threshold, to understand timing characteristics on each platform. During transmission, we log measured latencies for each bit to analyze the separation between cached and evicted access times and identify any marginal cases where latency falls near the threshold.

Test cases include messages of varying lengths from short single-character strings to maximum-length defined (e.g; 128-byte) messages to assess whether the error rate depends on message length. We test with different ASCII content, including alphanumeric characters, punctuation, and common words, to ensure the encoding scheme handles all valid byte values correctly. Multiple repetitions of each test case provide statistical confidence in reliability measurements. We deliberately introduce challenging conditions, including concurrent system load from background processes to evaluate robustness against cache pollution, and rapid repeated transmissions to stress the synchronization protocol.

4.3 Results

Bit error rates under controlled conditions remain extremely low, typically zero errors across hundreds of transmitted bytes on both platforms. The combination of statistical calibration, careful synchronization through atomic variables, and robust encoding techniques (double-flushing for zero bits, touch after prefetch for one bits) yields reliable communication. Occasional errors observed during testing typically correlate with specific system events such as timer interrupts or context switches that perturb cache state between the sender's operation and receiver's measurement. These errors remain rare enough that forward error correction could easily achieve error-free communication.

The platform abstraction layer successfully enables the same protocol to function on both gem5 (U54 core) [25] with standard instructions and BeagleV Ahead (C910) [3] with custom instructions. This portability validates the design approach of separating platform-specific primitives from protocol logic. The conditional compilation mechanism cleanly handles the differences in cache operation implementation while maintaining identical high-level behavior. This demonstrates that security analysis tools for RISC-V can accommodate implementation diversity through appropriate abstraction layers.

5 Countermeasures and Defenses

Several defense strategies could mitigate cache timing covert channels on RISC-V platforms, each with distinct tradeoffs between security, performance, and deployment complexity.

Restricting user-space access to cache management instructions provides a direct defense by preventing unprivileged processes from explicitly manipulating cache state. Since cache extensions are optional in RISC-V, security-critical systems can simply omit Zicbom and Zicbop support [34], [33], eliminating explicit cache management capabilities that enable covert channels. This approach works well for embedded systems or specialized processors where performance optimization through cache management is less critical. However, omitting extensions sacrifices functionality that general-purpose computing systems may require, and attackers can still exploit implicit cache state changes through memory access patterns even without explicit cache instructions. Vendor-specific cache instructions like those in the T-Head C910 [3] complicate ecosystem-wide security analysis. Each proprietary extension requires separate evaluation, and security properties may not generalize across vendors.

Hardware cache partitioning schemes assign dedicated cache ways or sets [40],[8] to different security domains, preventing cross-domain cache state observation. Intel's Cache Allocation Technology on x86 provides runtime cache partitioning, and similar mechanisms could be implemented in RISC-V processors. Partition-based defenses guarantee that one process cannot evict another's cache lines, eliminating the fundamental mechanism underlying cache covert channels. However, cache partitioning requires significant hardware complexity including additional tag bits, partition enforcement logic, and potentially separate eviction policies per partition. The performance impact depends on partition sizes and access patterns, with worst-case scenarios where rigid partitioning prevents effective cache utilization.

Randomized cache replacement policies break the deterministic timing relationships that covert channels exploit. Instead of using predictable LRU or pseudo-LRU replacement, caches could employ randomized selection when choosing eviction victims. This introduces uncertainty in whether a flush operation successfully evicts a line and whether a subsequent access observes a hit or miss. Wang and Lee proposed various randomization schemes that maintain reasonable cache performance while disrupting timing channels [40]. However, randomization typically reduces average cache hit rates, impacting performance for all applications. The security-performance tradeoff depends on the degree of randomization and cache size [32], [20].

Fuzzy time approaches reduce timing channel capacity by degrading timer precision or injecting noise into timing measurements [14]. The operating system could reduce rdcycle granularity, add random delays to the returned values, or rate-limit access to timing instructions. This defense raises the noise floor, making it harder to reliably distinguish cached from evicted access latencies. However, sophisticated attackers can often overcome noise through statistical averaging over many measurements. Hu's analysis showed that substantial noise

is required to meaningfully reduce channel capacity, and such noise levels impact legitimate timing-sensitive applications including multimedia processing, network protocol implementations, and performance profiling tools [26]. The tradeoff between security and functionality must be carefully balanced based on deployment requirements.

Statistical detection and anomaly-based monitoring offer deployment flexibility without modifying hardware or restricting instruction access. Security monitoring systems can leverage hardware performance counters to detect suspicious patterns of cache behavior including abnormally high flush rates, unusual cache miss patterns, or correlated timing between processes. Machine learning classifiers trained on known covert channel traffic can identify characteristic access patterns that distinguish covert communication from legitimate application behavior [6,28]. However, detection systems face challenges including the need for platform-specific training data, the potential for false positives that impact legitimate applications, and the computational overhead of continuous monitoring. Attackers aware of detection systems can adapt their techniques to evade signature-based detection through randomization, rate limiting, or mimicking legitimate access patterns.

Detection and mitigation of cache covert channels on RISC-V requires comprehensive strategies addressing multiple system layers. Hardware approaches including cache partitioning and randomized replacement provide strong isolation at the cost of implementation complexity and potential performance impact. Software approaches including instruction access restriction and timing noise injection offer deployment flexibility but may impact legitimate performance-sensitive applications. Statistical detection through performance counter monitoring enables runtime identification of covert channel activity but requires platform-specific tuning and sophisticated analysis. The optional nature of RISC-V cache extensions enables security-critical systems to omit these features entirely, though at the cost of performance optimization capabilities.

6 Discussion and Future Work

Our covert channel implementation enables several realistic attack scenarios in deployed RISC-V systems. In the malware coordination scenario, separate malware components executing as different processes could exfiltrate sensitive data without triggering network intrusion detection systems, similar to earlier demonstrations of cache-based covert communication [31,47]. One component could acquire cryptographic keys or passwords and transmit them bit by bit to another component that has network access for exfiltration. Because communication occurs through cache state rather than files, system calls, or network packets, the channel leaves no conventional audit trail [24]. In containerized environments, where processes run in different containers but share the same physical host, microarchitectural channels can circumvent namespace and cgroup isolation guarantees [44]. Covert channels, therefore, allow information leakage across container security boundaries even when higher-level isolation holds.

The covert channel may also serve as an exfiltration mechanism for data obtained from other side-channel attacks, enabling multi-stage attacks. An attacker could first perform a cache timing attack to recover partial cryptographic key material and then use the covert channel to transmit this information to another compromised process, enabling composite attacks that evade defenses focused on isolated threat models [5]. In cloud environments, cross-VM cache channels have previously been shown to bypass hypervisor isolation [35], suggesting that the principles demonstrated in our single-OS implementation extend to multi-tenant systems lacking cache partitioning.

But our implementation also exhibits several limitations that represent opportunities for future improvement. First, the protocol assumes cooperative processes that follow the turn-taking discipline enforced through atomic variables. An advanced adversarial scenario might involve non-cooperative processes that cannot explicitly coordinate, requiring the covert channel to operate through purely implicit timing observation without shared variables [47]. Second, the protocol lacks forward error correction or automatic retry mechanisms. Bit errors directly corrupt the decoded message without detection or recovery. Adding simple parity bits or checksums would enable error detection, while more sophisticated codes could enable correction.

Future research should explore several promising directions. Implementing forward error correction mechanisms would improve channel reliability and enable communication under higher noise conditions. Developing practical detection systems based on hardware performance counter monitoring would provide deployable defenses, requiring machine learning models trained on diverse RISC-V platforms [6]. Analyzing additional RISC-V implementations beyond gem5 and C910 would characterize timing channel properties across the broader ecosystem, including embedded processors and high-performance designs. Formal verification techniques could provide mathematical proofs about covert channel capacity or countermeasure effectiveness, complementing empirical evaluation.

7 Conclusion

This paper presented a comprehensive analysis of cache and prefetcher-based covert channels on RISC-V architectures through the design, implementation, and evaluation of a practical timing channel that exploits cache state. The covert channel achieves reliable communication by encoding binary information through selective cache prefetching and flushing operations, with the receiver decoding bits by measuring memory access latency and comparing against calibrated thresholds. Careful protocol design, including statistical calibration, atomic variable synchronization, and robustness techniques, enables consistent operation across varying system conditions. A significant contribution of our paper is demonstrating portability across heterogeneous RISC-V implementations through platform abstraction that supports both standardized extensions and vendor-specific instructions.

As RISC-V adoption accelerates in security-critical domains, including trusted execution environments and cryptographic accelerators, understanding and mitigating microarchitectural covert channels becomes increasingly vital. Our paper demonstrates practical cross-core exploitation techniques, informing both secure system design and the development of effective defenses. The open nature of RISC-V presents unique opportunities to integrate security features from inception rather than retrofitting them onto mature architectures, but realizing these opportunities requires sustained effort across the diverse RISC-V community. We hope this work contributes to building more secure RISC-V systems and catalyzes further security research on this important emerging architecture.

References

1. Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al.: The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 (2016)
2. Austa, C., Mühlberg, J.T., Dricot, J.M.: Systematic assessment of cache timing vulnerabilities on risc-v processors. In: European Symposium on Research in Computer Security (ESORICS) 2025, Lecture Notes in Computer Science, vol. 16055. pp. 23–42 (2025). https://doi.org/10.1007/978-3-032-07894-0_2
3. BeagleBoard.org Foundation: BeagleV-Ahead RISC-V Board Documentation. https://www.beagleboard.org/boards/beaglev-ahead (2024)
4. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. In: ACM SIGARCH Computer Architecture News. vol. 39, pp. 1–7. ACM (2011)
5. Canella, C., Van Bulck, J., Schwarz, M., et al.: A systematic evaluation of transient execution attacks and defenses. In: USENIX Security (2019)
6. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. In: Applied Soft Computing. vol. 49, pp. 1162–1174. Elsevier (2016)
7. Dessouky, G., Frassetto, T., Sadeghi, A.R.: Hardfails: Insights into software-exploitable hardware bugs. In: 29th USENIX Security Symposium. pp. 213–230 (2020)
8. Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., Ponomarev, D.: Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. ACM Transactions on Architecture and Code Optimization **8**(4), 1–21 (2012)
9. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**(1), 1–27 (2018)
10. Gerlach, L., Weber, D., Zhang, R., Schwarz, M.: A security risc: microarchitectural attacks on hardware risc-v cpus. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 2321–2338. IEEE (2023)
11. Gruss, D., Maurice, C., Mangard, S.: Prefetch side-channel attacks: Bypassing smap and kernel aslr. In: Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS). pp. 368–379. ACM (2016)
12. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+flush: A fast and stealthy cache attack. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 279–299. Springer (2016)

13. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, CA, 6 edn. (2017)
14. Hu, W.M.: Reducing timing channels with fuzzy time. Journal of Computer Security **1**(3-4), 233–254 (1992)
15. Inci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. In: IACR Cryptology ePrint Archive (2015)
16. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA). pp. 252–263. ACM (1997)
17. Khan, M., Mushtaq, M., Pacalet, R., Apvrille, L.: Evaluating kaslr break on risc-v using gem5: Microarchitectural side-channel analysis of page-table walks. In: EICC 2025: European Interdisciplinary Cybersecurity Conference (2025), `https://hal.archives-ouvertes.fr/hal-05097207`
18. Khan, M., Mushtaq, M., Pacalet, R., Apvrille, L.: Evict+spec+time on risc-v: Gem5-based implementation and microarchitectural analysis. In: 28th Euromicro Conference Series on Digital System Design (DSD). Salerne, Italy (2025), `https://hal.archives-ouvertes.fr/hal-05176064`
19. Khan, M., Mushtaq, M., Pacalet, R., Apvrille, L.: Prototyping custom hardware performance counters in gem5 simulator: A framework for risc-v side-channel attack assessment. In: 25th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS) (2025), `https://hal.archives-ouvertes.fr/hal-05097250`
20. Kim, T., Peinado, M., Mainar-Ruiz, G.: Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In: 21st USENIX Security Symposium. pp. 189–204 (2012)
21. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy. pp. 1–19. IEEE (2019)
22. Lampson, B.W.: A note on the confinement problem. Communications of the ACM **16**(10), 613–615 (1973)
23. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium. pp. 973–990 (2018)
24. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE Symposium on Security and Privacy (2015)
25. Lowe-Power, J.: Gem5 documentation (Sat 29 June 2024), [Online]. Available: `https://www.gem5.org/documentation/`
26. Martin, R., Demme, J., Sethumadhavan, S.: Quantification of the effect of architectural and application characteristics on arm cache side channels. arXiv preprint arXiv:1204.0497 (2012)
27. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Hello from the other side: Ssh over robust cache covert channels in the cloud. In: Network and Distributed System Security Symposium (NDSS) (2015)
28. Mushtaq, M., Akram, A., Bhatti, M.K., Chaudhry, M., Lapotre, V., Gogniat, G.: Whisper: A tool for run-time detection of side-channel attacks. In: IEEE Access. vol. 8, pp. 83871–83900. IEEE (2020)
29. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: Topics in Cryptology–CT-RSA 2006. pp. 1–20. Springer (2006)

30. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design RISC-V Edition. Morgan Kaufmann (2020)
31. Percival, C.: Cache missing for fun and profit. In: BSDCan (2005)
32. Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing digital side-channels through obfuscated execution. In: 24th USENIX Security Symposium. pp. 431–446 (2015)
33. RISC-V International: Risc-v cache-block prefetch instructions, version 1.0. Tech. rep., RISC-V International (2021), ratified specification
34. RISC-V International: Risc-v cache management operation isa extensions, version 1.0. Tech. rep., RISC-V International (2021), ratified specification
35. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: ACM CCS (2009)
36. Schrammel, D., Weiser, S., Schwarzl, M., Schwarz, M., Mangard, S.: Donky: Domain keys–efficient in-process isolation for risc-v and x86. In: 29th USENIX Security Symposium. pp. 1677–1694 (2020)
37. SiFive: SiFive U54 Core Complex Manual (2021), `https://starfivetech.com/uploads/u54_core_complex_manual_21G1.pdf`
38. T-Head Semiconductor: XuanTie C910 Processor Datasheet. T-Head Semiconductor Co., Ltd. (2020)
39. Thomas, F., Arribas, E.G., Hetterich, L., Weber, D., Gerlach, L., Zhang, R., Schwarz, M.: Riscover: Automatic discovery of user-exploitable architectural security vulnerabilities in closed-source risc-v cpus. In: Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS). Taipei, Taiwan (2025)
40. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. ACM SIGARCH Computer Architecture News **35**(2), 494–505 (2007)
41. Waterman, A., Lee, Y., Patterson, D.A., Asanovic, K.: The risc-v instruction set manual, volume i: User-level isa, version 2.0. In: Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley (2014)
42. Wistoff, N., Schneider, M., Gürkaynak, F.K., Benini, L., Heiser, G.: Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. CoRR **abs/2005.02193** (2020), `https://arxiv.org/abs/2005.02193`
43. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In: 21st USENIX Security Symposium. pp. 159–173 (2012)
44. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In: IEEE Symposium on Security and Privacy (2015)
45. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of l2 cache covert channels in virtualized environments. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 29–40 (2011)
46. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: USENIX Security Symposium. pp. 719–732 (2014). `https://doi.org/10.5555/2671225.2671271`
47. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-vm side channels and their use to extract private keys. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 305–316 (2012)