Memory Tagging Extension for the XiangShan RISC-V Processor

Joshua Mathew
Columbia University
New York, USA
jm5915@columbia.edu

Tyler Chang
Columbia University
New York, USA
tc3407@columbia.edu

Valeria Espinoza
Columbia University
New York, USA
vne2102@columbia.edu

Abstract

Memory safety vulnerabilities, such as buffer overflows and use-after-free errors, remain a critical security challenge in low-level systems programming. While software-based sanitizers provide detection capabilities, they often incur prohibitive runtime overheads. Hardware-assisted Memory Tagging Extensions (MTE) offer a promising alternative by enforcing pointer-memory consistency directly at the architectural level. This paper documents the implementation and preliminary evaluation of the RISC-V ZIMTE extension, utilizing a Virtually Indexed Tag Table (VITT), within the XiangShan GEM5 simulator. We detail the development of a robust experimental workflow, ranging from bare-metal verification using the Abstract Machine framework to full-system Linux boot configuration. Our results validate the functional correctness of the tag generation and checking logic on bare-metal workloads, establishing a foundation for future performance characterization of VITT-based protection.

ACM Reference Format:
Joshua Mathew, Tyler Chang, and Valeria Espinoza. 2026. Memory Tagging Extension for the XiangShan RISC-V Processor. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Memory safety issues such as buffer overflows and use-after-free errors remain among the most critical sources of software vulnerabilities in systems written in low-level languages like C and C++ [1]. Traditional software-based debugging and sanitization tools (like Valgrind [5] and Pin [4]) have proven valuable for detecting such problems, but their high runtime overhead often limits their use to testing environments. Hardware-assisted memory tagging provides a promising alternative by enforcing pointer–memory consistency directly at the architectural level, reducing the performance cost while improving reliability.

The RISC-V Memory Tagging Extension (MTE) [2] introduces a set of mechanisms, such as the Zimt and Svatag/Smvatag extensions, that associate tags with memory chunks and pointers. These tags act as a lock-and-key system to detect invalid memory accesses at runtime. Within this framework, the Virtually Indexed Tag Table (VITT) plays a central role by defining a virtual-memory–based storage structure for tag lookup and update operations. Each memory chunk's tag is stored in the VITT, indexed by its virtual address, allowing the hardware to verify pointer–memory tag matches during load and store operations.

While memory tagging enhances security and debugging capabilities, it also introduces additional computational steps, such as tag lookups, tag stores and privilege-mode checks. These operations can impact execution time, memory access latency and cache performance [11]. Understanding the magnitude and nature of this performance overhead is essential for evaluating whether VITT and the broader MTE design can be efficiently deployed in real-world systems.

To conduct this analysis, we set up and configured the XiangShan GEM5 simulator as our experimental platform for implementing and evaluating the RISC-V Memory Tagging Extension (MTE) with Virtually Indexed Tag Table (VITT) support. This environment enables modeling of realistic out-of-order execution while maintaining fine-grained control over tagging configurations, providing the necessary foundation for accurately measuring VITT's microarchitectural overhead.

The remainder of this report is organized as follows. Section 2 provides background and related work. Section 3 describes the architectural implementation of the Zimt extension (ZIMTE) for XiangShan's GEM5 emulator. Section 4 presents and analyzes our results. Section 5 documents our attempts at solving challenging blockers and Section 6 discusses future work.

2 Background and Related Work

Numerous software-based approaches have been proposed to detect or mitigate memory safety issues. Tools such as AddressSanitizer (ASan) [7] and Valgrind [5] use compiler instrumentation or dynamic binary analysis to detect memory violations during testing. ASan, introduced in 2011, leverages shadow memory and redzones to detect both spatial (buffer overflows) and temporal (use-after-free) errors. However, this precision comes at a cost, because typical overheads range from 1.5×–3× in CPU usage, memory and binary size [8]. These costs restrict ASan's applicability mainly to development and fuzzing environments rather than production. Furthermore, ASan's software-only design has several limitations: it cannot instrument assembly code, analyze precompiled binaries, or detect invalid kernel memory accesses. It also lacks hardening properties, hence its metadata and redzones can be bypassed by skilled attackers [8].

To overcome these constraints, researchers have explored hardware-assisted memory tagging, also known as memory coloring, tainting,

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

or lock-and-key protection. The key idea is to associate each aligned memory block (granule) of size TG bytes with a small TS-bit tag. The same tag is stored in the unused high bits of every pointer referencing that memory. During every load and store, the hardware checks whether the pointer's tag matches the tag stored for the corresponding memory granule; mismatches trigger a fault. This mechanism probabilistically detects both temporal and spatial errors. For instance, with 4-bit tags (TS = 4), the probability of catching a violation is approximately 94% [8].

Two notable implementations of hardware and hybrid memory tagging have been presented and evaluated:

SPARC ADI (Application Data Integrity): A fully hardware-based tagging mechanism available on SPARC M7/M8 CPUs running Solaris. It associates 4-bit tags with 64-byte memory regions and raises precise or imprecise exceptions on tag mismatches. Tag manipulation is performed using special instructions (stxa variants), and tagged memory must be explicitly mapped using the MAP_ADI flag. Tags are stored in hardware (likely within ECC bits), making them inaccessible to user software [3, 8].

AArch64 HWASAN (Hardware-Assisted AddressSanitizer): A compiler-based approach leveraging the ARMv8-A top-byte-ignore feature. It stores 8-bit tags in the upper byte of pointers and keeps corresponding memory tags in a directly mapped shadow memory region (one tag per 16 bytes). The compiler instruments load/store operations to perform tag checks, achieving partial hardware acceleration. HWASAN significantly reduces ASan's runtime overhead while serving as a prototype for fully hardware-assisted memory tagging [6, 8].

While these designs demonstrate that memory tagging can greatly enhance software reliability, their hardware and compiler dependencies limit deployment across architectures. Additionally, trade-offs between tag granularity (TG), tag size (TS), and storage overhead influence both performance and detection precision. These studies underscore the need for further architectural exploration of memory tagging mechanisms, particularly in open and extensible ISAs like RISC-V, where designs such as the Memory Tagging Extension (MTE) and its Virtually Indexed Tag Table (VITT) propose virtual-memory–based tag management.

For our project, we build on a prior work called ZIMTE [2], an implementation of memory tagging extensions for RISC-V built for the QEMU emulator. QEMU provides limited performance metrics, and we are interested in evaluating the performance of the MTE and VITT at the cycle-level granularity. Thus, our project aims to implement the ZIMTE extension within the more robust XiangShan GEM5 RISC-V emulator in order to quantify the microarchitectural impact of VITT operations on execution time, cache behavior, and memory latency.

3 Architectural Implementation of Memory Tagging

To support the ZIMTE extension within the XiangShan GEM5 simulator, we implemented the requisite architectural state, control logic and instruction set extensions. This design required modifying the RISC-V decoder, the Control and Status Register (CSR) dile, and the load/store execution unit to support Virtually Indexed Tag Tables (VITT).

3.1 CSRs and VITT Structure

The core of the ZIMTE protection mechanism relies on the Virtually Indexed Tag Table (VITT), a software-managed data structure that maps virtual addresses to their corresponding memory tags [2]. To support VITT lookups in hardware, we added specific privileged CSRs to the GEM5 `RiscvISA` class as specified in the ZIMTE specification:

- **`menvcfg` (Environment Configuration):** Modified to include the Memory Tagging (MT) enable bit. This allows the kernel to globally enable or disable tag checks [2, 5].
- **`mvitt` (VITT Base Address):** A new custom CSR that stores the virtual base address of the tag table. Hardware tag checks utilize this register to calculate the offset of a tag for a given virtual memory address [5, 6].

Since we are in a bare-metal environment, we are using these two registers for all modes (U/S/HS/M).

3.2 ZIMTE Instruction Implementation

We implemented the three primary instructions defined by the ZIMTE specification to manage tag generation and storage.

3.2.1 `gentag` *and* `addtag`.

The `gentag` (Generate Tag) instruction was implemented as an arithmetic operation that generates a pseudo-random tag and inserts it into the upper bits of a destination register. Similarly, `addtag` preserves the tag in the upper bits of a pointer while adding an immediate offset to the lower address bits. As these operations are strictly arithmetic and register-local, they were implemented using standard GEM5 logic without requiring memory subsystem interactions [10].

3.2.2 `settag`.

The `settag` instruction is responsible for updating the VITT with the tag currently held in a register. Unlike the arithmetic instructions, `settag` functions as a memory store operation [4].

Implementation challenge: In GEM5's Out-of-Order (O3) CPU, store operations are complex. We discovered that the decoder implicitly splits stores into separate address-calculation and data-write microoperations [9]. Initially, this caused execution misalignments where our custom `settag` logic would abort unpreditably. We resolved this by defining `settag` as a custom store implementation that bypasses standard splitting where necessary, ensuring the tag write to the VITT address is atomic and correctly ordered.

3.3 Checked Load/Store and Micro-op decomposition

The most significant architectural challenge was implementing Checked Loads and Stores. These operations must:

(1) Calculate the target virtual address.
(2) Calculate the corresponding VITT entry address.
(3) Load the reference tag from the VITT.
(4) Compare the refrence tag with the pointer's tag.
(5) Fault on mismatch or proceed with memory access on match.

3.3.1 Object-Oriented Macrooperations. To represent this complex sequence in GEM5's Object-Oriented C++ infrastucture, we could not simply add a "check" flag to the existing Load/Store classes. The 03 pipeline requires precise instruction scheduling.

We utilized GEM5's **MacroOp** (Macro-Operation) desing pattern. We defined `CheckedLoad` and `CheckedStore` as subclasses of `RiscvMacroInst`. When the decoder encounters a checked instruction, it does not issue a single operation to the execution unit. Instead, it expands the MacroOp into a sequence of MicroOps:

- **Tag Load MicroOp:** Fetches the expected tag from the VITT.
- **Check/Fault MicroOp:** Compares the tags and raises a `MemoryTaggingFault` if they differ.
- **Data Access MicroOp:** Performs the actual user-requested load or store.

This decomposition allows the O3 scheduler to handle dependencies correctly, ensuring that the data access does not commit if the tag check fails.

4 Results

We successfully verified the functional correctness of the new instructions present in the ZIMTE extension: `genttag`, `addtag`, and `settag`. To do this, we created a bare-metal application capable of running on XS-GEM5 that executes these instructions and validates expected behaviors. This section details the process of adding memory tagging support to the infrastructure for building bare-metal applications within XiangShan, and illustrates the successful operation of the implemented instructions.

4.1 Experimental Infrastructure

All experiments were conducted on the CloudLab research platform to ensure reproducibility and access to high-performance compute resources. We utilized a `c6525-25g` instance type with the following specifications:

- **Processor:** AMD EPYC 7302P (16 Cores, x86_64)
- **Memory:** 25 GB RAM, 100 GB Storage
- **OS:** Ubuntu 22.04 LTS

We developed automated shell scripts to standardize the deployment of the XiangShan-GEM5 (XS-GEM5) simulator and the RISC-V GNU cross-compiler toolchain. The environment was configured to support both Newlib (for bare-metal) and Linux-gnu toolchains.

4.2 ZIMTE Tool-chain Integration

The XiangShan project's simulation environments (including GEM5) only provide support for workloads in a flat binary format; thus, they provide Abstract Machine (AM) as a software tool to compile compatible workloads from C source code. AM acts as a lightweight bare-metal runtime library that abstracts hardware details, allowing C programs to be compiled directly into memory images loadable by XS-GEM5. In order for AM to support the creation of a memory-tagging-enabled workload, we had to modify the `riscv-gnu-toolchain`, which AM employs to cross-compile for its environment. We relied heavily on the existing work of Christoph Müllner, integrating his changes to `binutils`, `gcc`, and `glibc` for ZIMTE support into the `riscv-gnu-toolchain` and AM.

4.3 Bare-metal Workload

With these modifications, we created a bare-metal application written primarily in RISC-V assembly with a C wrapper, which verified the functional correctness of our implemented ZIMTE instructions. In this section, we provide a short description of each ZIMTE feature tested with our application and show the resulting output when run on GEM5.

`GENTAG` and `ADDTAG`: These instructions manipulate the target register to contain a four-bit tag in the top four bits. The `gentag` instruction does this randomly, whereas the `addtag` instruction takes in a register that already contains a tag and places the sum of this tag and a provided immediate into its destination register's top 4 bits. We tested these instructions on the pointer variables `gentag_ptr` and `addtag_ptr` and printed their values before and after executing the respective instructions.

```
-----------------------------------------------------
Testing GENTAG
        genttag_ptr: 0000000080009FB8
        executing "gentag genttag_ptr, zero"
        genttag_ptr: 1000000000000000
Testing ADDTAG
        addtag_ptr: 0000000000000000
        executing "addtag addtag_ptr, genttag_ptr, 1"
        addtag_ptr: 2000000000000000
-----------------------------------------------------
```

Both instructions clear and insert a tag into the top 4 bits of the register (pointer) as expected.

Enabling Memory Tagging: In order to enable memory tagging modes in various privilege modes, the ZIMTE extension adds two bits termed `MT_MODE` to the `envcfg` control status registers (CSRs). These two bits control whether memory tagging is enabled in the next privilege mode as well as the bit-width of the tag. Furthermore, the SVATAG [2] extension defines that tag storage is implemented as a large array in the virtual address space of the execution environment and defines the `mvitt` CSR which stores the base of tag storage. As mentioned in Section 3 for the purposes of testing we are using these as global CSRs to define memory tagging properties for all modes.

We tested the modified CSR configurations from these extensions by writing and reading to them via the `csrr` and `csrw` instructions. Furthermore, during testing, we realized it was pivotal to ensure virtual address translation was enabled during this process as it is not enabled by default.

```
------------------------------------------------------
Setting Virtual Address Translation Mode
        reading: satp = 0x0
        writing: satp = 0x9000000000080100
        reading: satp = 0x9000000000080100
        Translation mode: 9
        Sv48 address translation ENABLED
Enabling MT_MODE
        read menvcfg=0000000000000000
        writing menvcfg=0000000800000000
        read menvcfg=0000000800000000
Setting the VITT base register
        read mvitt=0000000000000000
        writing mvitt=0000000080100000
        read mvitt=0000000080100000
------------------------------------------------------
```

All of these modifications aim to set up VITT tag lookup and storage for the ZIMTE extension.

SETTAG: This instruction takes a tag from the provided pointer argument, calculates the correct place in virtual memory that corresponds to the pointer's VITT entry, and writes the tag to that location. It is important to note that the instruction presumes that the provided register is a pointer with a properly set up tag from either `gentag` or `addtag`. We tested these instructions on the pointer variable `gentag_ptr` and verified `settag` was correctly writing to the VITT by computing the virtual address the `settag` write should target, and then reading before and after the instruction. It is important to note that in a final implementation, the hart should raise an access fault if there is a load/store to a virtual address in the VITT range.

```
------------------------------------------------------
Testing SETTAG
        genttag_ptr set to 1000000080009FB8
        reading from vitt address 00000000841004FD: 0
        executing "settag genttag_ptr, 0"
        expecting tag=1 @ vaddr=00000000841004FD
        reading from vitt address 00000000841004FD: 1
------------------------------------------------------
```

The expected tag to be written into the VITT entry calculated above was 1; as shown, before `settag` was executed the entry was 0, and afterwards it changed to 1.

5 Challenges

We encountered significant technical challenges during the development of this project. From working with a large and unfamiliar codebase such as XiangShan GEM5 to assembling many complex and custom dependencies, the following highlights some notable challenges surrounding this project.

5.1 XiangShan Infrastructure and C++ Templating

One of the biggest challenges we ran into was simply navigating the XiangShan / gem5 infrastructure and its heavy use of C++ templating, especially given that we started with very little C++ background and only limited computer architecture experience. When it came to adding an instruction, it meant understanding how *.isa* descriptions expand into auto-generated C++ classes, how those classes interact with the out-of-order pipeline, and how flags and templates subtly change behavior. For example, when we first implemented *settag*, the instruction appeared to execute, but it consistently wrote the wrong value into memory. After a lot of digging, we discovered that the *IsSplitStoreAddr* flag was causing the store to be treated as a split address/data micro-op pair, even though our instruction only ever produced a single micro-op. This mismatch silently broke the store and resulted in zeros being written instead of the computed tag. A similar kind of complexity showed up when we tried to extend the load/store logic to perform tag checks. Conceptually, we wanted to load the tag, compare it, if it matched, load the data, and throw an exception if it the tags didn't match. But in the gem5 timing and out-of-order infrastructure, that effectively means doing a load inside another load, within a single instruction, which clashes with the core assumptions of the out-of-order pipeline and the Load/Store Queue. That pushed us deeper into thinking in terms of micro-ops, multi-phase memory requests, and how apparently simple semantics actually need to be split across multiple pipeline stages and helper templates.

5.2 Full-System Stack Configuration

A critical goal of this project is to characterize VITT overhead within a realistic userspace environment. This requires booting a Linux kernel on XS-GEM5. We attempted to reproduce the build environment for a known working binary (`linux.bin`) provided by XiangShan. Our methodology shifted from following deprecated documentation to a "version-matching" strategy, where we reverse-engineered the components of the working binary.

5.2.1 *Component Versioning.*

We identified and attempted to integrate the following specific version stack:

- **Bootloader:** OpenSBI v1.4 (Generic Platform)
- **Kernel:** Linux v6.1.83 (configured with `xiangshan_defconfig`)
- **Userspace:** A minimal `initramfs` containing a statically linked "Hello World" executable.

5.2.2 *Build Process and Debugging.*

To ensure compatibility, we extracted the Device Tree Blob (DTB) directly from the reference `linux.bin` using `binwalk` and `dd`, rather than generating a potentially incompatible one from source.

During the compilation of OpenSBI v1.4 with modern GCC toolchains, we encountered type definition conflicts regarding the C boolean type in `sbi_types.h` and `semihosting.c`. We applied source-level patches to manually define these types and modify function signatures, enabling a successful build of the firmware payload (`fw_payload.bin`).

5.2.3 *Current Status.*

The resulting payload successfully initializes OpenSBI and hands off execution to the Linux kernel. However, the system currently stalls during the kernel hardware initialization phase. This suggests a lingering mismatch between the extracted Device Tree and the specific hardware configuration parameters expected by the kernel version. Future work will focus on isolating the specific device driver causing the panic. For an in-depth step-by-step tutorial of all the steps we tried for running Linux, please refer to this **branch** of the repository,

5.3 Miscellaneous Hurdles

We also ran into simpler hurdles such as being unfamiliar with the development environment and understanding how simple debug prints worked. We ended up having to create our own printing class, allowing us to efficiently filter out irrelevant prints and speed up debugging our implementation. Other problems also included assumptions that we were not aware of, such as the default memory mode within GEM5 being *User Mode*. It wasn't until later, after robust testing, that we found out the default *User Mode* meant no virtual memory and direct physical memory access. Given the nature of our project, this was not the desired simulation behavior, resulting in us having to manually set the memory mode later on when testing our userspace program.

6 Future Work

As future work, we are interested in exploring and deciding on an architectural and implementation decision for including tag checks within load/store instructions. By coming out of this project having learned much of XS-GEM5's architectural infrastructure along with C++ templating knowledge, we look forward to exploring deeper architectural modifications such as micro-operations and multi-phase memory requests. In terms of workloads, we have only been calling our added memory tagging instructions manually in C programs so far. The next goal would be to modify Abstract Machine to insert our added memory tagging instructions throughout all compiled programs, allowing us to fully test memory tagging in action. After that, we are also interested in finding a solution for running Linux on gem5. We have already made a post on XiangShan's GitHub regarding this issue and look forward to working towards a solution.

References

[1] Nurit Dor, Michael Rodeh, and Mooly Sagiv. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 155–167.

[2] RISC-V International. 2025. *RISC-V Memory Tagging Extension Specification*. Technical Report.

[3] Georgios K Konstadinidis, Hongping Penny Li, Francis Schumacher, Venkat Krishnaswamy, Hoyeol Cho, Sudesna Dash, Robert P Masleid, Chaoyang Zheng, Yuanjung David Lin, Paul Loewenstein, et al. 2015. SPARC M7: A 20 nm 32-core 64 MB L3 cache processor. *IEEE Journal of Solid-State Circuits* 51, 1 (2015), 79–91.

[4] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[5] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[6] Jiwon Seo, Junseung You, Yungi Cho, Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. 2023. Sfitag: Efficient software fault isolation with memory tagging for arm kernel extensions. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 469–480.

[7] Kamil Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*. 309–318.

[8] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517

[9] The gem5 Project. 2025. O3CPU Model. https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU Accessed: Dec. 8, 2025.

[10] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 177–189. doi:10.1145/3579856.3590331

[11] Irene Wang, Prasenjit Chakraborty, Zi Yu Xue, and Yen Fu Lin. 2022. Evaluation of gem5 for performance modeling of ARM Cortex-R based embedded SoCs. *Microprocessors and Microsystems* 93 (2022), 104599.