LoongLeak: Architectural Cross-Privilege-Boundary Data Leakage on LoongArch CPUs

Lorenz Hetterich, Tristan Hornetz, Fabian Thomas, Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Recent research has revealed architectural vulnerabilities in widely deployed CPUs that break confidentiality and integrity. While x86-64, Arm, and RISC-V CPUs have received significant scrutiny, Loongson processors, which are built on the LoongArch ISA and are widely used in Chinese infrastructure, have not. This lack of analysis leaves a critical blind spot in global security, especially as China phases out foreign CPUs.

In this paper, we discover and analyze LoongLeak, a novel architectural vulnerability affecting multiple Loongson CPUs. LoongLeak exploits how 4-byte floating-point loads return stale bytes from the L1 data cache, enabling unprivileged attackers to leak confidential data across security domains, such as the kernel or hypervisor. We demonstrate that this leakage is architectural and requires no timing or side channels, giving attackers fine-grained control over cache sets and offsets. Our case studies include recovering full-disk AES keys from the kernel, partial root password hashes from user-space, and bypassing traditional software defenses such as ASLR and stack canaries, all within seconds. LoongLeak can be exploited from unprivileged user space, containers, or virtual machines. We explore software-based mitigations, including floating-point emulation, which incurs an overhead of 10 $\times$ to 21 $\times$ for floating-point heavy applications, and flushing the L1 data cache on kernel to userspace transitions in conjunction with turning off SMT threads. While these mitigations can effectively mitigate LoongLeak in software, a long-term solution requires hardware fixes.

1 Introduction

Modern computer security is built on the assumption that well-written software is executed on trustworthy CPUs. However, this assumption changed in early 2018 with the publication of Meltdown [22] and Spectre [18], showing that out-of-order and speculative execution, commonly referred to as *transient execution* [6], can expose secrets across protection boundaries. Multiple follow-up works [2, 5, 25, 32, 38, 40] have exploited similar transient effects, leading to a vast landscape of so-called transient execution attacks [6].

Architectural CPU vulnerabilities are more fundamental. When a vulnerability manifests in the retired instruction stream rather than only during transient execution, no additional side channel is required, and mitigations that protect against transient-execution attacks are typically ineffective. Such mitigations include fencing [6, 18], flushing [32, 40], or avoiding specific instructions [37, 39]. Well-known cases of architectural vulnerabilities, such as CacheWarp [43], GhostWrite [36], ÆPICLeak [4], Reptar [28], Zenbleed [29], or EntrySign [9], forced vendors to revise silicon or ship complex microcode patches, illustrating the high cost of discovering such problems. While these vulnerabilities pose a significant threat, discovering them early reduces the risk that malicious actors can exploit them over longer periods.

Unfortunately, the search for such vulnerabilities has focused almost exclusively on the instruction sets that dominate Western servers, desktops, and smartphones: x86-64, Arm, and, more recently, RISC-V. The Chinese domestic market, however, increasingly relies on the *LoongArch* ISA and its Loongson CPUs in desktops, laptops, and government infrastructure [35]. For the latter, China also plans to ban non-Chinese CPUs.[1] To date, Loongson CPUs have attracted little independent scrutiny, leaving a significant blind spot in the global security picture.

In this paper, we narrow this gap. We discover LoongLeak, a previously unexplored architectural leakage source on multiple Loongson CPUs. LoongLeak exploits that the upper 4-byte of a 8-byte register are "uncertain" after a 4-byte floating-point load according to the Loongarch manual [20]. Furthermore, the LSX and LASX vector extensions increase the size of floating registers to 16 and 32 bytes, leading to more uncertain bytes. We show that those "uncertain" bytes are not arbitrary, but that they return stale bytes directly from the L1 data cache. By reverse-engineering this behavior, we discover that an attacker can get fine-grained control over the

[1] https://www.reuters.com/world/china/china-blocks-use-intel-amd-chips-government-computers-ft-reports-2024-03-24/

Table 1: Comparison of architectural CPU vulnerabilities including their threat model, impact, and control over the affected entity.

Vulnerability	Unpriv.	Breaks	Control
LoongLeak	✓	Confidentiality	Cache set and cache-line offset
ZenBleed	✓	Confidentiality	Vector register
ÆPICLeak	✗	Confidentiality	Cache-line offset
GhostWrite	✓	Integrity	Physical address
CacheWarp	✗	Integrity	Single store
EntrySign	✗	Integrity	Instructions
Reptar	✓	Availability	–

leaked data. Executing a carefully aligned load and preceding it with a priming access that brings the involved buffers into the desired state allows attackers to select which L1 data cache set, offset, and way are leaked.

LoongLeak can be exploited by an unprivileged attacker from user space, and from within containers and virtual machines. LoongLeak works across security domains, i.e., it can leak kernel and hypervisor memory, on the same logical core as well as across hyperthreads. Thus, the unprecedented control over the data to leak across security domains and the broad applicability make it fundamentally different than other architectural CPU vulnerabilities, as compared in Table 1. As the leakage is fully architectural, no side channel or timing measurement is required. Since the leakage originates from a simple load hitting the L1 data cache, LoongLeak is fast and achieves leakage rates beyond 300 MB/s.

To demonstrate the severity of LoongLeak, we present 3 case studies. We extract AES keys from the kernel that are used for disk encryption. This demonstrates that kernel secrets are fully exposed, even without SMT. We employ LoongLeak to read secrets that are used by high-privileged user-space processes running on an SMT sibling thread, such as the `/etc/shadow` root password hash. This case study demonstrates that even when LoongLeak is not used in a targeted manner, its enormous speed allows leaking partial password hashes within seconds. Finally, we demonstrate how LoongLeak can break traditional software defenses like ASLR and stack canaries. Even though LoongLeak itself affects confidentiality, this case study shows that LoongLeak can ease other attacks aiming to compromise the integrity of a system.

A short-term mitigation could be implemented as a kernel patch to trap and emulate floating-point instructions, removing the leakage. While this prevents exploitation, it comes with a non-negligible performance overhead. In the SPEC CPU 2017 floating-point benchmarks, we measure an average performance overhead of 10 × to 21 × when switching from hardware floating-point to software floating-point instructions. A kernel patch would introduce additional overhead through context switches. The overhead for SPEC CPU 2017 integer benchmarks is smaller at −3.8 % to 14.2 %. Alternatively, the L1 data cache can be evicted on kernel exit, leading to a smaller performance overhead of up to 1.4 %. On the Loongson 3A6000, which supports 2-way SMT, this mitigation is only effective when disabling half of the logical cores. Thus, ultimately, LoongLeak has to be fixed in hardware.

Contributions. The contributions of this paper are:

- We discover LoongLeak, a CPU vulnerability on the Loongson 3A5000 and 3A6000 CPUs that allows unprivileged attackers to leak data across security domains.
- We analyze LoongLeak, discovering that stale data originates in the L1 data cache, giving an attacker control over the cache set, line offset, and cache way for targeted leakage.
- We use LoongLeak to recover AES encryption keys, root credentials, and other high-value secrets across different security boundaries.
- We explore different mitigations for LoongLeak and estimate their overhead.

Outline. The remainder of the paper is structured as follows: Section 2 provides background on transient and architectural vulnerabilities. Section 3 describes LoongLeak and reverse-engineers its properties. Section 4 outlines applicable threat models and attack scenarios. Section 5 demonstrates how LoongLeak can be used to leak disk encryption keys, partial password hashes, and break software mitigations. Section 6 introduces possible countermeasures. Section 7 discusses LoongLeak and related work. Section 8 concludes.

Responsible Disclosure. We reported our findings, including the analysis, to Loongson on July 23, 2025. Loongson can reproduce our findings. To confirm our analysis and adequately address the issue, Loongson requested an embargo period of one year. Loongson later confirmed that the leakage stems from the L1 cache.

Availability. We will publicly release the artifacts on the first conference day to comply with the embargo period.

2 Background

In this section, we provide the background required for the rest of the paper, covering microarchitecture and architecture, the LoongArch architecture, caches, and architectural CPU vulnerabilities.

2.1 Microarchitecture and Architecture

The CPU architecture (instruction set architecture, ISA) defines instructions, registers, memory ordering, and the exception model that software can rely on. The microarchitecture is the actual implementation of the architecture: pipelines, buffers, caches, predictors, and other implementation choices that vendors are free to change so long as the architectural behavior does not change. These invisible microarchitectural structures are critical for the performance and power efficiency of a CPU. However, they have also been a frequent target of so-called microarchitectural attacks. These attacks typically rely on side channels, such as timing, to measure

the internal state of the microarchitectural elements and, from that, infer secrets.

LoongArch. Introduced by Loongson in 2021, LoongArch [20] is a RISC architecture with fixed-width 4-byte instructions, a three-privilege-level model (PLV0-2), and optional extensions including a floating-point unit (FPU) and two SIMD variants: LSX (16 bytes) and LASX (32 bytes). Feature bits in the Extension-Unit Enable (EUEN) control and status register allow firmware or the kernel to enable or mask those units at runtime. LoongArch supports a 32-bit version (LA32) and a 64-bit version (LA64). LA64 is binary compatible with LA32, and all processors we use in this paper support LA64. When we mention LoongArch in the remainder of this paper, we specifically refer to LA64.

Commercial off-the-shelf CPUs implementing LA64 are the Loongson 3A5000 (LA464) and the 3A6000 (LA664). These CPUs issue up to six instructions per cycle, execute out of order, and support multiple hardware page sizes (4 kB, 16 kB, 64 kB) with a default of 16 kB on most Linux builds. The 3A6000 implements simultaneous multithreading (SMT), but the 3A5000 does not. Linux distributions supporting these CPUs include Debian, Arch, and openSUSE.

2.2 Microarchitectural and Architectural CPU Attacks

Microarchitectural attacks exploit side effects in microarchitectural elements. Typical vectors are data-dependent latencies (cache state [23, 42], TLBs [11]), contention (port contention [1], bus contention [30]), or transient instructions that speculatively execute past checks [18, 22]. Timing or power differences [21] encode secret information that an attacker later recovers. Microarchitectural attacks, such as cache attacks or contention-based attacks, often persist because the root cause (e.g., caching) is essential for performance and cannot simply be disabled.

Architectural attacks are more severe. They arise when the committed state itself is wrong: A load returns stale data that is architecturally visible [4], or an instruction writes to logically inaccessible memory [36]. Examples include ÆPICLeak [4] (stale data read from the APIC), CacheWarp [43] (reverting stores to their previous value), or ZenBleed [29] (wrongly reverted speculation). Such flaws bypass the need for precise timing and can sometimes even be triggered from user space [28, 29, 36], forcing vendors to issue microcode workarounds or, as a long-term fix, redesign silicon.

2.3 CPU Caches

Modern CPUs bridge the speed gap between DRAM and the pipeline with a hierarchy of caches. Each level is split into *sets*, where every set contains several *ways* holding *lines* of fixed size. Caches are indexed by either the virtual or physical address of the data to be stored. The address is typically split into a *line offset*, a *set index*, and a *tag*. For typical cache implementations, the lower $\log_2(\text{line size})$ bits select a byte within the line, and the next $\log_2(\#\text{sets})$ bits choose the set. The upper bits are used as the tag. The CPU-specific replacement policy decides which way to evict on conflict.

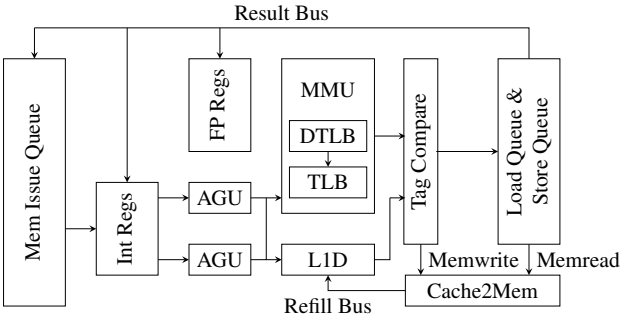

Figure 1: Overview of the Loongson 3A5000 memory subsystem. Adapted from the official documentation on the Loongson 3A5000 CPU [19].

LoongArch L1 Data Cache. The LA464/LA664 cores use a 64 kB, 4-way set-associative L1 data cache with 64-byte lines. The cache is divided into $\frac{64\,\text{KiB}}{4\,\text{ways}\times 64\,\text{B}} = 256$ sets. Addresses are split into 6 bits line offset ($2^6 = 64$ B), 8 bits set index, and the remaining bits as tag. As the page offset covers all index bits, the index is identical in virtual and physical addresses for 16 kB pages (14 offset bits). The choice of 16 kB pages compared to 4 kB used by x86 allows for a sizable 64 kB L1 data cache whilst only requiring 4 cache ways.

2.4 Memory Subsystem

Modern LoongArch cores follow the familiar split-load/store datapath of out-of-order CPUs, but implement it with a set of buffers that differs slightly from better-documented x86 designs. We focus on the LA464 (3A5000) and LA664 (3A6000) cores because both are affected by LoongLeak.

The relevant parts of the memory subsystem are depicted in Figure 1. Loads cause entries in the "Mem Issue Queue" to be created. This queue interfaces with the integer register file (Int Regs) presumably to access the register(s) required for address generation. This information then flows into one of two Address Generation Units (AGUs) for the Loongson 3A5000. These AGUs compute the full virtual address for the memory access. The computed virtual address is then fed in parallel into the Memory Management Unit (MMU) and the L1 data cache (L1D). The MMU translates the upper virtual address bits into a physical address using a first-level data Translation Lookaside Buffer (DTLB), and on a miss, a bigger second-level TLB (TLB) that is shared for data and code pages. The translated physical address is forwarded to a "Tag Compare" component. In parallel, the data cache can select

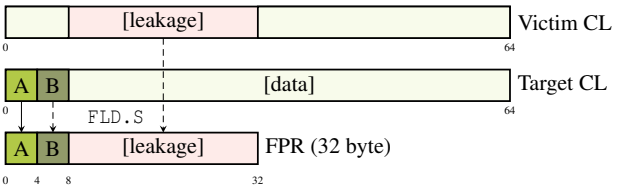

Figure 2: Overview of LoongLeak. If an `FLD.S` instruction with an 8-byte aligned destination hits the L1 cache, a total of 8 bytes are loaded from the destination. The remaining 24 bytes are loaded from a victim cache line. A previous priming load determines the cache set of the victim cache line, while the `FLD.S` destination sets the cache-line offset and the cache way. On the Loongson 3A5000, any offset from the victim cache line can be leaked, while on the 3A6000, only the second half of the victim cache line can be leaked.

the correct cache set based on address bits 6-13, which are the same between virtual and physical addresses. The L1 data cache forwards information about the stored tags for each way, possibly alongside stored data, to the "Tag Compare" component. This component can then select the correct way and forward the information to the "Load Queue & Store Queue" component. For memory writes, the "Tag Compare" component also connects to the component handling the higher levels of the memory hierarchy (Cache2Mem), likely to handle cache coherency. It is unclear whether the information forwarded to the "Load Queue & Store Queue" contains the actual data on a L1 data cache hit or the Load Queue does a targeted load from the data cache. For memory reads that miss the L1 data cache, the Load Queue issues a request to the "Cache2Mem" component that takes care of higher-level cache lookups. The resulting data is then fed into the L1 data cache using a "Refill Bus". Once a request in the Load Queue is fulfilled, the resulting data is forwarded using a so-called "Result Bus" that connects to both register files (integer and vector) as well as the "Mem Issue Queue" to signal that the load has completed.

3 LoongLeak

In this section, we introduce LoongLeak, an architectural CPU vulnerability in the Loongson 3A5000 and 3A6000 CPU that allows leaking stale data across security domains. Section 3.1 provides an overview. Section 3.2 outlines our fuzzing approach that discovers LoongLeak. Section 3.3 describes our experiments to reverse engineer the properties of LoongLeak.

3.1 LoongLeak Overview

On Loongson processors, we observe that "uncertain" bits in the target register of the unprivileged `FLD.S` instruction contain values that occasionally reveal data from previous

```
1  ; Memory barrier to ensure ordering
2  DBAR 0
3  ; Load that determines cache set of leakage
4  XVLD xr0, %[addr_a], 0
5  DBAR 0
6  ; 4-byte load leaks in upper register bytes
7  FLD.S f0, %[addr_b], 0
8  ; Store entire 32-byte register to memory
9  ;  to unveil "uncertain" bytes
10 XVST xr0, %[x0_value], 0
```

Listing 1: Minimal variant of LoongLeak. The target cache set to be leaked is the one of `addr_a`. `addr_b` is a dummy accessible address from which the first 8 bytes are loaded.

loads on the CPU core, independent of their security domain. LoongArch defines `FLD.S` as a 4-byte single-precision load into a 8-byte floating-point register (FPR). The LoongArch manual specifies that for this operation, the upper 4-byte are *uncertain* [20]. Moreover, the FPR is aliased with the vector register, extending the uncertain bytes to the entire register size of 32 bytes. An uncertain value can assume any bit pattern. Thus, software is expected to mask and not use these bytes. The hardware does not prevent software from reading these bytes, though. Crucially, the behaviour is architectural: the stale bytes are visible in the retired register file and survive interrupts, context switches, and full pipeline flushes. As we reverse-engineer in Section 3.3, LoongLeak leaks stale data from the L1 data cache. Moreover, an attacker can even specify the cache set by priming the microarchitecture with a preceding load. Figure 2 shows an overview of LoongLeak, illustrating the leakage through "uncertain" values.

Listing 1 shows the basic code for LoongLeak that leaks stale data from a given L1 data cache set on a Loongson 3A5000 CPU. Assuming `addr_b` is aligned to 8 bytes, the lower 8 bytes of `f0` are loaded from `addr_b`. This includes the architecturally defined lower 4 bytes of the register. The remaining 24 bytes of the aliased `xr0` vector register consistently contain bytes that were resident in the *same* L1 data cache set that `addr_a` maps to.

3.2 Discovery

In this section, we outline our differential fuzzing approach that discovers symptoms of LoongLeak. We first identify all defined instructions in `qemu-user-loongarch` and on a Loongson 3A5000 CPU by executing each of the 2^{32} possible instructions. We limit our fuzzer to instructions that are defined on both. The fuzzer randomly chooses one of these instructions and executes it with 100 random initial states. The initial state consists of 32 general purpose registers, 32 vector registers and a 16 kB (1 page) memory mapping. The random initialization works on an 8-byte granularity and chooses one of the following values:

```
2b22b745    fld.s    $fa5, $s3, -1875
sample 0:
v5: 0xffffffffffffffffffffffffffffffff
  ffffffffffffffffffffffffd3aadff vs.
0xffffffffffffff075c4e9c09f6f2dd000000
  0800000a250f149059e3fd3aadff
(...)
```

Listing 2: Different outputs for an `fld.s` instruction reported by the differential fuzzer. QEMU (first) and Loongson 3A5000 CPU (second).

```
1 FLD.S f0, %[addr], 0
2 XVST xr0, %[x0_value], 0
```

Listing 3: Simple PoC to get the "uncertain" bytes. `FLD.S` loads 4 bytes from `addr`. `XVST` stores the 32 bytes of the destination register to memory, thus writing 4 loaded bytes and 28 uncertain bytes.

- 30 %: A pointer into the memory mapping.
- 30 %: A uniformly random value.
- 20 % each: 0 and -1 (0xffffffffffffffff).

After execution, the values of each register and memory location (8-byte granularity) that changed during execution are reported. We run the fuzzer inside QEMU and on a Loongson 3A5000 for 10 000 instructions with the same random seed, i.e., causing the same instructions to be executed with the same initial states. A script then compares the outputs and visualizes differences for manual inspection as seen in Listing 2. Our fuzzing reveals a number of differences in vector and floating-point loads: The architecturally defined lower bytes of the vector register contain the same correct value while the upper bytes differ. To determine the source of the upper bytes on the Loongson 3A5000 CPU, we proceeded with our analysis outlined in the following section.

3.3 Leakage Analysis

In this section, we analyze the leakage properties of LoongLeak on the Loongson 3A5000 CPU. We derive experiments to attribute the leakage to the L1 data cache and determine how previous loads influenced the leakage. Finally, we combine our observations to leak complete cache sets.

Setup. All tests run on an otherwise idle system under Linux 4.19.0 from the official Loongnix Desktop 20.6 distribution (latest at the time of writing). We use a Loongson 3A5000 for all experiments unless otherwise specified. We do not change any hardware or operating system settings (e.g., prefetchers or frequency scaling).

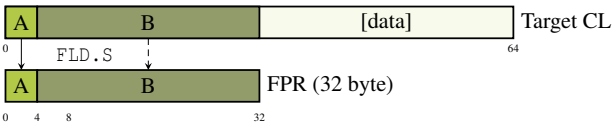

Figure 3: If `FLD.S` misses the L1 cache, an aligned `FLD.S` loads a 32-byte value from the destination.

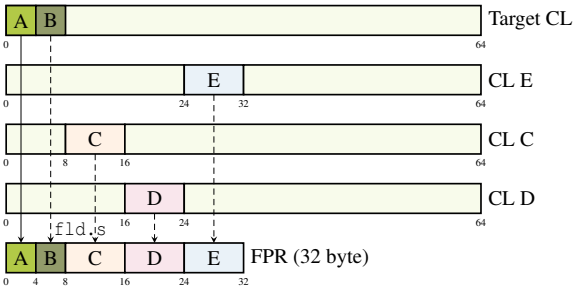

Figure 4: A 4-byte floating-point load (`FLD.S`) targeting the start of a cache line behaves like an architectural 8-byte floating-point load if the L1 data cache is hit. The remaining data in the 32-byte vector register originates from up to three different cache lines, 8 bytes each.

Basic Leakage. Listing 3 shows the simplest instruction sequence that returns values for the "uncertain" upper bytes. In this experiment, we always align our accesses to the start of a cache line. The lower 4-bytes of `f0` hold the expected word from `addr`. The upper 4-bytes of `f0` contain the bytes expected from a 8-byte load. The remaining bytes in the aliased `xr0` vector register contain at this point unpredictable bytes, including kernel addresses and values used previously in the application. When a large memory buffer with a cyclic pattern is accessed before executing the code from Listing 3 to bring controlled data into the cache, the leakage disappears and the load behaves like a 32-byte load as depicted in Figure 3. This is the case because the accesses evict the load target of the `FLD.S` instruction from the L1 data cache, and leakage is only present if `FLD.S` hits the L1 data cache. By adding another load to the attacker-controlled `FLD.S` target after accessing the cyclic pattern buffer, the leakage reappears. Using the cyclic values, we determine that the leakage consistently (more than 90 %) comes in bunches of 8 bytes with cache-line offsets following the load. Figure 4 depicts this: While the leakage may come from up to 3 different cache lines, it is always 8 consecutive bytes with cache-line offsets of 8, 16, and 24. In fewer than 10 % of cases, the leaked bytes are all zero.

Other Load Instructions. We repeat the experiment with load instructions other than `FLD.S` listed in Table 2. We find that only floating-point and vector load instructions exhibit leakage, while integer loads zero or sign-extend the upper register bytes beyond the architectural load size. For `LD.D` and

Table 2: Load instructions available on LoongArch CPUs, their type, architectural bytewidth, and whether they leak data. Only floating-point and vector loads leak.

Instruction	Type	Bytewidth	Leakage
`FLD.S`	Float	4	✓
`FLD.D`	Float	8	✓
`VLD`	Vector	16	✓
`XVLD`	Vector	32	N/A
`LD.B(U)`	Integer	1	✗
`LD.H(U)`	Integer	2	✗
`LD.W(U)`	Integer	4	✗
`LD.D`	Integer	8	N/A

`XVLD`, leakage is not applicable as the architectural load size equals the target register size. This is in line with the LoongArch manual [20], which describes that integer loads zero- or sign-extend the value, while `FLD.S` leads to "uncertain" upper bytes of the register. From all leaking loads, `FLD.S` leaves most bytes of the target register in an "uncertain" state. Thus, we focus on `FLD.S` for the remaining experiments.

Leakage Source. We move the victim, including the cyclic pattern buffer, across different isolation domains, including to another user application on the same core, another user application on a different core, the kernel, a different VM pinned to the same core, and the hypervisor. To rule out any effects of old kernel versions, we verify LoongLeak on the 3A5000 with Arch Linux and kernel version 6.15.0 with all default mitigations enabled. For our VM setup, we use Linux 6.13.0 with KVM running QEMU 10.0.50 as hypervisor and Linux 6.15.0 as guest. As shown in Table 3, there is no leakage across physical cores. On the Loongson 3A6000, we also observe leakage across SMT threads, which are unavailable on the Loongson 3A5000. Hence, the leakage likely originates from a per-core microarchitectural component.

Of those components, the leaked values are only feasibly present in the integer register file, the vector register file, or in some microarchitectural component belonging to the memory subsystem, such as the L1 data cache. When filling the entire integer register file and vector register file with known values before executing the leaking load, we still observe leakage. Thus, the values do not come from either of the register files and likely originate from the memory hierarchy.

Targeted Leakage. To determine whether the leakage can be influenced, we run the leakage primitive with different preceding instruction sequences. Adding a preceding 32-byte load (`XVLD` instruction) to the same address as the `FLD.S` instruction removes any leakage. However, if the preceding

Table 3: LoongLeak leakage across different isolation boundaries. We pin different VMs to the same core.

Isolation Boundary	Leakage
Same core	✓
Kernel/User	✓
Different core	✗
Different VM	✓
Hypervisor/VM	✓

`XVLD` instruction targets a different L1 data cache set, we consistently observe 24 consecutive bytes of leakage originating from that cache set as depicted in Figure 2. Thus, an attacker can specify the L1 data cache set that is being leaked and obtain bytes 8-31 from a cache line within this cache set.

Unaligned Loads. We analyze how the leakage changes with different load offsets, as all previous experiments use loads targeting the start of a cache line (i.e., address mod $64 \equiv 0$). We use the code from Listing 1 and vary the cache-line offset of `addr_a` and `addr_b`. If `addr_b` is offset by one byte, only 7 bytes are loaded from `addr_b`, in contrast to the 8 bytes on an aligned load. The amount of consecutive bytes loaded from `addr_b` depending on the cache-line offset is depicted in Figure 5. The pattern can be explained by dividing the load into 8-byte aligned slots while `FLD.S` architecturally loads 4 bytes. Each slot that contains at least one byte of architectural data is initialized to load 8 consecutive aligned bytes. Thus, cache-line aligned loads load 8 bytes of architectural data and loads with addresses offset by 4 bytes only load 4 bytes of architectural data because only the second half of the slot is used to fill the register. For loads that cross a slot boundary, up to 11 consecutive bytes are loaded from `addr_b`: 8 bytes originate from the second slot and up to 3 bytes come from the first slot. No more than 3 bytes can be loaded from the first slot since the 4-byte load must have at least one architectural byte in the second slot to initialize it. Figure 6 visualizes a 4-byte `FLD.S` load to cache-line offset 38: The architecturally loaded data overlaps with two slots. Thus, both slots are initialized with data from the target cache line and loaded to the register. Since only the last two bytes of slot A are loaded, a total of 10 bytes (2 from slot A + all 8 from slot B) come from the target cache line.

Keeping the offset of `addr_b` 8-byte aligned leads to leaked cache-line offsets "following" the load address: For `addr_b` offset 16, the leaked data begins at cache-line offset 24. Since the first 8 bytes of the register are filled with architectural data, offset $24 = 16 + 8$ perfectly follows the load offset. When the offset exceeds the cache-line boundary (e.g., `addr_b` offset 56), the leakage cache-line offset wraps around. Thus, `addr_b` offset 56 leaks bytes 0-23 from a cache line. If the cache-line offset of `addr_a` differs from that of `addr_b`, parts of the

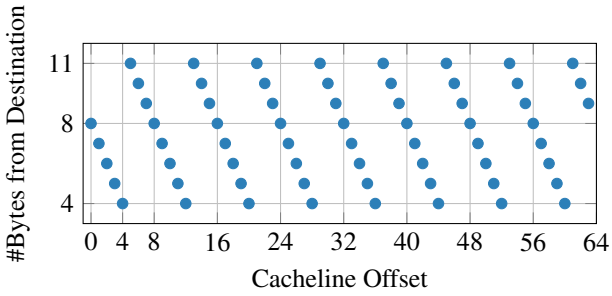

Figure 5: Number of bytes loaded from the destination of `FLD.S` depending on the cache-line offset targeted by `FLD.S`.

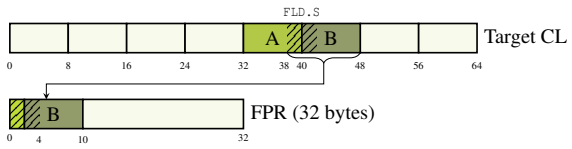

Figure 6: An unaligned 4-byte floating-point load (`FLD.S`) to cache-line offset 38 leads to 10 bytes loaded from the target cache lines. 8-byte aligned blocks explain this: If a block contains a single byte that is architecturally accessed, the entire 8-byte block is loaded from that cache line (green).

leakage are no longer from a cache set determined by `addr_a`. For instance, if `addr_a` is cache line aligned and `addr_b` is offset by 8, only 16 bytes of the leakage stem from the target cache set, and the upper-most 8 bytes originate elsewhere. If `addr_a` covers multiple cache lines and the cache-line offset matches with `addr_b`, the leakage only originates in the cache set from the first cache line. By adjusting the cache-line offset of the priming and leaking load, an attacker can leak 24 consecutive bytes from any 8-byte aligned cache-line offset.

Fine-Grained Leakage Control. As observed in the previous experiments, leakage is comprised of 8-byte aligned slots. Instead of using a 32-byte `XVLD` load instruction as a priming load, smaller loads can be used. For instance, an 1-byte `LD.B` load instruction only ever affects a single 8-byte aligned slot. In an experiment, we use multiple 1-byte loads to prime each slot with a different cache set. We also extend the leaking to two `FLD.S` instructions, one targeting cache-line offset 0 and one targeting cache-line offset 32, to leak a total of 48 bytes in a single run. With this experiment, we leak 8 bytes each from 6 different cache sets in a single run, as expected. Figure 7 shows this: Two instances of 8 bytes of the leaked data come from the `FLD.S` target cache line starting at offsets 0 and 32. The remaining 48 bytes originate from different offsets of 6 different cache lines from different cache sets that the priming load instructions (`LD.B`) determine.

If we instead use two 32-byte `XVLD` loads to prime all 8-byte aligned slots with a single cache set, the leakage of the two

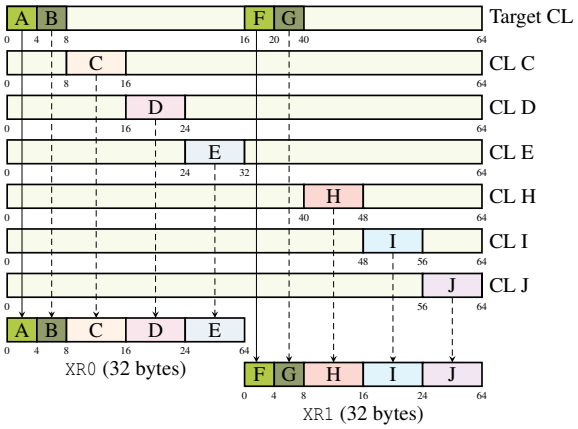

Figure 7: Using six priming loads to different cache sets and two leaking loads to `xr0` and `xr1`, 8 bytes are leaked from six different cache lines, each residing in one of the target sets.

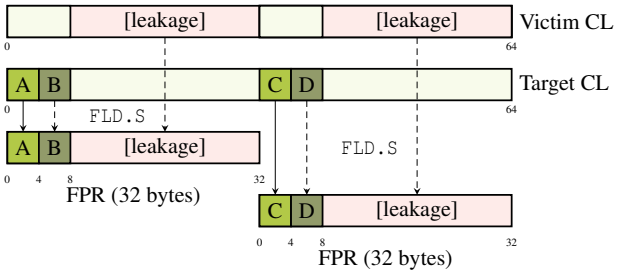

Figure 8: LoongLeak leaking 48 bytes from one cache line using two `FLD.S` instructions to different floating point registers. The first `FLD.S` instruction loads from cache-line offset 0, and the second `FLD.S` instruction loads from cache-line offset 32. Bytes 8-31 and 40-63 of a victim cache line leak.

`FLD.S` instructions now yields 48 bytes from the same cache line as depicted in Figure 8. The missing 16 bytes consist of two 8-byte aligned 8-byte values, which are 32 bytes apart (so two `FLD.S` can cover the entire cache line). Thus, in a single execution, an attacker can leak 48 bytes from one cache line with some limited control over which 16 bytes are missing.

Leaking a Whole Cache Line. By combining two 48-byte leaks that use the same addresses for priming and that target the same cache line with the leaking `FLD.S` instructions only at different offsets, we can leak a whole cache line. This is the case because the microarchitectural buffer is primed the same way twice, resulting in the same cache set leaking. By using different offsets but the same cache line with the leaking `FLD.S` instructions, the missing 8-byte regions of the cache line can be chosen not to overlap. Hence, each 8-byte region of the cache line is leaked at least once, allowing recovery of the entire cache line.

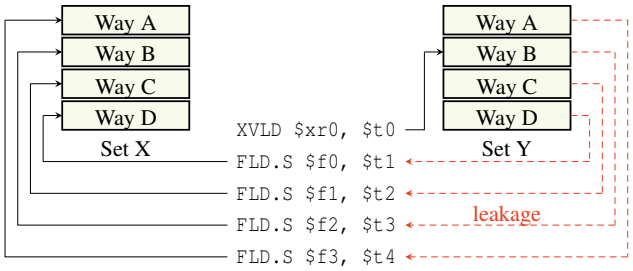

Figure 9: The cache way leaked by `FLD.S` from cache set 'Y' is the same as the cache way that is architecturally accessed in cache set 'X'.

Determining The L1 Data Cache Way. Repeatedly leaking a cache line from the same cache set using the technique described above leads to the same cache way leaking repeatedly. However, when targeting different addresses with the `FLD.S` leaking load, different cache ways leak. We hypothesize that the cache way that is hit by the `FLD.S` instruction determines the cache way of the leakage. To validate our hypothesis, we leak a cache line from the same cache set multiple times using four different `FLD.S` targets, each mapping to the same cache set, i.e., each residing in a different cache way. For each `FLD.S` target, we consistently leak the same cache way while the leaked value differs between all targets (Figure 9). This strengthens our hypothesis that the cache way hit by `FLD.S` determines the leaked cache way.

Leaking a Complete Cache Set. Using this technique of four different cache lines residing in different cache ways targeted by the `FLD.S` instruction, the leakage can be extended to a complete cache set. Since the priming load must also hit the target cache set, one of four cache ways is evicted and replaced with the cache line hit by the priming load. Thus, one of the four leaked cache lines always contains attacker-controlled data, limiting LoongLeak to leak the three most recently accessed cache lines of each cacheset.

Loongson 3A6000. On the Loongson 3A6000, the leakage characteristics differ. While leakage exists on this CPU, it is limited to the second half of a cache line. An attacker can still specify the L1 data cache set of the leakage with a preceding priming load. In contrast to the Loongson 3A5000, the 3A6000 supports SMT. When the victim runs on an SMT sibling thread, we still observe leakage. However, when an SMT sibling thread is active, the cache set cannot be controlled. Still, in most cases, an attacker obtains 24 consecutive bytes from a single cache line with known cache-line offsets.

Speed and Accuracy. To evaluate the speed and accuracy of LoongLeak, we set up a victim buffer spanning one page (16KB) of random data. The victim buffer is first accessed so that it entirely resides in the L1 data cache. Then, for each cache set, all ways are leaked, and the correct way is determined based on the first 8 bytes of leaked data. To obtain the whole cache line, the cache line is leaked twice, once missing bytes 0-7 and 32-39, and once missing bytes 8-15 and 40-47. To leak all ways, leakage of a cache line is repeated four times with `FLD.S` loads targeting different cache ways. The leaked cache way with the first 8 bytes matching the victim data is chosen. If no such way exists, the last leaked way is selected. This way is compared against the victim array, and the number of correctly leaked bytes is calculated. The required time and the number of correctly leaked bytes is recorded. This is repeated a total of 1 000 000 times. LoongLeak leaks at an average rate of 315.8 MB/s, correctly recovering 90.3 % of the victim data. When the comparison of the leaked data with the victim data is omitted, a raw leakage speed of 483.3 MB/s is achieved. With this, LoongLeak is amongst the fastest data-leaking attacks and achieves high correctness even with a single shot. Thus, the bottleneck of exploitation is not LoongLeak itself, but the speed at which data can be processed and victim data can be brought into the L1 data cache. Moreover, in practice, data can only leak across privilege boundaries when a kernel transition occurs, as there needs to be stale victim data in the L1 data cache while the attacker executes. To leak from a different user process, a full context switch must occur. Hence, the operating system's scheduling algorithm additionally limits leakage rates in this case. To also gauge LoongLeak's leakage rates in such scenarios, we use it to construct a covert channel across two user processes or VMs. Our implementation transmits arbitrary files and uses checksums to ensure integrity. Furthermore, it systematically yields to the operating system at specified points, maximizing the amount of context switches over time. We evaluate this covert channel by transmitting 16 MB of random data, using a Loongson 3A5000 with Linux v6.15.0 and KVM. This way, we achieve average transmission rates of 6.2 MB/s across processes on the host and 4.5 MB/s across VMs, always with perfect integrity ($n = 32$). This demonstrates that, even in realistic scenarios, LoongLeak can leak data at high speeds.

4 Attacker Model & Attack Scenarios

The attacker executes unprivileged native code on a target system but has *no* control over kernel, hypervisor, or co-tenants. We assume a trusted operating system and no software vulnerabilities. The attacker and victim share at least one physical core, either via process scheduling or simultaneous multithreading (SMT). No co-location within the same address space or privilege level is required. This weak threat model is in-line with most side-channel attacks and several CPU vulnerabilities [18, 22, 28, 29, 32, 36].

On the hardware level, we assume a Loongson CPU with the LoongArch ISA. This includes, e.g., the 3A5000 and 3A6000. While these CPUs are rare in the US and Europe,

they are relevant in China, where they are used in the majority of government computers and servers, as well as in personal computers [35] and supercomputers [27]. LoongLeak does not require SMT. However, as LoongLeak works across logical CPUs, SMT increases the attack surface. No side channel, e.g., timing or contention, is required for LoongLeak. The attacker merely reads architectural unprivileged registers.

Attack Scenarios. We consider the following scenarios:

User-to-User Attack. A user application might target a different user application and leak its secrets. As many secrets are stored within applications, e.g., browsers or password managers, this is a dangerous scenario for end users.

User-to-Kernel Attack. An unprivileged application can leak data accessed by the kernel, e.g., during a syscall. This may include disk encryption keys or data of other applications.

VM-to-VM Attack. An attacker-controlled virtual machine can leak data from other co-located virtual machines. This is especially threatening in a cloud scenario where an attacker could be co-located with VMs from other customers. This allows, e.g., leakage of private certificates or SSH keys.

VM-to-Hypervisor Attack. An attacker-controlled virtual machine can also leak data from a hypervisor. If a virtual machine is used to isolate untrusted, possibly malicious, code like malware, LoongLeak can cross the virtual machine boundary and leak host data from inside a VM.

5 Case Studies

In this section, we demonstrate the security impact of LoongLeak. In Section 5.1, we show how an unprivileged attacker can leak disk encryption keys from the kernel on the Loongson 3A5000. In Section 5.2, we use LoongLeak to leak a partial root password hash on the Loongson 3A6000. Section 5.3 demonstrates how LoongLeak can leak stack canaries and ASLR offsets, aiding traditional software exploitation.

5.1 Leaking Disk Encryption Keys

To demonstrate that LoongLeak can leak data from privileged contexts, we show that it can extract the 512-bit master key used by a LUKS-encrypted volume.

Target. LUKS is the de-facto standard for full-disk encryption on Linux [8]. It encrypts and decrypts data using a master key stored in kernel memory after unlocking the volume. This key is itself encrypted on disk and decrypted during volume activation using a user-supplied credential, e.g., passphrase or TPM key. AES-XTS is the default used for disk encryption, requiring two 256-bit AES keys: one for encryption and one for tweak generation.

Setup. We configure a LUKS block device with AES-256 in XTS mode, yielding two 256-bit keys (512 bits total). To allow an unprivileged attacker to repeatedly trigger encryption, we create a world-writable file on the encrypted volume. The attack is executed on a Loongson 3A5000 using LoongLeak to leak the master key from the L1 data cache. While unprivileged code execution and physical access is not the typical threat model for disk encryption, the case study demonstrates reliable leakage of kernel secrets. We leak the entire L1 data cache 10 times by iterating over all cache sets and ways, as the virtual address of the in-kernel master key is unknown. This leads to a total of 640 kB of leaked data. To increase the probability that the master key resides in the L1 data cache at the time of leakage, a second unprivileged attacker process continuously writes to a fixed offset in the world-writable file using the `O_SYNC` flag to force synchronous writes and trigger block device encryptions. While this setup simplifies the attack for demonstration purposes, a real-world attacker could instead wait for legitimate system activity to cause the master key to be loaded into the cache.

Results. Across 10 independent trials with different master keys, both 256-bit AES subkeys are contained at least once within the leaked data. Since both keys are cache-line aligned, there are a maximum of 10240 candidates for each key.

As expected, the write-heavy second attacker process is essential for the success of the attack, since LoongLeak leaks from the L1 data cache. Notably, we occasionally recover the master key even without active writes, suggesting that kernel-space prefetching or implicit I/O activity can load the key into the L1 data cache opportunistically. These results confirm that LoongLeak enables the extraction of high-value secrets—such as full-disk encryption keys—from kernel memory.

5.2 Leaking the Root Password Hash

On the Loongson 3A6000, LoongLeak can leak data from SMT sibling threads. In this case study, we use this to leak password hashes from the `/etc/shadow` file.

Target. The `/etc/shadow` file is used to store password hashes on Linux systems. As such, this file is a critical part of authentication mechanisms, and its contents must be protected from unauthorized access. Hence, the file is only accessible to the privileged root user. Interaction for unprivileged users is only possible through several root-owned setuid binaries. These binaries always execute with the effective user ID of root and can thus interact with the shadow file.

Setup. In line with other work [40, 41], we run the `passwd -S $USER` command in a loop pinned to a logical CPU core. This command executes the `passwd` root-owned setuid binary, which reads the shadow file and prints metainformation about

the user's password. Pinned to the SMT sibling core, we run LoongLeak with an `FLD.S` load targeting random offsets and filter for patterns matching the start of SHA-512 password hashes, i.e., starting with `$6$`. Whenever such a string is found, it is printed to stdout. The attack is executed on a Loongson 3A6000 CPU running Loongnix Desktop Version 20.6.

Results. We execute the attack 100 times for 10 seconds each. For each run, we filter the output for strings that match the character set of valid password hashes (`[a-zA-Z0-9./\$]`). We take the most common longest string that occurs at least 3 times and compare it to the root password hash. In all 100 runs, the selected candidate is a correct substring of the root password hash. The length of the leaked partial password hash is between 20 and 23 characters, with an average of 22.76. By not only matching for the start of the password hash but also previously leaked substrings, the leakage can be extended to 27 characters. The leaked hash contains the full salt and 7 Base64 characters (i.e., 42 bits) of the password hash. These results show that even though leakage cannot be targeted when an SMT thread is active, LoongLeak is capable of recovering sensitive data such as password hashes from an SMT sibling thread.

5.3 Breaking Software Mitigations

In this case study, we demonstrate that an attacker can leak both the ASLR base address and the stack canary from a colocated application. To mitigate memory corruption attacks, such as stack-based buffer overflows, modern compilers harden the generated code with stack canaries by default—random values placed between stack variables and saved return addresses [3]. Additionally, position-independent executables are produced that can benefit from random addresses through Address Space Layout Randomization (ASLR).

Target. As the victim application, we run ffmpeg v7.1.1 [10], compiled as an ELF executable with stack canaries. To verify the correctness of the leaked stack canary, we add a small function that prints the stack canary when ffmpeg starts up. We modify no other code.

Setup. We pin ffmpeg to a single CPU core and configure it to perpetually encode a self-generated video feed with its built-in GIF encoder. On the same core, we then use LoongLeak on all cache sets, saving values that match properties we expect our target values to have. For stack addresses, this is every naturally aligned 64-bit value where bits 48 to 63 are '0', and bits 24 to 47 are the binary's well-known address prefix for stack addresses (e.g., `0x7ffffb`). To locate the stack's top, we round the highest address matching these criteria to the next highest page boundary. For canaries, we save naturally aligned 64-bit values where the least significant byte is '0'. Furthermore, since the canary value's 56 upper bits are random, we can expect its Hamming weight to be close to $56/2 = 28$. Hence, we only save candidates where the Hamming weight is between 18 and 38. A uniformly random stack canary has a probability of $\sum_{k=18}^{38} \binom{56}{k} * 2^{-56} \approx 99.5\,\%$ to be in this range, whereas most other values we leak fall outside it. Finally, we rank candidates for the stack canary based on their Shannon entropy, and use the observed frequency to break ties, with the highest-ranked number being the final output. Our implementation runs in increments of 0.1 s for stack addresses and 0.25 s for canaries, and repeats until at least one suitable output value is leaked.

Results. We execute this attack 100 times on a Loongson 3A5000, targeting separate victim processes in each run. In 96 % of cases, we correctly recover the stack's top address, taking 0.11 s on average. We correctly recover the canary in 99 % of cases, taking 0.31 s on average. This demonstrates that LoongLeak is highly effective for circumventing such randomization-based defenses. Note that while we only target user-mode applications, this technique can also be applied to leak addresses from the kernel. We verify using `/proc/kallsyms` that LoongLeak can also leak kernel pointers, breaking KASLR on kernels that support it.

6 Countermeasures

In this section, we explore different strategies for mitigating the exploitation of LoongLeak. As Loongson does not support microcode, we focus on mitigations that can be purely implemented in software, either on the kernel level, hypervisor level, or software level, depending on the threat model. Finally, we propose a hardware fix that can be implemented in upcoming CPUs.

6.1 Kernel-based Mitigations

Kernel-based mitigations are required for non-virtualized environments, such as personal computers. In line with our threat model, they assume an unprivileged attacker.

Disable Floating Point and Vector Extension. The most straightforward solution is to fully disable the FPU. Consequently, floating-point registers and instructions are unavailable, stopping execution of leaking instructions. While the Linux kernel does not provide a boot parameter to disable the FPU, the kernel provides such functions to disable it at runtime. The LoongArch privileged register `EUEN` already allows the operating system to mask the FPU. We patch the Linux kernel to disable the FPU at startup, fully disabling the FPU for kernel and user space. With this patch, we do not see any leakage with LoongLeak, as the `FLD.S`, `FLD.D`, and `VLD` instructions simply fault. However, this straightforward

patch, while effective, also has a huge performance impact. We see a mean slowdown of −3.8 % to 14.2 % on SPEC CPU 2017 integer workloads and 10 × to 21 × on floating-point workloads (Figure 13 and Figure 14 in Appendix A).

Future work could investigate an optimization of this approach. The operating system could still report the FPU as being available, leading to traps whenever an FPU instruction is executed. The kernel could then temporarily re-enable the FPU, re-execute the instruction, and deactivate it again. Before returning the result to user space, the uncertain bytes can be zeroed to eliminate the leakage.

L1 Data Cache Flush. As the leakage comes from the L1 data cache, the kernel can flush the L1 data cache on transition to userspace. If SMT is not available (e.g., on the 3A5000), this ensures that no secrets are in the cache when the attacker is scheduled. Such a mitigation is also used on Intel CPUs to mitigate MDS [16]. Unfortunately, Loongson does not expose an instruction to flush the L1 data cache. Consequently, the only option is to evict the L1 data cache on each transition to userspace. As the L1 data cache uses LRU as replacement policy, a simple `memset` with the size of the cache is sufficient to reliably evict the entire cache. Our measurements show that a complete L1 eviction takes on average 820 ns on LA464/LA664. Even though this results in slower transitions to userspace, we confirm that the leakage is fully eliminated. In the SPEC CPU 2017 benchmarks, we could not measure a meaningful slowdown (Figure 15 and Figure 16 in Appendix B).

Core Scheduling with SMT. LoongLeak works both within a core and across SMT siblings. Loongson does not support disabling SMT in hardware. While Linux can offline one logical core per physical core to effectively disable SMT, this also results in a performance degradation, as many microarchitectural resources are statically split between the logical cores—disabling a core leads to half of the resources being unavailable. A cheaper alternative is core scheduling: the OS ensures that at any point in time, both logical CPUs of a core run code from the same privilege level. This idea has proven effective against fill-buffer attacks on x86 [17] and provides the same isolation benefit while retaining SMT for homogeneous workloads such as web-server threads.

6.2 Hypervisor-based Mitigations

In the virtualization setting, the hypervisor can also fully disable the FPU or flush the L1 data cache on machine entry. Moreover, a hypervisor has additional mechanisms for mitigating LoongLeak.

Disable Floating Point and Vector Extension per VM. Additionally, the hypervisor can turn off the vector extension solely for a virtual machine. We verify that using `qemu-system-loongarch64`. Disabling the FPU for the virtual machine by using a virtualized CPU instead of the host CPU fully prevents LoongLeak. Trying to run LoongLeak inside the virtual machine leads to an illegal instruction exception as soon as the `FLD.S` instruction is executed.

Scheduling. As LoongLeak leaks data from a single core, a hypervisor can ensure that mutually untrusted VMs do not share a CPU core. This is especially relevant for SMT, where both logical cores should be assigned to the same VM to avoid cross-VM data leakage. Additionally, to prevent leakage from the hypervisor, the hypervisor still has to flush the L1 data cache before a VM enter.

6.3 Software-based Mitigations

Since leakage originates from the memory hierarchy close to the L1 data cache, data stored in other places may not be leaked. With 32 32-byte vector registers, the Loongson 3A5000 and 3A6000 provide up to 1024 bytes of memory that are unaffected by LoongLeak. Similar to the approach proposed by Hornetz et al. [14], these registers can be used to protect sensitive values, such as cryptographic keys.

6.4 Hardware Fix

Ultimately, this issue should be fixed in hardware for upcoming CPUs. The most straightforward approach would be to always clear the "uncertain" upper register bytes on loads smaller than the register size. For integer loads, this is already the architecturally-defined behavior. While we do not have access to the source code of the affected CPUs, we speculate that parts of the integer load pipeline could be replicated for floating-point loads to ensure this behavior.

7 Discussion

In this section, we discuss our microarchitectural root hypothesis of LoongLeak, the differences between architectural and transient leakage, LA64 and x86, other CPU vulnerabilities, and related work.

7.1 Microarchitectural Root Cause

In this section, we present our root-cause hypothesis for LoongLeak. Some publicly available documents [19, 20] including some about processors preceding the Loongson 3A5000 [15, 24] support our hypothesis and it explains our observations. Still, as no in-depth documentation about the relevant microarchitectural components is available, we can neither confirm or deny the hypothesis.

Banked Cache Design. As described in Section 3.3, the leakage stems from the L1 data cache and is separated into 8-byte aligned slots. We hypothesize that this is caused by a banked cache design with eight banks each handling an 8-byte aligned region. This is in line with documentation on the loongson 3A2000 CPU which divides each 64-byte cache line into "8 8-byte wide blocks" [24]. Each of these banks can perform independent accesses to their corresponding 8-byte "block" which allows parallel accesses to multiple cache lines.

Split Data and Tag Caches Hypothesis. As outlined in multiple publicly available diagrams for related processors [15, 19, 24], tag comparison and cache lookups are done in separate steps. We hypothesize that separate caches store data and tags such that the tag comparison to select a cache way can be done independently of reading any data. A possible implementation of this design would lookup and compare the tags for all ways of a target cache set. Based on the result, the implementation decides whether the cache is hit and in case of a hit which cache way is hit. Since this tag comparison is only possible once the physical address is available, the L1D cache can already select the correct cache set and output all ways for this set in parallel. Once tag comparison is complete and a cache hit is determined, the corresponding way can be selected from the four ways output by the data cache. This data is then forwarded to its destination, e.g., a 32-byte vector register located in the vector register file.

Usage of Partially Initialized Banks Hypothesis. For floating-point and vector loads, we hypothesize that the cache set is only correctly initialized for the cache banks that (partially) load architectural data while the cache way from the tag comparison is used for all banks. If this is the case, the data in the upper register bytes originates from partially initialized cache banks. More specifically, the cache set of the data is determined by a previous load using the cache bank while the cache way is the same cache way hit by the floating-point or vector load. This is in line with the leakage we analyze in Section 3.3. The leakage of LoongLeak under our root-cause hypothesis is modelled in Figure 11. Beforehand, two priming `XVLD` loads initialize all banks with the target cache set `X` (Figure 10). During tag comparison, the correct way `A` is used for the cache line accessed by `XVLD`. Then, two `FLD.S` instructions are executed. Each of these instructions only initializes one cache bank with the correct cache set `Y`. The stale cache set `X` from the previous `XVLD` load is used for the remaining banks. Then, the cache way `B` hit by the `FLD.S` instructions is used for the stale banks leading to leakage of possibly attacker-inaccesible data from an attacker-controlled cache set. As the way is determined by the way hit by `FLD.S`, an attacker also has partial control over the leaked cache way.

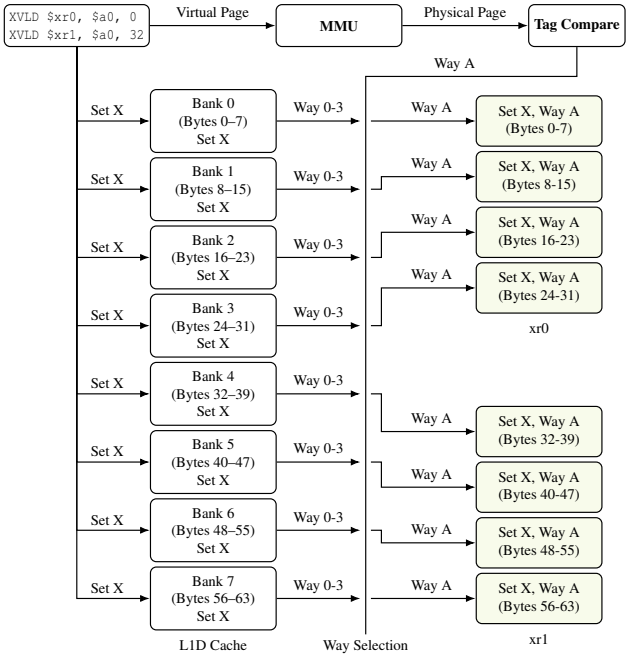

Figure 10: LoongLeak using two `XVLD` loads to prime all cache banks with cache set `X`. Green represents architecturally accessible data. The visualization is based on our root-cause analysis in Section 7.1.

7.2 Architectural vs. Transient Leakage

Architectural leakage manifests in components belonging to architectural state, such as registers or memory. These components can be directly accessed through instructions. Such leakage persists until other instructions that modify this architectural state are executed. Transient leakage appears in microarchitectural components. Such components are usually not directly accessible through instructions. Thus, an additional side channel is required to transform microarchitectural to architectural state. As the leakage only exists in microarchitectural state, it might be destroyed by non-architectural events such as pipeline flushes or code running on an SMT sibling thread. As the leakage from LoongLeak manifests directly in architectural vector registers, we consider LoongLeak to be an architectural vulnerability.

7.3 LoongLeak with Transient Loads

LoongLeak leaks data from the L1 data cache. While data that is actively accessed by a victim is loaded into the L1 data cache, data-at-rest cannot be targeted directly. Recent work used hardware prefetchers [13] or transient loads [12, 33] to bring data-at-rest into the cache. We confirm that transient loads of a half-Spectre gadget suffice to bring data into the L1 data cache, such that it can subsequently be leaked with LoongLeak.

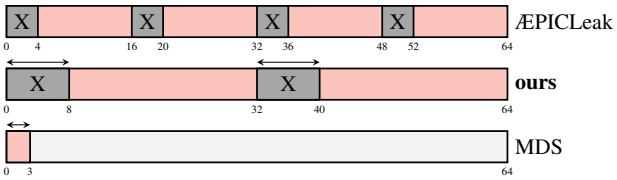

Figure 12: Leakage characteristics comparison of multiple CPU vulnerabilities. ÆpicLeak leaks a full cache line missing fixed 4-byte regions. MDS typically leaks three consecutive bytes. LoongLeak leaks a full cache line missing two attacker-controlled 8-byte regions.

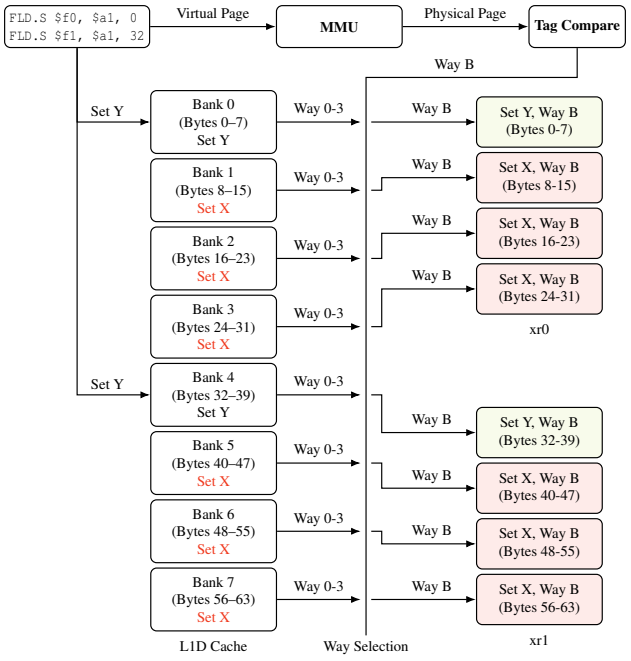

Figure 11: LoongLeak using two leaking loads after priming to leak a total of 48 bytes. Red represents leakage while green represents architecturally accessible data. The visualization is based on our root-cause analysis in Section 7.1.

7.4 LoongArch vs. x86

While x86 dominates the western market for desktop and server CPUs, LoongArch CPUs are on the rise in eastern markets, especially China. Currently, consumers are still mostly relying on x86 and Arm, but critical actors like governments have already largely switched to LoongArch CPUs with plans to increase the reliance on domestically manufactured CPUs [34]. Since these Loongson CPUs implementing the LoongArch ISA are employed in high-value targets like governments, their security is of utmost importance. LoongLeak shows that there is a blind spot in recent research, and LoongArch CPUs do not receive the attention they warrant.

7.5 Comparison to Other CPU Vulnerabilities

In this section, we compare LoongLeak to existing data-leaking CPU vulnerabilities on x86 and RISC-V.

Architectural CPU Vulnerabilities. Table 1 compares LoongLeak with recent *architectural* bugs, i.e., bugs that do not require a side channel to exploit. ÆPICLeak [4] affects a small set of Intel CPUs and requires a privileged attacker to leak from the superqueue, i.e., the buffer between L2 and L3 cache. In contrast, LoongLeak is limited to the same core, but can be mounted by an unprivileged attacker. Moreover, LoongLeak provides an attacker control over the cache set and cache-line offset, whereas ÆPICLeak has no control over the leaked data. Compared to GhostWrite [36], affecting a subset of RISC-V CPUs, LoongLeak provides a read primitive, whereas GhostWrite offers a write primitive with fine control (i.e., byte-granularity). These attacks can both be mounted by an unprivileged attacker. CacheWarp [43], a privileged attack on AMD SEV-SNP, reverts stores architecturally, while LoongLeak instead leaks the *contents* of the victim's cache line without altering system state. ZenBleed [29] leaks data from the vector register file on AMD CPUs, while LoongLeak targets data in the L1 data cache. Similar to ZenBleed, LoongLeak also works across SMT sibling threads on LoongArch processors that implement SMT.

Transient Execution Attacks. Meltdown-type transient-execution attacks, including Meltdown [22], RIDL [40], ZombieLoad [32], and Fallout [5], rely on microarchitectural vulnerabilities observable indirectly via side channels. LoongLeak differs in two respects: (i) the leakage is architectural, i.e., no speculation, gadget, or side-channel attack is required, and (ii) the disclosure is deterministic and noise-free.

Meltdown can leak arbitrary data by virtual address from the L1 data cache, as long as a privileged mapping to this data exist in the page table. In contrast, LoongLeak also requires data to reside in the L1 data cache but does not need a mapping, as LoongLeak can specify the cache set, way, and cache-line offset of the leaked data. Further, LoongLeak on the Loongson 3A6000 is limited to the second half of a cache line, and leakage across SMT sibling threads cannot be targeted. RIDL, ZombieLoad, and Fallout leak stale data from microarchitectural buffers, giving an attacker only control over the cache-line offset, without being able to further restrict the cache line. Spectre [18] attacks can divert transient control-flow of a victim to leak data from registers, the cache, or main memory. However, Spectre attacks rely on special code snippets called "gadgets" in the victim code, i.e., the kernel, that are regularly patched if discovered [6]. For all of these attacks, the leakage is only available transiently, i.e., the leaked data is not directly retired in architectural state. Thus, an attacker needs an additional step to encode the leakage into a microarchitectural state such as the cache and later recover

the encoded leakage using a side channel. Typically, attacks use the cache as the microarchitectural element and Flush+Reload [42] to recover it via timing. In contrast, LoongLeak directly obtains the leakage as part of architectural vector registers. Thus, no side channel is necessary.

Leakage Characteristics. Figure 12 compares the leakage characteristics of LoongLeak, ÆPICLeak, and MDS attacks (including ZombieLoad, RIDL, Fallout, and Meltdown) in a single-shot leakage scenario. We do not consider vulnerabilities that do not directly leak data, such as CacheWarp, GhostWrite, and Reptar, as well as ZenBleed and Downfall, which directly leak register contents. For ÆPICLeak, an attacker leaks an entire cache line, but the first 4 bytes of every 16-byte block are missing. For LoongLeak, an attacker leaks a full cache line, with two blocks of 8 bytes missing. However, while the amount of leaked bytes is the same for ÆPICLeak and LoongLeak, LoongLeak has the advantage that the missing bytes are not at a fixed position. An attacker can partially choose at which position the two 8-byte blocks are missing, making this leakage primitive more flexible. The leakage of MDS looks entirely different: there, an attacker can choose which approximately 3 bytes from a cache line leak. Thus, it has a very similar flexibility as LoongLeak, but the amount of bytes leaked in a single shot is much lower.

7.6 Related Work

Unintended data exposure has been a long-standing issue in high-end CPUs. For instance, *transient execution attacks* near-ubiquitously affect the microarchitectures of Intel, AMD, and ARM [6, 18]. This includes attacks like Spectre [18], Meltdown [22], Foreshadow [38], and so-called Microarchitectural Data Sampling (MDS) attacks [5, 26, 32, 40]. As these attacks target hardware facilities, fixing them in software is challenging, often forcing hardware manufacturers and OS developers to sacrifice performance or functionality. In contrast to LoongLeak, the effects of such vulnerabilities only manifest during transient execution [6], which means that attackers must use side-channels to retrieve secrets. While the floating-point unit has been abused to facilitate transient execution attacks on x86 [31], no direct data leakage was possible.

More similar to LoongLeak are other *architectural CPU attacks*, which affect the program state directly. For instance, ZenBleed [29] exploits an implementation bug on AMD processors, allowing for cross-SMT leakage of vector register values. The ÆPICLeak vulnerability [4] exposes the contents of shared microarchitectural buffers to MMIO, thus breaking the confidentiality of Intel SGX. In addition to leaking inaccessible data, some architectural attacks can manipulate a victim program's state. Zhang et al. present CacheWarp [43], which allows attackers to selectively revert the memory of AMD SEV-SNP-protected virtual machines to a stale state. The GhostWrite vulnerability on T-Head RISC-V CPUs [36], enables unprivileged attackers to write data to arbitrary physical memory addresses. EntrySign [9] allows for loading unsigned microcode in AMD CPUs, making it possible to change instruction behavior and override architectural restrictions. Finally, the Reptar bug [28] on Intel CPUs changes instruction behavior in ways that are largely unknown to date, allowing for unprivileged denial of service. In contrast to these previous attacks, LoongLeak is the only public vulnerability affecting LoongArch CPUs.

8 Conclusion

Architectural CPU vulnerabilities remain one of the most serious threats to computer security, as they transcend privilege boundaries and cannot be masked by conventional timing defenses. Our work broadened the scope of such analysis to the LoongArch ecosystem and discovered LoongLeak, a data-leak vulnerability present in commercial off-the-shelf Loongson CPUs. Our case studies confirmed the severity: we recover full-disk AES encryption keys, extract root password hashes from user space, break ASLR, and leak stack canaries–all from unprivileged code. As the leakage is architectural, it requires neither high-resolution timers nor traditional side-channel amplification, and it grants the attacker precise control over cache set and line offset. We presented short-term software countermeasures that vary in complexity and overhead. While effective, a full mitigation still requires hardware changes.

Ethical Considerations

Affected Stakeholders. The affected stakeholders are:

- The company Loongson, which manufactures CPUs affected by LoongLeak.
- Entities that use affected CPUs. As Loongson does not publish details about all CPU sales, the exact set of customers is not known.
- People who rely on services running on affected CPUs.
- CPU designers.

How Stakeholders are impacted. Loongson is affected by our research in multiple ways. First, if LoongLeak is not mitigated, the vulnerability makes their affected products less appealing to customers. Second, exploitation or public knowledge of LoongLeak could damage the brand's reputation.

Entities that use affected CPUs either directly or indirectly (i.e., through a service provider) are affected by our research since exploitation of LoongLeak could lead to data leakage. This could include sensitive data from individuals and companies.

CPU designers can benefit from our research being published, as it helps them avoid similar vulnerabilities and design more secure CPUs. This is especially true for ISA design, as our research shows that unspecified values should be used with care.

Mitigations taken for negative impacts to stakeholders. If we immediately published the results of our research, malicious actors could exploit the vulnerability before patches are available and widely deployed, negatively affecting all stakeholders. To minimize the negative impact on stakeholders, we conducted responsible disclosure with Loongson and agreed to a one-year embargo. We also suggested software mitigations to keep affected CPUs usable, which can be rolled out during the embargo period.

How we concluded to conduct and publish our research. The reason for conducting our research is that vulnerabilities we might find can likely be fixed by the manufacturer in future CPUs, leading to more secure CPU designs. More secure designs increase the effort required by malicious actors to find and exploit new CPU vulnerabilities. Further, many CPU vulnerabilities can be mitigated in software, allowing affected hardware to remain usable. We thus concluded that researching CPU vulnerabilities on Loongson CPUs is ethical.

We further believe that publishing our research will have two positive effects: First, inspire further research targeting Loongson CPUs, which will likely lead to more secure CPU designs in the future. Second, demonstrate potential pitfalls in CPU design that can prevent similar issues in future CPU designs, including those from other entities. As LoongLeak can be mitigated in software (by flushing the L1 data cache on kernel exit and disabling SMT if necessary), publishing our research once mitigations are deployed will not lead to widespread malicious exploitation. We thus reached the conclusion that publishing our research after a long embargo period minimizes the negative effects on stakeholders while at the same time maximizing positive impacts. Thus, we believe the decision to conduct and publish our research is ethical.

Open Science

We provide a basic demonstration of LoongLeak, code to evaluate the speed and correctness of LoongLeak, and code required to run our case studies. For licensing reasons, we do not include benchmarks relying on the SPEC CPU 2017 benchmark suite. Thus, our artifact includes:

- **fuzzer**: Our differential fuzzer to compare QEMU to hardware.
- **leak_primitive**: A minimal proof-of-concept showing how LoongLeak leaks data.
- **evaluate_primitive**: Code used to evaluate speed and correctness of LoongLeak.
- **sudo_hash_poc**: The code for our case-study to leak partial password hashes on the Loongson 3A6000.
- **disk_encryption_leak**: The code for our case-study to leak disk encryption keys on the Loongson 3A5000.
- **aslr_break**: The code for our case-study to leak stack canaries and ASLR addresses on the Loongson 3A5000.
- **covert_channel**: The code for our covert channel we used to evaluate cross-process and cross-vm leakage accuracy and speed.

The artifacts can be downloaded from `https://zenodo.org/records/17974800`. More information on the individual artifacts is detailed in the file `README.md`. A Linux system running a Loongson 3A5000 processor is required to run the experiments. Further, the `sudo_hash_poc` requires a Loongson 3A6000 processor.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *USENIX Security*, 2022.

[3] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the Stack Protector for Fun and Profit. In *ICT Systems Security and Privacy Protection*, 2018.

[4] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*, 2022.

[5] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*. ACM, 2019.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[7] crosstool ng. Crosstool-ng version 1.27.0, 2025.

[8] cryptsetup. Luks - linux unified key setup, version 2.8.1, 2025.

[9] Josh Eads, Tavis Ormandy, Matteo Rizzo, Kristoffer Janke, and Eduardo Vela Nava. Zen and the art of microcode hacking, 2025.

[10] FFmpeg. Ffmpeg 7.1.1, 2025.

[11] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[12] Mathé Hertogh, Dave Quakkelaar, Thijs Raymakers, Mahesh Hari Sarma, Marius Muench, Herbert Bos, and Erik van der Kouwe. Rain: Transiently leaking data from public clouds using old vulnerabilities. In *S&P*, 2026.

[13] Lorenz Hetterich, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, Nils Bernsdorf, Eduard Ebert, and Michael Schwarz. ShadowLoad: Injecting State into Hardware Prefetchers. In *ASPLOS*, 2025.

[14] Tristan Hornetz, Lukas Gerlach, and Michael Schwarz. Lixom: Protecting Encryption Keys with Execute-Only Memory. In *FC*, 2025.

[15] Wei-Wu Hu, Fu-Xin Zhang, and Zu-Song Li. Microarchitecture of the godson-2 processor. In *Journal of Computer Science and Technology*, 2005.

[16] Intel. Microarchitectural Data Sampling, November 2021.

[17] The kernel development community. Core scheduling, 2019.

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

[19] Loongson Technology Corporation Limited. Loongson 3a5000/3b5000 processor reference manual - multicore processor architecture, register descriptions and system software programming guide, version 1.03, 2021.

[20] Loongson Technology Corporation Limited. Loongarch reference manual - volume 1: Basic architecture, version 1.10, 2023.

[21] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2020.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[24] Ltd. Loongson Zhongke Technology Co. Godson 3a2000 / 3b2000 processor user manual - volume ii, gs464e processor core v1.03, 2016.

[25] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[26] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security*, 2023.

[27] Chen Na. A teraflop computer "with loongson inside", 2008.

[28] Tavis Ormandy. Reptar, 2023.

[29] Tavis Ormandy. Zenbleed, 2023.

[30] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*, 2021.

[31] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[32] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[33] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. In *FC*, 2021.

[34] Roshan Ashraf Shaikh. China blocks intel and amd cpus for government offices and servers, plans to switch to domestic-made alternatives, 2024.

[35] Simon Sharwood. China claims breakthroughs in classical and quantum computers, 2025.

[36] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. In *CCS*, 2025.

[37] Paul Turner. Retpoline: a software construct for preventing branch-target-injection, 2018.

[38] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[39] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[40] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *S&P*, 2019.

[41] VUSec. RIDL test suite and exploits (GitHub), 2020.

[42] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[43] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

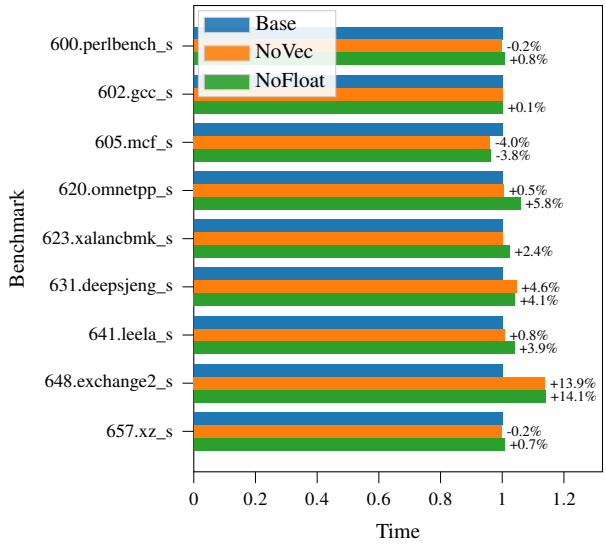

Figure 13: SPEC CPU 2017 integer benchmark results for three configurations: `NoVec` (hardware floating-point instructions but no LSX or LASX vector instructions), `NoFloat` (software floating-point emulation), and `Base` (hardware floating-point instructions and LSX and LASX vector instructions). `Base` is normalized to 1.

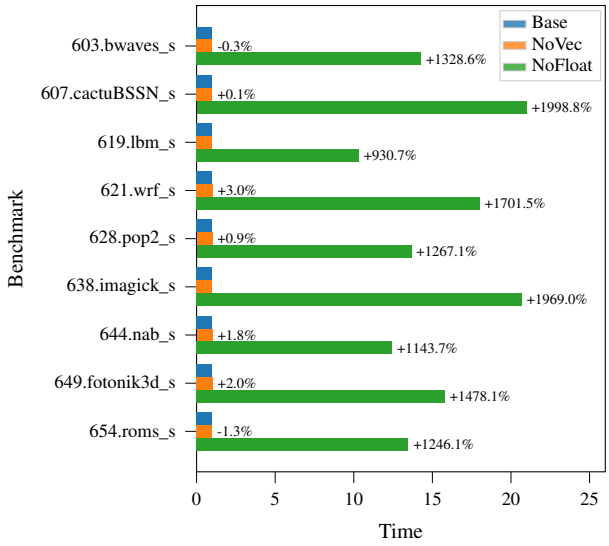

Figure 14: SPEC CPU 2017 floating-point benchmark runtime for three configurations: `NoVec` (hardware floating-point instructions but no LSX or LASX vector instructions), `NoFloat` (software floating-point emulation), and `Base` (hardware floating-point instructions and LSX and LASX vector instructions). `Base` is normalized to 1.

A SPEC CPU 2017 Overhead of Disabling Floating-Point and Vector Instructions

To determine the overhead of mitigations trapping floating-point and/or vector instructions, we execute the SPEC CPU 2017 benchmark. Since we are only interested in the overhead of emulating floating-point and/or vector instructions and not how performance scales with multiple threads, we execute the fpspeed and intspeed benchmark suites limited to a single thread.

Environment. As SPEC CPU 2017 does not provide support for LoongArch CPUs, we compile the SPEC tools ourselves. Further, the standard C library that ships on Desktop Loongnix version 20.6 which we use for all experiments if possible, does not support the `lp64s` Application Binary Interface (ABI). This ABI is used when compiling applications with software floating-point support and is not compatible with the default `lp64d` ABI. To ensure a fair comparison, we compile all required compilers (gcc for C, g++ for C++, and gfortran for Fortran) and the standard libraries from source using `crosstool-NG` [7]. We include the gnu C library version *2.41* and build GCC version *14.2.0*. We build one version of the toolchain using the `lp64s` API such that the resulting libraries can be used with software floating point. For our reference, we build the toolchain using the default `lp64d` ABI.

Setup. We build and run the SPEC benchmarks in three different configurations: `Base` which allows for hardware floating-point and vector instructions to be used, `NoVec` which disables `LSX` and `LASX` vector extensions, and `NoFloat` which fully relies on software floating point instructions. The benchmarks are executed on an Loongson 3A5000 running Desktop Loongnix version 20.6. However, the standard library and loader from the self-compiled environment are used for benchmark execution.

Results. In each configuration, One benchmark of the `intspeed` suite (`625.x264_s`) and one benchmark of the `floatspeed` suite (`627.cam4_s`) fail to compile. Additionally, one benchmark of the `intspeed` (`631.deepsjeng_s`) and two benchmarks of the `floatspeed` (`621.wrf_s` and `638.imagick_s`) fail to verify the results. As these benchmarks still execute successfully and fail to verify results with the same error regardless of configuration, we still include the runtime in our plots. The results are visualized in Figure 13 and Figure 14: The overhead of disabling only vector instructions is small with $-1.3\,\%$ to $3.0\,\%$ in floating-point and $-4\,\%$ to $13.9\,\%$ in integer benchmarks. Utilizing software floating-point instructions also has only a small impact on the execution time of Integer benchmarks of $-3.8\,\%$ to $14.2\,\%$. However, for floating-point benchmarks, `NoFloat` has an enormous overhead of $10\times$ to $21\times$. For such workloads, a mitigation relying on software floating-point emulation comes with a significant performance penalty.

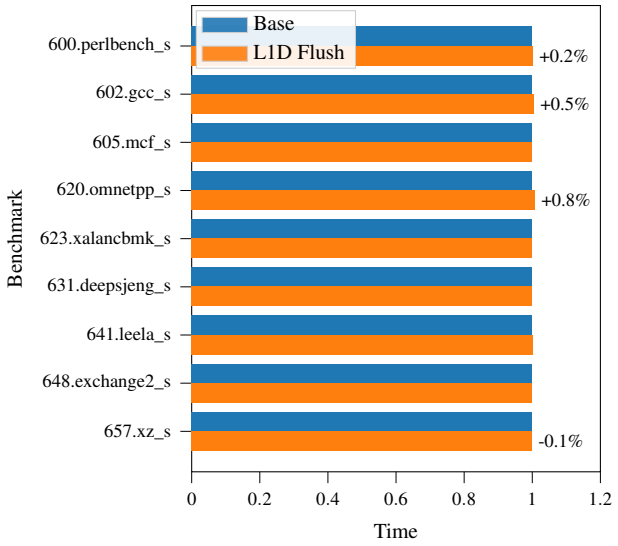

Figure 15: SPEC CPU 2017 integer benchmark results for two configurations: `L1DFlush` (evicting the L1 data cache on kernel exit) and `Base` (baseline with no mitigation). `Base` is normalized to 1.

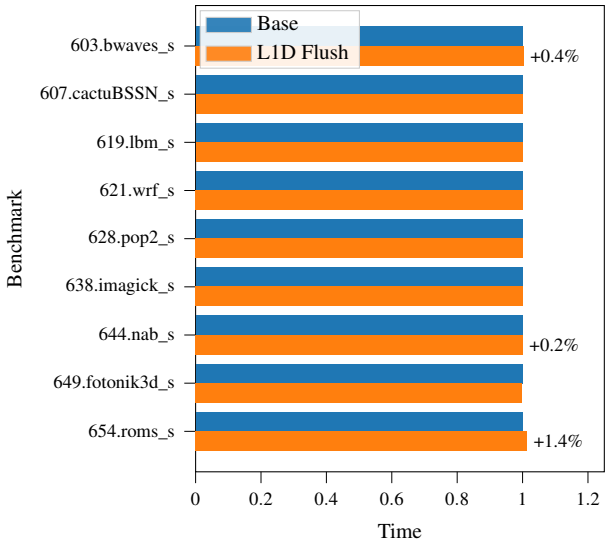

Figure 16: SPEC CPU 2017 floating-point benchmark runtime for two configurations: `L1DFlush` (evicting the L1 data cache on kernel exit) and `Base` (baseline with no mitigation). `Base` is normalized to 1.

B Spec CPU 2017 Overhead of L1D Flushing

To determine the overhead of mitigations flushing the L1 data cache on kernel to userspace transitions, we execute the SPEC CPU 2017 benchmark. We again execute the fpspeed and intspeed benchmark suites limited to a single thread.

Environment. Flushing the L1 data cache requires us to modify and re-compile the linux kernel. We build the Linux kernel version 6.15.0-rc7 from source, once with our mitigation evicting the L1 data cache on kernel exit and once in its original form as baseline. Analogous to Appendix A, we compile the Spec tools and the required compilers from source. We include the gnu C library version *2.42* and build GCC version *14.2.0*.

Setup. We build and run the SPEC benchmarks in two different configurations: `Base` which uses the unmodified linux kernel and `L1DFlush` which includes L1 data cache eviction on kernel exit. The benchmarks are executed on a Loongson 3A5000 CPU.

Results. The same benchmarks mentioned in Appendix A fail to compile for both configuration (`625.x264_s` and `627.cam4_s`) and to verify the results (`631.deepsjeng_s`, `621.wrf_s`, and `638.imagick_s`). Benchmarks that fail to verify their results are included in our plot as they fail with the same error in both configurations and are thus comparable. Overall, the performance penalty of `L1DFlush` on the Loongson 3A5000 is only observable in the `654.roms_s` floating-point benchmark at 1.4 % (cf. Figure 16). No significant overhead can be observed in the integer benchmarks (Figure 15). Hence, it is a promising stop-gap solution to mitigate LoongLeak on CPUs that do not support SMT.