Baobab Merkle Tree for Efficient Secure Memory

Samuel Thomas, Kidus Workneh, Ange-Thierry Ishimwe, Zack McKevitt, Phaedra Curlin, R. Iris Bahar, Joseph Izraelevitz, and Tamara Lehman

***Abstract*—Secure memory is a natural solution to hardware vulnerabilities in memory, but it faces fundamental challenges of performance and memory overheads. While significant work has gone into optimizing the protocol for performance, far less work has gone into optimizing its memory overhead. In this work, we propose the *Baobab Merkle Tree*, in which counters are memoized in an on-chip table. The Baobab Merkle Tree reduces spatial overhead of a Bonsai Merkle Tree by 2-4X without incurring performance overhead.**

***Index Terms*—Security, secure memory, encryption, integrity.**

I. Introduction

With the growing use of remote services for computation on personal data, the issue of providing security and privacy has gained growing importance. When clients offload sensitive information to a remote machine, they do it in trust that they are protected from several attacks orchestrated by an untrusted OS or cloud administrators [1]. While subject to remote computation, some level of protection must be employed to compute on sensitive data such as encryption keys, genetic information, blockchain transactions, etc.

In most scenarios today, this protection is guaranteed through secure computation solutions like Intel SGX that implement *secure memory* [2]. Secure memory is defined by a protocol that makes use of a Bonsai Merkle Tree (BMT) [3]. This is a tree of hashes that is built on top of encryption-counters (to implement counter-mode encryption), and is coupled with data message authentication codes (MACs), which are secure hashes of data. To authenticate data coming from outside the trusted boundary (the chip), the data block's decryption counter is fetched and the counter's integrity is verified against the Bonsai MT by traversing it all the way up to the root, which is stored on-chip and thus its value is trusted. In addition, the data's integrity is verified against its previously stored MAC. However, secure memory has two fundamental limitations: (1) the memory authentication protocol requires additional work on memory fetch, which limits performance; and (2) secure memory metadata requires reserving a significant amount of in-memory space, which limits the amount of data accessible memory. While there has been significant work towards resolving (1) [4], [5], [6], [7], [8], there has been far less work towards resolving (2) [9], [10], [11].

To alleviate this problem, we propose the *Baobab Merkle Tree*. The Baobab Merkle Tree takes advantage of the observation that many counters in memory have the same value. Given this observation, we propose an alternative protocol where encryption counter values are memoized in an on-chip table and the indices into the memoization table become the leaves of the integrity tree. As a result, the integrity tree memory overhead is shrunk to 2–4X less compared to the BMT, as the index size is 2–4X smaller than the counters. Furthermore, the Baobab Merkle Tree increases the likelihood of finding a metadata value in an on-chip metadata cache because a Baobab Merkle Tree node protects more data than its BMT equivalent.

In this letter, we present the following contributions:

1) We propose the Baobab Merkle Tree, which memoizes encryption counters in an on-chip table, decreasing the spatial overhead of the integrity tree by $2-4X$.
2) We define a technique to memoize encryption counters on-chip.
3) We evaluate the Baobab Merkle Tree in gem5 [12], and discuss the trade-offs of its design.

II. Background

A. Threat Model

We assume a well understood threat model where an attacker has physical access to the device. The attacker can snoop and/or modify data while it is in transport and stored in memory. We assume the processor chip is within the trusted computing base and data on-chip cannot be tampered. Secure memory ensures that the values that may be corrupted in memory do not cross the trusted boundary (on-chip components are trusted). As such, defending against software vulnerabilities, side channels, and denial of service attacks are out of scope.

B. Secure Memory

Data is encrypted with counter-mode encryption [3], where each data has a unique counter value that provides spatially and temporally unique encryption keys. Data integrity is preserved by storing a keyed hash of that data and counter in memory (i.e., MAC). The MAC alone is not sufficient to protect the system against replay attacks, where the attacker replaces the data, MAC, and encryption counter with a stale value [11].

The Bonsai Merkle Tree (BMT) protects against replay attacks. The BMT is a tree of hashes in which the root of the tree is stored on-chip (i.e., is trusted) and it is built on top of

Manuscript received 4 October 2023; revised 22 December 2023; accepted 28 January 2024. Date of publication 31 January 2024; date of current version 26 February 2024. *(Corresponding author: Samuel Thomas.)*

Samuel Thomas is with the Department of Computer Science, Brown University, Providence, RI 02912 USA (e-mail: samuel_thomas@brown.edu).

Kidus Workneh, Ange-Thierry Ishimwe, Phaedra Curlin, Joseph Izraelevitz, and Tamara Lehman are with the Electrical, Computer and Energy Engineering Department, University of Colorado Boulder, Boulder, CO 80309-0401 USA (e-mail: kidus.workneh@colorado.edu; ange-thierry.ishimwe@colorado.edu; phaedra.curlin@colorado.edu; Joseph.Izraelevitz@colorado.edu; tamara.lehman@colorado.edu).

Zack McKevitt is with the Computer Science Department, University of Colorado Boulder, Boulder, CO 80309-0401 USA (e-mail: zack.mckevitt@colorado.edu).

R. Iris Bahar is with the Department of Computer Science, Colorado School of Mines, Golden, CO 80401-1887 USA (e-mail: iris_bahar@brown.edu).

Digital Object Identifier 10.1109/LCA.2024.3360709

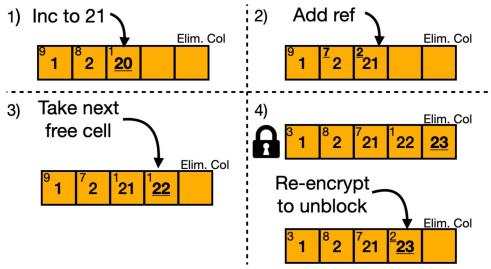

Fig. 1. Incrementing counters using the elimination column.

the encryption counters. To verify a counter with the tree, its hash is computed and compared against the stored value in the tree, traversing the tree until a trusted value is reached. On-chip metadata caches can be used to further optimize the BMT. By caching recently accessed nodes on-chip, the authentication process is shortened as it can stop as soon as a node is found in the metadata cache — values in the cache reside on-chip within the trusted boundary.

C. Secure Memory Optimizations

The notion of using memoization for accelerating the secure memory protocol is not immediately novel. Recent work [13] makes a similar observation — many counters have similar values. However, prior work is concerned with redundant AES computation on these addresses from a performance rather than redundant storage perspective. Indeed, much of the attention in the state-of-the-art has been focused on reducing the performance overheads of secure memory [4], [5], [6], [7], [8]. On the spatial optimization side, prior work implements variable arities in BMTs to reduce the overall size of the integrity tree itself [9]. This approach suffers from performance degradation. In contrast, the Baobab Merkle Tree does not impact performance. Work in Synergy [14] describes a system by which the performance to access data MACs is improved by storing these values in the ECC chips. This design eliminates an additional memory access to fetch the data MAC from memory—it is fetched at the same time as the data itself. An alternative approach is to dynamically adjust the arity of the tree on demand as it is done in Morphable Counters [10]. However, the key drawback of this approach is the frequent counter overflows and subsequent re-encryption that counteracts the spatial overhead savings with performance overhead.

III. Design

The Baobab Merkle Tree is a modification of the traditional BMT design that adds a single layer of indirection. The tree, instead of protecting each data block's counter, now protects the block's associated *index* into a *memoized counter table*. The table, containing all counter values, is stored within the on-chip memory controller. The table is divided into rows (i.e., *entries*), and each row contains a group of encryption counters (i.e., *cells*). Each data block is assigned to a fixed memoization table row, and its associated index (from the tree) indicates the column of its current counter value. The total number of cells in the memoization table is significantly smaller than the number of data blocks—the indices allow blocks to share counter values.

A. The Memoization Table

The memoization table is a fixed size buffer stored on-chip. This buffer is composed of r memoization table *entries*, and each entry has c cells. The data stored in each cell reflects a counter value that can be used for counter-mode encryption.

To reduce the likelihood of overflow and maximize utilization of space, each counter in the memoization table occupies $(64 - n)$ bits, which essentially resembles the traditional major counter in the split-counter design. When incrementing a counter value (described in Section III-C), the Baobab system needs to consider the number of blocks currently using said counter value. Thus, the proposed design includes a reference counter to track the number of blocks actively using the encryption counter value. Only the $64 - n$ bit counter is used for encryption/decryption, not the reference counter.

The remaining n bits of the column values are used to keep track of the number of blocks currently using the counter. These bits represent a "sticky counter" [15], commonly used for reference counting. For example, suppose we assume a 60 bit encryption counter and 4 reference counter bits. The 4 bits are incremented every time a new block uses the counter value and it is decremented when a block changes to a new counter value. When the 4 bits reach their maximum value of 15 (i.e., `0xf`) the reference counter reaches the "non-decrement state." and can only be reset by finding all blocks pointing to it and re-encrypting them with a new counter value.

B. Baobab Merkle Tree

The Baobab Merkle Tree is a tree of indices rather than a tree of counters. The leaves of the Baobab Merkle Tree are composed of n indices and each value is composed of $log_2 c$ bits, where c is the number of cells per memoization table entry. The physical address of the data determines where the cell index is stored, which is similar to how encryption counters are found in the traditional BMT design. Once the leaf node storing the index is accessed, the value stored determines the cell in the memoization table entry with the counter to be used for en/decryption.

C. Incrementing Counters

Incrementing a counter in the memoization table depends on its reference count and the state of the other counters within its entry. In particular, there are four types of increment scenarios in the memoization table: (1) in-place increment (2) next-cell increment, (3) free-cell increment, and (4) blocking increment. We use Fig. 1 to demonstrate each case.

In-place increment occurs when the block that requires incrementing the counter is the only block using that cell (Fig. 1, scenario 1). If the current cell holds the largest counter value in the entry or if its counter value is at least two less than the next highest counter value (to avoid duplication of counter values), then it is safe to increment the current counter value in the column. The corresponding index in the Baobab Merkle Tree does not need to change. As such, no secure memory metadata access to main memory is required because leaves in the Baobab Merkle Tree refer to indices, which in this case do not change. This is a performance savings versus baseline BMT implementations.

Next-cell increment (scenario 2) occurs when the data is mapped to a cell with a reference counter greater than one and where there is another cell in the entry with a higher encryption

counter. It also occurs when a cell has a reference counter of one and another cell in the entry has an encryption counter one more than the current encryption counter to avoid duplication of counter values. In this case, the data block needs to now use the index of the next greatest encryption counter in the row. As such, this new index is stored in the Baobab Merkle Tree, whose state has changed requiring an update to the tree. In terms of memory operations, this case exhibits similar behavior to a standard write in a BMT.

Free-cell increment (scenario 3) occurs when data uses the cell with the highest encryption counter which is not held exclusively, but another cell in the entry has a reference count of zero (i.e., a free cell). In this case, the increment uses the free cell, filling it with the value of the incremented prior encryption counter.

Blocking increment (scenario 4) occurs when the data uses the cell with the highest encryption counter with a reference counter greater than one, and the entry has no free cells available to reuse. For this case, the system reserves the last column of the memoization table entry as the "elimination column." Suppose, after some time, the entry takes the state of the upper row in Fig. 1 scenario 4. In order to increment from 22 (i.e., next cell increment), the elimination column is filled. This locks further authentications to the entry to avoid conflicts. Then, in the lower row, unblocking is achieved by scanning for the least referenced cell in the entry and re-encrypting those data with the new encryption counter value created in the elimination column (i.e., re-encrypted with 23). The encryption counter from the elimination column then replaces the cell with the fewest references, and that data is re-encrypted with the new counter value. To find which data need to be re-encrypted, we need to perform a *reverse mapping* from counters to data to check which data points to that column and needs to be re-encrypted. To ensure that there is adequate hardware, while re-encryption is happening we block all authentications that require this memoization row.

D. Assigning Blocks to Memoization Entries

The assignment of data to memoization table entries is an important feature of the Baobab Merkle Tree. To improve effectiveness, assignment of data to an entry works from a heuristic to increase the likelihood of in-place increment and decrease the likelihood of needing a blocking increment.

We work from the observation that, like virtual memory, physical memory exhibits spatial locality (especially within a page). As such, contiguous data blocks (64 bytes) within a page should be mapped to different memoization table entries. By doing so, the frequently used data within a page will have its counters increase monotonically in-place in different memoization table entries. If no physical locality is observed, blocks will need to increment counters at similar but slightly different rates, which will occupy more cells per entry. We "stripe" the memory address in their mapping to memoization table entry, as per Fig. 2.

E. Security Implications

In order to uphold secure memory semantics, Bonsai MTs protect the integrity of encryption counters and use data MACs to ensure that data has not been corrupted [11]. The intuition is that only the untampered encryption counter can produce the decryption key that decrypts the data to plaintext that matches the MAC. In the Baobab Merkle Tree, counters cannot be tampered as they are stored on-chip. Any attempts to tamper or replay the pointer will be detected by the integrity tree in the exact same way that the BMT would detect tampering or replaying of encryption counters in memory.

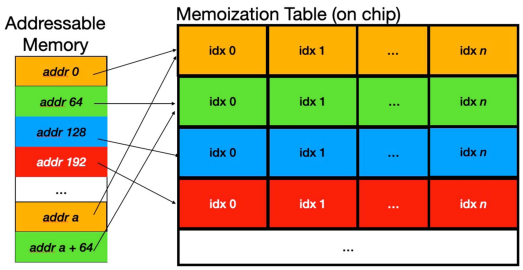

Fig. 2. Memory assignment from address to memoization table row.

TABLE I
DESCRIPTION OF THE SPATIAL TRADE-OFFS IN THE BAOBAB MERKLE TREE FOR VARYING MEMORY SIZES

	Blks/Row	Baobab MT	Bonsai MT
256G	16M	**2.5GB**	5GB
1TB	67M	**9.5GB**	19GB
8TB	536M	**78.5GB**	157GB

IV. EVALUATION

A. Methodology

We implement the Baobab Merkle Tree as an extension to gem5 [12], a cycle-accurate full system simulator. We configure a four-core simulation where each core has private L1 and L2 caches, with a shared 8MB L3 cache. The integrity tree is 8-ary, and the "leaf" arity is n-ary (configuration dependent, but either 128-ary or 256-ary, described below). We use a 32 kB metadata cache and a 224 kB memoization table. Each cell in the table is 8-bytes, with 58 bits belonging to the encryption counter and 6 bits acting as the sticky reference counter. We run two baseline approaches, one with a comparable metadata cache size to the Baobab Merkle Tree (i.e., 32 kB metadata cache) and one with a comparable on-chip resource size (i.e., 256 kB metadata cache). We use SimPoint to determine the region of interest in each benchmark, and run 500 million instructions from this region of interest. In order to avoid inaccuracies in modeling due to cold-boot, we prefill the memoization table state. The prefilled contents are collected from memory traces of each of the SPEC 2017 CPU benchmarks [16] run back-to-back while modeling what the table state would be offline from the simulation. We then run our Baobab Merkle Tree implementation using the SPEC 2017 CPU benchmarks and the Belgian street network workloads from the GAP benchmark suite [17].

B. Spatial Overhead

The spatial overhead of Merkle Trees in secure memory scales proportionally to the overall memory size. Table I shows the amount of reserved memory space required to store the integrity tree across configurations, showing both a *Baobab* and traditional *Bonsai* Merkle Tree. The fact that we can protect and authenticate twice as much data per leaf in the Baobab Merkle Tree versus the Bonsai means that the Baobab Merkle Tree *requires half as much space* in memory as the Bonsai MT.

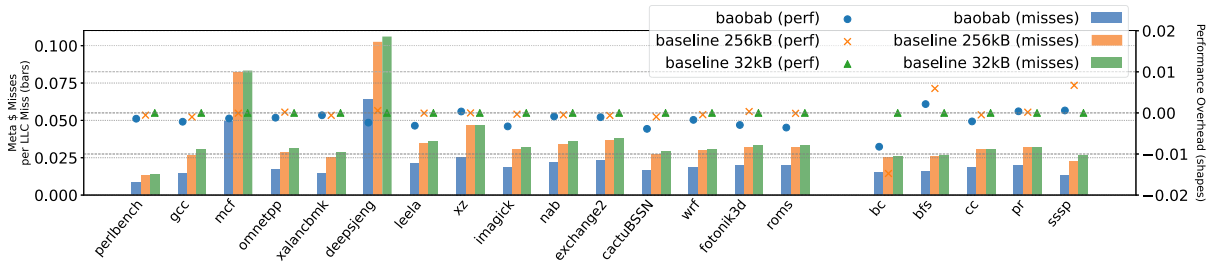

Fig. 3. Evaluation across SPEC CPU 2017 and GAP benchmark suites. (Shapes) Execution overhead of the benchmark normalized to baseline secure memory protocol with 32 kB metadata cache. (Bars) Metadata cache misses per LLC miss. In both metrics, lower is better.

The Baobab Merkle Tree size strictly depends on the number of cells within a memoization table entry. If, for example, we store 4 cells per entry, then only 2 bits are required to track the index into the entry, and thus the Baobab Merkle Tree has a spatial reduction of $4X$ rather than $2X$ (256-ary versus 128-ary leaf level). However, we opted for 16 cells per entry in our approach in order to limit the number of blocking cases, requiring 4 bits to index into the memoization entry. Blocking cases can be done in parallel with accesses to different memoization table entries, so they do not impact performance, but they should still be avoided as much as possible to reduce the bandwidth requirement to service these requests.

C. Runtime Evaluation

Fig. 3 (shapes) shows that, on average, the Baobab Merkle Tree implementation does not impact performance; it has an average performance benefit of less than two percent. That is, any differences in performance cannot be attributed to anything other than noise. As per [13], the latency to update a memoization table entry is 2ns, which is negligible relative to the memory access latency. For this reason, the overhead due to indirection incurred by the Baobab Merkle Tree is similarly negligible. While there is some additional information being tracked in the memoized data itself (i.e., the sticky reference counters), updating these values can be done at the same cycle and do not incur additional execution overheads. Furthermore, we find that the metadata cache hit rates are very high in the baseline approaches. Given these factors, the Baobab Merkle Tree has no significant overhead relative to the baseline secure memory model.

The Baobab Merkle Tree has a significant reduction in metadata cache misses relative to the Bonsai MT baseline, even though more on-chip space is used by the metadata cache. Fig. 3 (bars) shows the number of overall metadata cache misses per last-level cache miss, comparing Baobab against the baseline secure memory systems with different metadata cache sizes. In every case, Baobab makes better use of the metadata cache capacity, resulting in a reduction in metadata cache misses.

V. Conclusion

In this letter we present the Baobab Merkle Tree. We show that, because indices require fewer bits than the counters themselves, the Baobab Merkle Tree reduces the spatial overhead of the integrity tree by 2-4X. Furthermore, by making data more compact, the Baobab Merkle Tree reduces metadata cache misses, which can be a promising approach for metadata cache dependent secure memory protocols. The Baobab Merkle Tree is a promising direction for future optimizations in both performance and spatial overheads of secure memory.

References

[1] Y. Kim et al., "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Comput. Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.
[2] F. McKeen et al., "Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave," in *Proc. Hardware Architectural Support Secur. Privacy*, 2016, pp. 1–9.
[3] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS-and performance-friendly," in *Proc. IEEE/ACM 40th Ann. Int. Symp. Microarchitecture*, 2007, pp. 183–196.
[4] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in *Proc. Int. Symp. Comput. Architecture*, 2011, pp. 177–188.
[5] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-NVM: Persistency for integrity-protected and encrypted non-volatile memories," in *Proc. Int. Symp. Comput. Architecture*, 2019, pp. 104–115.
[6] F. Hou, H. He, N. Xiao, F. Liu, and G. Zhong, "Efficient encryption-authentication of shared bus-memory in SMP system," in *Proc. Int. Conf. Comput. Inf. Technol.*, 2010, pp. 871–876.
[7] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe speculation for secure memory," in *Proc. Int. Symp. Microarchitecture*, 2016, pp. 1–13.
[8] P. Zuo, Y. Hua, and Y. Xie, "SuperMem: Enabling application-transparent secure persistent memory with low overheads," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 479–492.
[9] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 665–678.
[10] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proc. IEEE/ACM 51st Annu. Int. Symp. Microarchitecture*, 2018, pp. 416–427.
[11] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *ACM SIGARCH Comput. Architecture News*, vol. 34, no. 2, pp. 179–190, 2006.
[12] J. Lowe-Power et al., "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.
[13] X. Wang, D. Talapkaliyev, M. Hicks, and X. Jian, "Self-reinforcing memoization for cryptography calculations in secure memory systems," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture*, 2022, pp. 678–692.
[14] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *Proc. Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 454–465.
[15] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, "Taking off the gloves with reference counting immix," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 93–110, 2013.
[16] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 41–42.
[17] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015, *arXiv:1508.03619*.