Extending RISC-V Keystone to Include Efficient Secure Memory

Zach Moolman
zach.moolman@colorado.edu
University Of Colorado Boulder
Boulder, Colorado, USA

Tamara Silbergleit Lehman
tamara.lehman@colorado.edu
University Of Colorado Boulder
Boulder, Colorado, USA

Abstract

Given that mobile and embedded devices are at the center of day to day activities, they are often the target of cyber attacks. Despite their heightened security criticality, these devices do not often protect their data in memory. The reason for this lack of protections is resource limitations. In this work we propose an efficient mechanism to extend the Trusted Execution Environment (TEE) in RISC-V, Keystone, to include Secure Memory features to protect data in memory from physical and remote memory attacks.

1 Introduction

Embedded and mobile devices have infiltrated most day-to-day operations. As a result, they are responsible for handling sensitive information, such as passwords, social security numbers, credit cards, among many others. Since embedded and mobile devices are typically not housed within robust physical protections, they are vulnerable to memory attacks. Memory attacks refer to physical or remote attacks that leverage the vulnerable nature of memory devices to extract sensitive information or change values in memory to gain unauthorized access to the system.

Typical embedded and mobile devices are not equipped to provide protection for memory attacks, given they have limited re sources. Typical protection mechanisms for data in memory involve both encrypting and integrity protecting the data. To encrypt the data before going to memory, the system needs to include a hard ware encryption engine. This additional circuit is typically found in SOC but usually used for protecting data in storage only, not memory. One reason for this design decision is likely due to resource and latency limitations, as memory accesses already take a long time to retrieve data, and this latency is on the critical path of computation. For integrity verification, secure systems use cryptographic hashes stored in memory alongside the data. This approach to integrity verification requires too much memory overhead and additional hashing latency on the critical path to retrieve data from memory.

In this work, we explore existing optimization mechanisms to alleviate the overheads of secure memory and extend the proposed RISC-V Keystone features to aid in secure memory operations. Our goal is to produce an efficient implementation of a trusted execution environment (TEE) with Secure Memory suitable for embedded and mobile devices.

In this work, we are looking to address the performance and resource limitations of secure memory systems by investigating the potential for the RISC-V TEE, Keystone, to enable secure memory features efficiently. As part of our research, we are considering extending the Physical Memory Protections (PMP) table to include some of the secure memory metadata. This extension is a significant step in alleviating the overhead. Furthermore, we are actively working on implementing a prototype on an FPGA, a crucial step in exploring the feasibility of the design in a real-world context.

Threat Model.

The proposed security mechanism is based on the threat model that assumes an adversary has remote (or physical) access to the machine and can probe the hardware to infer information about the victim program. The adversary can run any program they desire on the target machine to subvert the processor to leak secrets through various vulnerabilities (including privilege escalation and physical attacks). These vulnerabilities are well-understood and part of the threat model, which has been thoroughly studied [1, 8–13, 16–18]. The traditional solution to protect against these types of attacks involves secure memory and TEE.

2 Background

Trusted Execution Environment (TEE), such as Intel Software Guard eXtensions (SGX) and ARM Trustzone commonly found in modern processors, can boost the security of end-user devices by isolating the execution of well-defined programs [1, 2, 5]. The idea behind a TEE is that anything coming from outside the chip boundary is untrusted, including the kernel data and code, as a kernel can be taken over by a malicious party and modified to steal sensitive data. TEEs offer protection at the hardware level to prevent rogue access or compromise. TEE's main goal is to secure a system without trusting the privileged software (such as the operating system). Applications developed for the TEE, need to have a restricted interface with the insecure world. It's important to remember that strict software practices are not just recommended, but crucial for passing data in and out of the trusted boundary. This responsibility lies with the developers and users of TEEs, reinforcing the importance of these practices in maintaining the security of the system.

TEEs also often have the benefit of remote attestation [1, 4]. When using the TEE technology, a program running in a remote device can attest to the security of the processor in which it is running, as well as the security of the binary before it runs. Remote attestation uses a hardware-embedded cryptographic key, machine-only readable, to derive keys that can prove that a machine has the necessary features to run a program securely. For example, Intel processors with SGX enabled have an embedded key inside each processor that can prove its identity as a verified Intel processor. This remote attestation can be leveraged to verify the integrity of the software by signing it with a private key to verify its origin. The integrity verification of the binary is done by comparing a signature previously recorded by the developer with the recently computed signature when the binary is loaded [10, 15].

TEEs alone cannot protect data in memory. The TEE is capable of protecting the execution environment but it does nothing to protect the data once it is at rest in memory. Physical and remote attacks have been shown to be effective in extracting data even in the presence of TEEs [6]. As a result, we want to investigate secure memory features to protect the data in memory.

Secure Memory. Data in memory should retain its confidentiality property to detect and prevent memory attacks by an attacker. An attacker should not be able to read the values that cross the communication bus between the processor and the memory. Confidentiality prevents sensitive data from being disclosed. However, confidentiality alone does not prevent an active attack, in which the attacker modifies the values in memory to control the program's behaviour. To protect against active memory attacks, secure memory systems should also guarantee data integrity, which is the property that ensures that no data is modified in an unauthorized way.

Systems with secure memory often include a hardware encryption engine that enables the encryption of the data before it is stored in the main memory. When the data is fetched from the main memory, in its encrypted form, it is first decrypted and only then forwarded upstream to the cache hierarchy and the processor for execution. When the data is updated and sent to the main memory, the value is re-encrypted, and its ciphertext is placed in the main memory. Encrypting data in memory avoids unauthorized access to the secure memory region, as accessing it without the proper channels would simply return the ciphertext. The cryptographic keys used throughout the secure hardware are guarded within the trusted execution environment and tied to a particular thread.

Typical secure memory systems use counter-mode encryption due to its ability to parallelize the data fetch with the encryption itself. Counter mode works well for secure memory because it is able to hide the slow part of the encryption process while the memory controller fetches the data from memory. While the memory controller sends the request to fetch the data from memory, the AES hardware engine computes a one-time-pad (OTP) for the particular block to then XOR it with the ciphertext to decrypt the data or the plaintext to encrypt it. To ensure each block of data has a temporal and spatial unique OTP, counters are assigned to each block and incremented on an update to the corresponding block. Given that counters are larger than what can be held on chip, they are normally placed in memory in a specific address, that can be calculated based on the physical address of the block the counter protects.

To enable integrity verification, cryptographic hashes (HMAC) are used as a signature for what the data should be. When the processor writes to memory, the memory controller computes the corresponding HMAC and stores it in memory alongside the data. Efficient implementations of secure memory use the Error Correcting Code (ECC) chip of the memory device to store the HMAC in order to save the additional memory access to fetch it [14]. When fetching data from memory, the memory controller recomputes the hash of the data and compares it to the previously stored hash. If the hashes are the same, the integrity of the data is verified and it is forwarded to the processor to continue execution.

HMACS alone cannot protect the system against replay attacks, where the attacker records both the HMAC and the data and then replays them both simultaneously at a later point in time. To provide detection and protection against replay attacks, secure memory systems usually use Bonsai Merkle trees (BMT), a hash tree built over the encryption counters [11]. The BMT establishes the root of trust by storing the root of the tree always on chip. When a piece of data is fetched from memory, the tree is traversed from leaves to the root, comparing hashes along the way, to verify the integrity of the data.

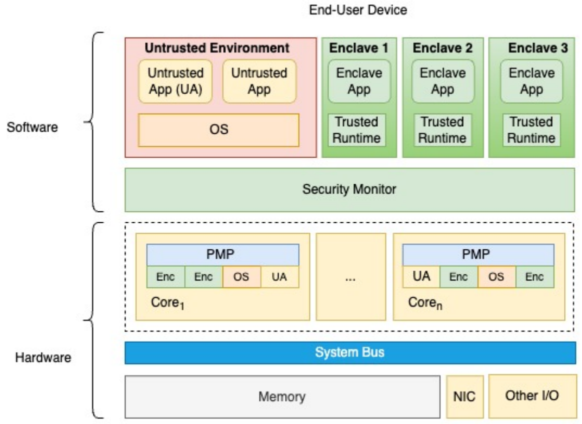


Figure 1: RISC-V Keystone

All of these features of secure memory can be leveraged to improve the security of an embedded or mobile device running untrusted programs. The confidentiality and integrity of TEEs can prevent the device from allowing rogue applications from accessing sensitive information. Furthermore, the integrity of the binary in the TEE can prevent adversaries from modifying the functionality of trusted programs. The remote attestation feature allows trusted devices to verify that they have a secure environment to prevent man-in-the-middle attacks.

RISC-V TEE The RISC-V community proposed a new style of TEE, Keystone, which allows for hardware developers to customize the interface of the TEE with the rest of the system [7]. RISC-V Keystone achieves this flexibility by introducing the idea of the machine mode. This mode is an elevated privilege execution mode that can manage the memory partitions (including ranges and access permissions) for the whole system. The different memory partitions are tracked in a structure called the physical memory protections (PMP) table.

The RISC-V Keystone TEE isolates memory by using the PMP table, which establishes permissions and priority of access for the different computing modes (user, supervisor and machine mode). Software running in machine mode, called the security monitor, is responsible for managing the context switches, memory accesses, and establishing access control primitives. Each core has its own PMP table, which describes the access privileges for each memory range. The PMP table is checked any time there is a memory access in the user or supervisor mode. Given that the PMP table is implemented as fast access registers, they do not add significant overhead to the memory accesses, even when accessing the first level cache. The system design of RISC-V Keystone is shown in Figure 1.

3 System Design

Our main contribution is to extend the physical memory protection (PMP) table described in the RISC-V privilege specification to include secure memory. To accomplish the necessary modifications, we propose a new structure in the memory controller, which contains a copy of some of the information in the per core PMP tables, aggregated to create a system-wide view called the extended PMP (ePMP). In addition, we proposed to include a memory encryption engine that extends the memory controller with the logic necessary to encrypt and decrypt, compute secure hashes, and decide when the security features need to be applied.

The MEE constructs an integrity tree per memory region to enable the dynamic allocation of the secure memory region described in the PMP tables. The MEE then tracks the root of each tree in its own table (the aggregated ePMP table), indexed by the PMP entry table index and the core ID that made the memory request.

A key feature of the proposed design is that we can use PMP entries without relying on a specific TEE implementation. Secure memory is orthogonal to TEE, allowing for much more flexible implementations. On the other hand, it can seamlessly be integrated with Keystone without any changes to the Keystone TEE implementation.

3.1 The Aggregated Extended PMP Table

The proposed aggregated extended PMP (ePMP) table residing in the memory controller incorporates metadata for enabling an efficient and dynamic implementation of secure memory (shown in Figure 2). The circled numbers on the figure is to match the components within the FPGA design to the security metadata stored in memory shown on the right side of the figure.

The ePMP status registers enable the dynamic allocation of secure memory regions. To accomplish this dynamicity, the memory controller needs to be aware of any new ranges defined by the core. We propose creating the aggregated ePMP table at the memory controller to enable this feature. The MEE will monitor all data requests as they pass through the MEE and update the meta entries in the ePMP according to their respective entries in the PMP of each core.

To be able to discretize the differing memory regions, the Bonsai Merkle tree needs to be built separately and independently for each region. In order to implement multiple independent Bonsai Merkle trees, the aggregated PMP table in the memory controller needs to also include the corresponding tree root, which needs to reside on-chip at all times. The tree root can be a small 8B HMAC that encompasses all the values in the memory region it protects.

In addition to the tree root, given that Bonsai Merkle trees have fixed mappings from data to metadata based on the physical address and size of the secure memory region, the aggregated PMP table needs to also track the size of the memory region. To track this information in a concise format, we synthesize this information into the field which we call the tree mode. The tree mode is used to find the corresponding mapping from data to metadata. The tree mode is simply the number of levels in the integrity tree based on the size of the memory range. When we know how many levels in the tree we have, then we can derive the mapping of data to metadata by simply using the data physical address. The size of each memory range can be easily derived by the core at the time of the memory range entry creation. The core communicates the memory range size to the memory controller when the security monitor creates the new range in the M mode.

3.2 The Memory Encryption Engine

The memory encryption engine (MEE) extends the original memory controller with an encryption engine, capable of performing cryptographic operations, the logic necessary to decide when a memory request needs the security features and the aggregated ePMP table. The MEE monitors all data requests that come into the memory controller. Requests that fall in a memory address range that belongs to a secure environment will necessitate to go through the security features of secure memory: decryption for reading from memory, encryption for writing to memory and integrity verification for both. To allow the MEE to make this decision without completely replicating the information on the core's PMP table, we expand the communication protocol (TileLink) to indicate when an address falls within a secure memory region by simply adding a secure bit to the request. Given that the MEE needs to be aware of any changes to each of the cores PMP tables, the communication protocol is also expanded to include two new commands.

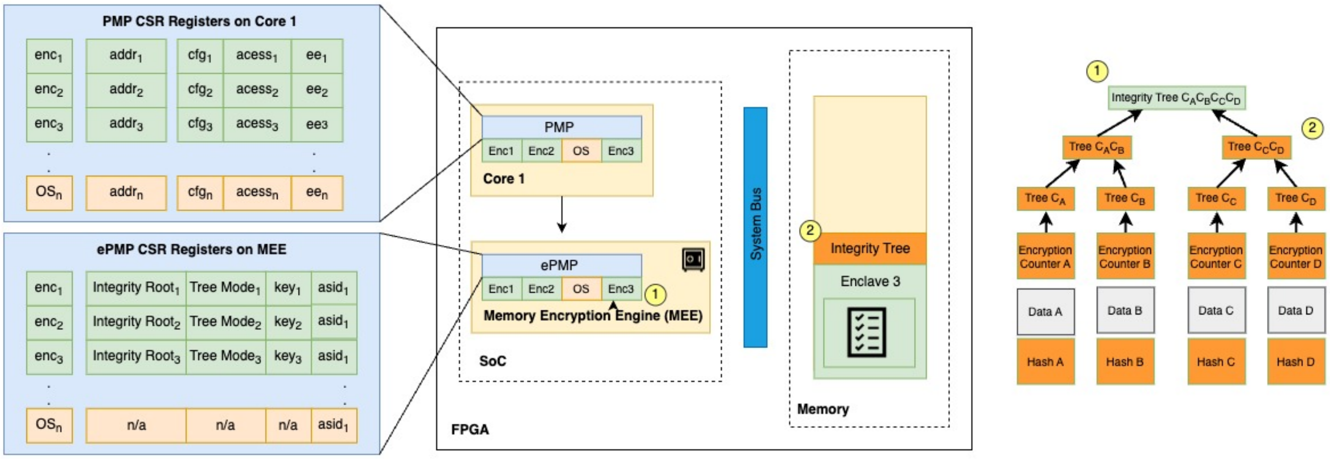
In order to synchronize the information in the cores' PMP tables and the information at the MEE, the communication protocol is extended to include two new types of commands: create new entry and delete entry. The create new entry command is initiated at the core whenever the security monitor creates a new secure memory range in the PMP table. This command needs to communicate to the MEE which entry in its own version of the ePMP table needs to be updated. To communicate which entry, we concatenate the core's PMP entry index with the core identifier. The MEE, once it receives this command, updates the corresponding entry by resetting the root of the integrity tree and the tree mode size to the corresponding mode according to the new range size. The range size information is embedded into the command's payload.

The second command, the delete entry, is initiated by the core when the security monitor in the machine mode removes one of the ePMP entries. Once the MEE receives this type of command, it resets the integrity tree root and mode to zero to indicate that the corresponding entry is invalid.

The memory encryption engine needs to include NxP entries on a chip, where N is the number of cores in the system, and P is the number of PMP entries in each core. Typical designs include up to 16 entries for the per-core PMP entries. For a system with eight cores, the MEE will require a total of 128 entries in its table. Each entry contains the root of the BMT for the region (up to 8B of data) and the tree mode (a 1B value, which allows for 256 different modes). With these values, the total amount of additional on-chip storage is a little over 1KB (9B * 128 = 1152 B), a minimal overhead to incur for the sake of securing the data in memory.

3.3 Modifications to the Security Monitor

In addition to the extension of the PMP table, the security monitor needs to be modified to be able to allocate a memory range that includes the additional space required to store the security metadata in contiguous memory. When the security monitor requests a range



PMP CSR Registers on Core 1

enc_1	$addr_1$	cfg_1	$acess_1$	ee_1
enc_2	$addr_2$	cfg_2	$acess_2$	ee_2
enc_3	$addr_3$	cfg_3	$acess_3$	ee3
.				.
OS_n	$addr_n$	cfg_n	$acess_n$	ee_n

ePMP CSR Registers on MEE

enc_1	Integrity $Root_1$	Tree $Mode_1$	key_1	$asid_1$
enc_2	Integrity $Root_2$	Tree $Mode_2$	key_2	$asid_1$
enc_3	Integrity $Root_3$	Tree $Mode_3$	key_3	$asid_1$
.				.
OS_n	n/a	n/a	n/a	$asid_1$

Figure 2: High Level Design

of memory from the Operating System(OS), it normally requests the same amount that was requested by the user creating the new secure environment. The security monitor can be extended to request of the untrusted OS the original size with additional space (in traditional systems this additional space required is about 20%) to accommodate the metadata [11]. The 20% comes from needing a total of 1.5% for the encryption counters, 12.5% for the data hash, and 6% for the integrity tree.The additional space should not be readable or writable by any mode, as no application (not even ones running on the secure environment) should be able to address these locations explicitly. The only component able to operated on these values is the memory controller which does not go through the PMP checker.

By creating an independent integrity tree per memory region, the system can alleviate the overhead by only reserving the necessary space for the metadata. For example, the operating system region, which is assumed to be all of memory at system boot, requires no secure memory metadata, as it is assumed that the OS is not trusted. Therefore, when no secure environment exists, there will be no region in memory reserved for metadata, as this will not be needed by any of the regions.

4 Prototype and Future Experimentation

To build our secure System-on-Chip (SoC) we are expanding RocketChip, an open-source System-on-Chip generator along with Berkeley Out-of-Order Machine (BOOM) core, a synthesizable and parameterizable open-source RV64GC RISC-V core [3, 19]. These tools leverage Chisel, a high-level hardware construction language derived from the Scala language that emits synthesizable RTL. We leverage RocketChip to generate the modified SonicBoom (BOOMv3) cores, caches, and memory encryption engine (MEE) and interconnects to design an integrated SoC.

Figure 3 shows the changes we make to the RochetChip to implement the proposed design. We extend the BOOM core Control Status Record(CSR) to indicate whether a memory region should be encrypted or not. The pmpchecker validates each memory access requested by the TLB. We have updated the pmpchecker to obtain the encryption-enabled signal that will be sent with any memory request. RocketChip uses the TileLink protocol as its primary interconnect. We update the protocol to include the encryption-enabled signal. We also extend the memory controller with a Memory Encryption Engine. The memory encryption Engine examines each request. If the request does not require data encryption, the request bypasses the Encryption Engine(EE). This is true for reading or writing requests. If an encryption-enabled signal is asserted on write request, data is encrypted before it leaves the memory encryption engine. On the other hand, read requests are forwarded to

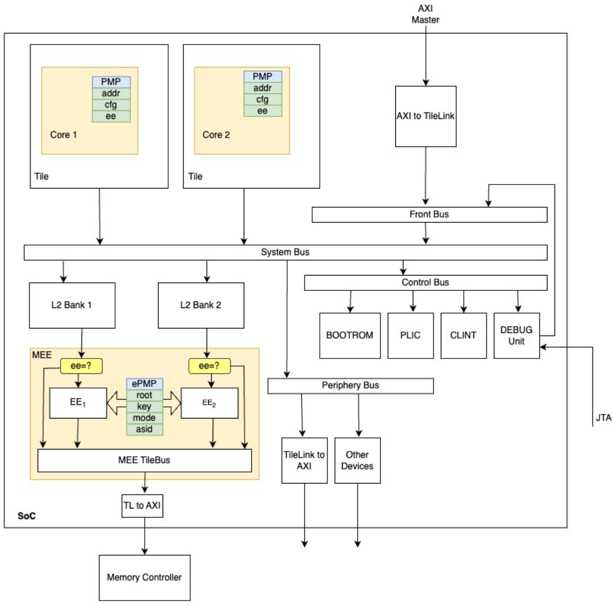


Figure 3: SoC incorporates our Memory Encryption Engine (MEE). The colored portion indicates addition and changes made to the original design.

the external memory controller. Once the read request is available, the Encryption Engine decrypts the data using the metadata stored in the ePMP table.

5 Conclusions and Future Work

Given the fact that embedded and mobile devices are entrenched in our day to day activities, they are bound to operate on sensitive data. As a result, these devices need to offer adequate protection mechanisms to protect the system against memory attacks and rogue operating systems. Existing approaches to provide defenses against this threat model require too many resources or incur significant performance overheads, making them an inadequate option for these smaller devices. The proposed design has the promise of enabling a robust and efficient secure system which could enable sensitive computations on mobile and embedded devices. The opportunities that RISC-V introduces can be further leveraged by the proposed design to alleviate the memory spatial overhead requirements of BMTs and improving the performance of existing approaches of secure memory. The proposed design is ongoing work and we will publish the performance results once we get a fully working prototype.

References

[1] Anati, I., Mckeen, F., Gueron, S., Haitao, H., Johnson, S., Leslie-Hurd, R., Patil, H., Rozas, C., and Shafi, H. Intel software guard extensions (Intel SGX). In *Tutorial at International Symposium on Computer Architecture (ISCA)* (2015).

[2] ARM. Arm architecture reference manual.

[3] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 4* (2016), 6–2.

[4] Costan, V., and Devadas, S. Intel SGX explained. Tech. rep., Cryptology ePrint Archive, Report 086, 2016.

[5] Holdings, A. Arm security technology: Building a secure system using trustzone technology, 2009.

[6] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Computer Architecture News* (2014).

[7] Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., and Song, D. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[8] Lehman, T. S., Hilton, A. D., and Lee, B. C. PoisonIvy: Safe speculation for secure memory. In *International Symposium on Microarchitecture (MICRO)* (2016).

[9] Lehman, T. S., Hilton, A. D., and Lee, B. C. MAPS: Understanding metadata access patterns in secure memory. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2018).

[10] Lie, D., Chandramohan, T., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. *SIGPLAN Notices* (2000).

[11] Rogers, B., Chhabra, S., Prvulovic, M., and Solihin, Y. Using address independent seed encryption and Bonsai Merkle trees to make secure processors OS- and performance-friendly. In *International Symposium on Microarchitecture (MICRO)* (2007).

[12] Rogers, B., Prvulovic, M., and Solihin, Y. Efficient data protection for distributed shared memory multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2006).

[13] Rogers, B., Yan, C., Chhabra, S., Prvulovic, M., and Solihin, Y. Single-level integrity and confidentiality protection for distributed shared memory multiprocessors. In *International Symposium on High Performance Computer Architecture (HPCA)* (2008).

[14] Saileshwar, G., Nair, P. J., Ramrakhyani, P., Elsasser, W., and Qureshi, M. K. Synergy: Rethinking secure-memory design for error-correcting memories. In *International Symposium on High Performance Computer Architecture (HPCA)* (2018).

[15] Shi, W., Lee, H.-H. S., Ghosh, M., and Lu, C. Architecture support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2004).

[16] Suh, G. E., Clarke, D., Gassend, B., Van Dijk, M., and Devadas, S. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing (ICS)* (2003).

[17] Thomas, S., Workneh, K., Ishimwe, A.-T., McKevitt, Z., Curlin, P., Bahar, R. I., Izraelevitz, J., and Lehman, T. Baobab merkle tree for efficient secure memory. *IEEE Computer Architecture Letters* (2024).

[18] Thomas, S., Workneh, K., McCarty, J., Izraelevitz, J., Lehman, T., and Bahar, R. I. A midsummer night's tree: Efficient and high performance secure scm. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2024), pp. 22–37.

[19] Zhao, J., Korpan, B., Gonzalez, A., and Asanovic, K. Sonicboom: The 3rd generation berkeley out-of-order machine.