Verifying Data Integrity with Few Queries to Untrusted Memory

Nachiketh R. Potlapally,
Dept. of Electrical Engineering,
Princeton University, Princeton, NJ 08544
npotlapa@princeton.edu

Abstract - *We present a novel technique for verifying the integrity of data stored in an untrusted memory with a small number of memory accesses. Memory integrity verification, which enables detection of tampering of data stored in untrusted memory, is an essential requirement of secure processors that provide private and tamper-proof computation. Limited on-chip storage in a secure processor makes it necessary for it to store data (including program code) in an untrusted external memory where it is easily susceptible to adversarial tampering. Thus, to ensure validity of computation, it is extremely important to have techniques that can verify integrity of data stored in untrusted memory. Existing memory integrity verification techniques, like Merkle trees, impose very high communication overhead,* i.e., *large number of queries from processor to memory, in order to perform data integrity verification. Given that memory latency is very high compared to execution speed of the processor, this imposes a significant running time penalty for applications executing on the processor. Our proposed technique, which is based on Chinese remaindering theorem, performs integrity verification with low communication overhead while incurring a modest increase in on-chip storage requirement. We present the details of the proposed technique and provide corresponding proofs of security and correctness. Our technique can not only be used by itself, but can also be incorporated into existing techniques, like Merkle trees, to reduce their communication overhead.*

1 Introduction

Memory integrity verification refers to the process of detecting any unauthorized tampering of data stored in external memory. This verification is an important component of secure and trusted processing architectures. Most of the proposed architectures for secure and trusted computing comprise a tamper-proof processor enclosing on-chip memory for storing cryptographic keys, and highly sensitive code and data (*e.g.*, that of the trusted kernel), and special-purpose cryptographic hardware for carrying out efficient cryptographic computations [1, 2, 3]. However, code and data of trusted applications that cannot be stored in the limited trusted on-chip memory is sent to the external memory which is outside the secure perimeter imposed by the tamper-resistant processor casing. Thus, it is imperative to have a memory integrity verification scheme that can detect any unauthorized tampering of data between the time it is written into and read back from the external memory by the secure processor.

The external memory, containing program code and data, is assumed to be in total control of an adversary who can alter values in any memory location in any manner. Memory integrity verification is done through a program running on the secure processor and ensures the integrity of code and data requested by any program in the course of its normal execution. In order to verify the integrity of external memory values requested by a program, the verification program makes queries to the external memory for additional data. The number of additional queries made by the verification program is referred to as its *communication complexity*. The verification program has dedicated on-chip private memory for storing values used for checking the integrity of untrusted external memory. The size of on-chip private memory used by the verification program is referred to as its *space complexity*. The two main requirements of any effective memory integrity verification program are as follows:

- It should detect any form of adversarial corruption of values stored in external memory.
- Since integrity verification is done on code and data requested by an executing program, it is important that verification should not impose a significant execution time overhead.

For the sake of efficient performance, the communication complexity of an integrity verification program should be as low as possible (since memory accesses are quite time-consuming). However, reducing communication complexity comes at the cost of increased space complexity. This observation can be formally described as follows. Given an external memory of size n blocks that is partitioned into sets of size m blocks each, cryptographic hashing is applied to all the $\lceil \frac{n}{m} \rceil$ sets, and the resulting $\lceil \frac{n}{m} \rceil$ hash values are stored in the on-chip private memory of the verification program. When a memory block belonging to set i is read by an executing program, in order to verify the integrity of the block read, the remaining $(m-1)$ blocks in set i have to be read so that the hash value of set i can be computed. This computed hash value is compared with the pre-computed hash value (of set i) stored safely on the processor, and a match implies that the set (and, thereby the block) was not tampered with. The space complexity of this scheme is $O(\frac{n}{m})$, and the communication complexity is $O(m)$. They are inversely related. An ingenious solution with $O(1)$ space complexity is offered by Merkle tree [4] for applications which require low space overhead. A Merkle tree is a tree data structure in which the memory sets comprise leaf nodes of the tree, and every internal node holds the hash of the concatenation of contents of all its children. Only the hash in the root node is stored securely on the processor. In order to verify the integrity of a block read from partition i, the contents of every node and its siblings lying on the path from the leaf node (holding partition i) to the root node have to be read, in order to compute the root hash. The computed root hash is compared with the pre-computed root hash value stored on the processor. The communication complexity of Merkle tree-based scheme is $O(m + log_2(\frac{n}{m}))$ which can be quite significant for large values of n. Moreover, reducing m decreases one factor of Merkle tree communication complexity, but increases the other. Thus, there is a limit to performance gained by reducing m. However, with continuing trend towards rapidly widening gap between processor and memory speeds, high communication complexity of a memory integrity scheme can significantly increase the latency of memory accesses. Due to increasing processor die sizes, we can afford to tolerate space complexity. However, it is highly desirable that the communication complexity of a memory integrity verification scheme be low.

1.1 Contribution of the paper

We propose a memory integrity verification technique that significantly reduces the communication complexity without incurring a corresponding increase in space complexity (as indicated by the inverse relationship between communication and space complexities). Our technique achieves its stated goal by employing the Chinese remainder theorem (CRT) [5].

As before, let us assume that n blocks of main memory are divided into $\lceil \frac{n}{m} \rceil$ sets each having m blocks. In the proposed method, the hash of a set is calculated in such a manner that in order to verify the integrity of a block belonging to it, we only need a small subset of blocks (say, k) rather than all the m blocks in the set. Here, each set i, $1 \leq i \leq \lceil \frac{n}{m} \rceil$, is further sub-divided into $\lceil \frac{m}{k} \rceil$ partitions of size k blocks each. The hash value of set i, H^i, is computed as a function of the hash values of the contents of the $\lceil \frac{m}{k} \rceil$ partitions, H^i_j, $1 \leq j \leq \lceil \frac{m}{k} \rceil$, in such a manner that there is an invariant relation between H^i and each H^i_j. This invariant relation is established by CRT. When a block is read from partition j of set i, its integrity is verified as follows:

- Read the remaining $(k-1)$ blocks from partition j.
- Compute the hash value, H^i_j, of partition j.
- Check if the partition hash value H^i_j satisfies the invariant relation with the hash value of the set i, H^i (which is pre-computed and stored in the private on-chip memory).
- If the property is satisfied then it implies that the block was not tampered with.

The salient feature of the proposed scheme is that the communication complexity is independent of memory size (n) and set size (m), and is dependent only on a constant k which is independent of n and m. The scheme achieves this reduction in communication complexity with $O(\frac{n}{m})$ space complexity. Table 1 summarizes the complexities of the three memory integrity verification schemes described above: store hashes (of the sets), Merkle tree and the proposed scheme.

Table 1: Comparison of space and communication complexity

Scheme	Communication complexity	Space complexity
Store hashes	$O(m)$	$O(\frac{n}{m})$
Merkle tree	$O(m + log_2(\frac{n}{m}))$	$O(1)$
Proposed	$O(k)$	$O(\frac{n}{m})$

1.2 Related work

Most of the previous work in memory integrity verification has relied on the use of Merkle tree (or hash tree) [4] which was originally proposed as a way of authenticating data between untrusted entities using a minimum amount of memory. Blum *et al.* [6] were the first to propose the use of hash tree for verifying the correctness of data stored in large untrusted memories and provided theoretical proofs of security. Subsequently, hash tree-based memory integrity verification has been used in various scenarios: for building secure databases using untrusted storage [7], managing the persistent state in digital rights management systems [8], verifying data structures, like stacks and queues, stored in untrusted memory by memory-constrained embedded systems such as smartcards [9], and certifying program execution on a trusted processor [10]. Though the space complexity of hash tree-based schemes is small ($O(1)$ since only the root hash is stored securely), the communication complexity is $O(log_m N)$ where m is number of children per node and N is the number of leaf nodes in the hash tree (also, the number of blocks in the memory if there is a block per leaf node). Wide usage of hash trees prompted research aimed at mitigating the performance drawback of hash-tree based memory integrity verification schemes. Gassend *et al.* [11] proposed cache-centric architectural enhancements directed at reducing the latency of tree-based integrity verification. Williams and Sirer [12] use analytical modeling to determine the size of the leaf node, *i.e.*, the number of memory blocks per leaf, that will result in an optimal performance. Both these works [11, 12] improve performance by fine-tuning implementation-specific parameters, and do not propose any conceptual changes. Any architectural recommendations which involve increasing the size of on-chip storage will also improve the performance of the proposed scheme. On the theoretical side, there are some works which propose hash functions that enable fast incremental hash recomputation on modified data which find application in memory integrity checking [13, 14, 15]. However, these works are broad in scope, and they do not specifically target reducing communication complexity of memory integrity checking. Our proposal significantly reduces communication complexity compared to the existing techniques by proposing a conceptual advancement based on a number-theoretical construct. In addition, the space overhead for realizing this reduction in communication complexity is modest.

The rest of the paper is organized as follows. Section 2 provides formal definitions of the routines used by the proposed technique, and their corresponding security definitions. Section 3 gives a brief description of some cryptographic tools and the manner in which they can be used to obtain concrete implementations of the abstract formulations given in the previous section. It also enumerates the security proof of the implementation. Section 4 presents results comparing communication complexity of the proposed scheme with existing methods for a typical system configuration. Section 5 concludes with some observations and directions for future work.

2 Definitions

In this section, we begin by giving a formal description of a memory integrity verifier (or checker), and follow it up with a formal definition of the sub-routines used by our proposed scheme. We end the section by formulating the security requirements to be satisfied by our technique in order for it to be considered secure against adversarial attacks.

2.1 Basic model and definitions

A memory integrity checker is a program that detects any corruption of data stored in a large untrusted memory. The checker runs within a secure and tamper-proof perimeter while an adversary is assumed to have total control over the external memory. As part of its operation, the checker maintains a *state* of the

external memory in a much smaller private memory, and uses this state value to detect any corruption of values stored in the untrusted memory. The checker program handles the loads and stores issued by the processor in the following manner:

- STORE(x,y): The checker updates its *state* variable to reflect that value y was written into memory location x.
- LOAD(x): Using its *state* variable, the checker verifies if value z read from memory location x is the same as the value last written into it. If the checker determines with sufficiently high confidence that value z is uncorrupted, then it passes it to the processor with the signal *"accept"*. Else, it sends a *"reject"* signal to the processor.

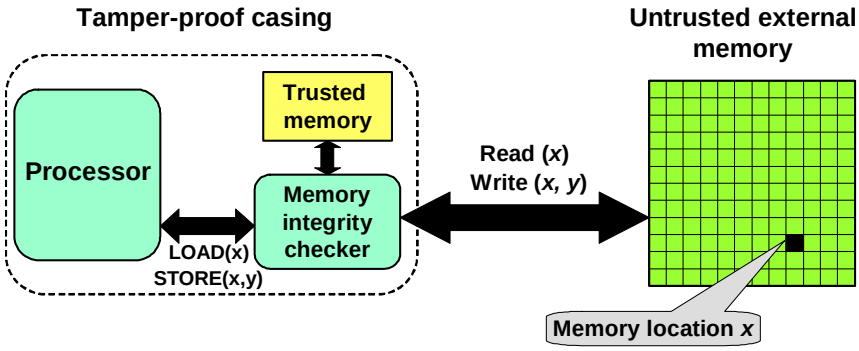

Figure 1: Memory integrity checker

Figure 1 illustrates the working of a memory integrity checking program. The size of the trusted memory used by the checker to store the state variable determines its space complexity. The functioning of a memory checker can be formally specified as follows:

Definition 1: *A memory integrity checker is a probabilistic program C that maintains a private value state which is updated on writes to an untrusted memory M. If y is a value written to location x in M, and z the value read subsequently from the same location, then the following conditions hold:*

- *Completeness:* $Pr[\forall x, y, z\text{: } y = z \ni C(state, x, z) = accept] \geq (1 - \epsilon)$.
- *Soundness:* $Pr[\forall x, y, z\text{: } y \neq z \ni C(state, x, z) = accept] < \epsilon$

Completeness implies that valid inputs are accepted by C with overwhelming probability, and soundness implies that wrong inputs are accepted by C with negligible probability. A challenge is to compute the state in such a way that its size is reasonable while being able to detect corruption of values in memory with a high probability, *i.e.*, make ϵ as small as possible.

The proposed memory integrity checking technique comprises two routines: `UPDATE()` and `CHECK()`. The `UPDATE()` routine constructs and updates the state of the checker to reflect new additions and modifications to existing values in untrusted memory, and `CHECK()` utilizes the state of the checker to determine whether the values read in from external memory are corrupted. Given the functionality of the two routines, we can see that they have asymmetric constraints: `UPDATE()` can afford to incur a time penalty in order to construct and maintain the checker state such that it accurately encapsulates the characteristics of legitimate changes made to values in the memory while `CHECK()` has to have low latency as it is called on every load instruction issued by a program executing on the processor (assuming loads are much more frequent than stores which is usually the case). We now proceed to formally define both the routines.

Definition 2: *Let $M^i = \{M_1^i, M_2^i, \ldots, M_n^i\}$ be a set of n equal-sized partitions.* `UPDATE()` *takes set M^i as input, and outputs a pair of values, $HASH_{M^i}$ and $STATE_{M^i}$,* i.e., *$(HASH_{M^i}, STATE_{M^i}) \leftarrow$* `UPDATE`*$(M^i)$. $HASH_{M^i}$ is the public hash value of set M^i while $STATE_{M^i}$ is a private state value (stored securely in on-chip memory) used by the checking procedure to detect any corruption in the partitions of set M^i. Then, the* `CHECK()` *procedure should satisfy the following properties:*

- *Completeness:* $Pr[\forall M^i,\ (HASH_{M^i}, STATE_{M^i}) \leftarrow$ `UPDATE(`M^i`)`, $\forall M^i_j \in M^i \ni$ `CHECK(`$HASH_{M^i}, STATE_{M^i}, j, M^i_j$`)`$= 1] \geq (1-\epsilon)$

- *Soundness:* $Pr[\forall M^i,\ (HASH_{M^i}, STATE_{M^i}) \leftarrow$ `UPDATE(`M^i`)`, $\forall M^i_{j'} \notin M^i \ni$ `CHECK(`$HASH_{M^i}, STATE_{M^i}, j', M^i_{j'}$`)`$= 1] < \epsilon$

A subtler form of soundness is the property of *collision resistance* which says that two different sets of n partitions, M^i and M^j, should not be mapped by `UPDATE()` to the same hash value, *i.e.*, $Pr[\forall\, M^i, M^j,\ M^i \neq M^j,\ (HASH_{M^i}, STATE_{M^i}) \leftarrow UPDATE(M^i),\ (HASH_{M^j}, STATE_{M^j}) \leftarrow UPDATE(M^j) \ni HASH_{M^i} = HASH_{M^j}]$ is negligible. Collision resistance is related to soundness because if $M^i \neq M^j$ collide, then the hash value and state generated for M^i can be used successfully to fool the checking routine for input M^j.

2.2 Security definitions

We make the assumption that routines `UPDATE()` and `CHECK()` are executed securely, and an adversary cannot interfere with their operation. However, it is possible that an adversary can gather enough information by observing the inputs and outputs of the memory integrity checker program to be able to successfully simulate its operation. Such an adversary is known as a *passive adversary*. In contrast, an *active adversary* is one who is able to provide inputs of his choice to the checker program and observe the corresponding outputs with the aim of being able to successfully simulate the memory checker program. The two routines, `UPDATE()` and `CHECK()`, are considered to be secure against an adversary (passive or active) if he is successful in simulating the memory checker program with only a negligible probability.

Definition 3: *A memory integrity checker C comprising routines* `UPDATE()` *and* `CHECK()` *is said to be (p,k)-secure against a passive attack by a computationally bounded adversary A if*

$$Pr\left[A\left(T^k\left(\begin{array}{c}(X_{M^i}, Y_{M^i}) \leftarrow O^{UPDATE}(M^i),\\ O^{CHECK(Y_{M^i})}(X_{M^i}, j, M^i_j)\end{array}\right)\right) = (X', j', M^i_{j'}) \right] < p$$
$$\ni\ O^{CHECK(Y_{M^i})}(X', j, M^i_{j'}) = 1$$

where $M^i = \{M^i_1, M^i_2, \ldots, M^i_n\}$ is a set of n partitions, $O^{UPDATE}(M^i)$ indicates A's black-box access to the `UPDATE()` *routine with M^i provided as input to generate the corresponding hash value and state (X_{M^i}, Y_{M^i}, respectively), and $O^{CHECK(Y_{M^i})}(X_{M^i}, j, M^i_j)$ represents A's black-box access to the* `CHECK()` *routine such that the secret parameter Y_{M^i} is not revealed, however, it can observe the result of running the routine on public inputs (X_{M^i}, j, M^i_j). $T^k(.,.)$ is a random variable comprising k samples of independent executions of* `UPDATE()` *and* `CHECK()` *on uniformly generated values of set M^i and index j (M^i_j is dependent on M^i and j), and $A(T^k(.,.)) = (X', j', M^i_{j'})$ indicates A's ability to generate a tuple $(X', j', M^i_{j'})$ (different from values observed in the k samples) after observing the k executions such that the tuple successfully fools the* `CHECK()` *routine, and p is the upper bound on the probability of A generating a successful tuple.*

A formal definition of security against an active adversary can be similarly defined wherein adversary A has the additional ability to generate his choice of set M^i and observe the execution of `UPDATE()` and `CHECK()` on these inputs. The number of samples k is assumed to be polynomially bounded. Also, in each case, the adversary is assumed to be computationally bounded, *i.e.*, is a probabilistic polynomial time algorithm. Security against an active adversary is a much stronger notion than security against a passive adversary.

3 CRT-based memory integrity verification

In this section, we present concrete details regarding the working of the proposed memory integrity verification technique. We begin by enumerating the mathematical tools which form the underpinning of

our method (Section 3.1), followed by a high-level description of the proposed method (Section 3.2), and a discussion on security intuition (Section 3.3). Next, we give implementation details of two routines, `UPDATE()` and `CHECK()`, which are built using the mathematical tools previously described and form the building blocks of the proposed method (Section 3.4). Then we provide the description of the proposed memory integrity verification method which uses the two routines described (Section 3.5). We conclude the section with an analysis and discussion of the security properties of the proposed method (Section 3.6).

3.1 Mathematical tools

The proposed technique employs four mathematical tools: cryptographic hashing, fingerprinting function, universal hashing and CRT [5]. We enumerate the essential features of these concepts and their functionality in the proposed scheme.

- *Cryptographic hashing*: A cryptographic hash is a collection of functions parameterized on a key K that maps arbitrary-length strings to fixed-length strings, *i.e.*, $\{H_K\colon \{0,1\}^* \rightarrow \{0,1\}^a\}$, in a way such that the following three properties are satisfied,
  - *Pre-image resistance*: It should be computationally infeasible to find an input value which hashes to the specified output value (also called non-invertibility).
  - *Second pre-image resistance*: It should be computationally infeasible to find a second input value that hashes to the same output value as the specified input value.
  - *Collision resistance*: It should be computationally infeasible to find two input values that hash to the same output value, *i.e.*, $Pr[\forall\, m, m' \in \{0,1\}^*, m \neq m' \ni H_K(m) = H_K(m')]$ is negligible.

  Construction of most of the standard cryptographic hash functions is based on the Merkle-Damgard method [13]. In our method, a hash function is used to map contents of arbitrary-sized partitions of main memory blocks to fixed-length strings. Any of the standard cryptographic hash functions (SHA-1, MD5, RIPEMD-160, *etc.*) would suffice for our scheme.

- *Fingerprinting function*: A fingerprinting function is a collection of functions parameterized on a key P which produces short tags (*fingerprints*) for a collection of larger objects, *i.e.*, $\{F_P : \Omega \rightarrow \{0,1\}^b\}$ where Ω is the set of all possible objects of interest and b is the length of the fingerprint [16]. For any set $S \subset \Omega$ of n distinct objects, and function f chosen randomly from the family of fingerprint functions, $f(x) \neq f(y)$ implies $x \neq y$ for any $x, y \in S$. However, in adversarial situations, fingerprinting functions are not as collision-resistant as cryptographic hashes. We use fingerprinting in our method to map the cryptographic hashes to smaller-length strings with the aim of reducing the size of output generated by subsequent CRT computation. We use the Rabin fingerprinting technique which maps an m-bit binary input value $A = (a_1, a_2, \ldots, a_m)$, represented as a polynomial of degree $m-1$, $A(x) = a_1x^{m-1} + a_2x^{m-2} + \ldots + a_m$, to a b-bit fingerprint $F(x) = A(x)\, mod\, P(x)$ where $P(x)$ is an irreducible polynomial of degree b [17]. The operation of division modulo a polynomial over $GF(2)$ is implemented through a simple linear feedback shift register whose feedback connections are determined by the coefficients of the dividing polynomial $P(x)$.

- *Universal hashing*: A family of functions H from domain D to range R is said to be Δ-universal if for all $x, y \in D$ with $x \neq y$ and all $\delta \in R$ (an Abelian group), $Pr_{h \in H}[h(x) - h(y) = \delta] \leq 1/|R|$. We use a Δ-universal hash function to make the selection of primes used in the CRT computation dependent on a randomly chosen subset of the cryptographic hash function. This improves the overall security of the proposed technique (explained further in the next section). In the proposed scheme, we realize strongly universal hashing through the use of the square hash [18] which is defined as $h_x(m) = \sum_{i=1}^{k}(m_i + x_i)^2 \bmod q$, where $x = (x_1, \ldots, x_k), m = (m_1, \ldots, m_k) \in Z_q^k$ and q is prime.

- *CRT*: CRT states that given n relatively prime positive integers, p_1, $p_2, \ldots,$ p_n, and integers r_1, r_2, $\ldots, r_n$ such that $r_i \in \mathcal{Z}_{p_i}$, there exists a unique integer $X \in \mathcal{Z}_{\prod_{i=1}^{n} p_i}$ such that $X \equiv r_i\, mod\, p_i$. This property of CRT, which establishes a well-defined invariant relation between the value X, and each member of the set $\{r_1, r_2, \ldots, r_n\}$, is pivotal for our method, *i.e.*, every r_i is a residue of X

reduced with modulus p_i. In the proposed method, the modular residues r_i's are fingerprints of cryptographic hashes of contents of the memory partitions in a set, and the integrity of memory partition i is verified iff its fingerprint $r_i = X \bmod p_i$ (where p_i is the prime corresponding to partition i and was used in the CRT computation of X).

3.2 Overview of the proposed technique

We provide an overview of the proposed technique by illustrating the manner in which the tools described in the previous section are used to achieve the desired objective of memory integrity verification with low communication complexity. At the same time, we do not want to impose a high space overhead for achieving this goal. Let $M^i = \{M_1^i, M_2^i, \ldots, M_n^i\}$ be a set of n equal-sized partitions such that $|M_j^i| = l_1$, $1 \leq j \leq n$ ($|x| = \lfloor(\log_2 x + 1)\rfloor$ indicates the bit-width of x). We want to generate the hash value for set M^i, and its corresponding state information which is used to verify the integrity of any partition $M_j^i \in M^i$ independently of the other partitions in M^i. To keep the space overhead low, we need to make the size of the state information as small as possible. The generation of hash value and state information corresponding to set M^i is done as follows:

1. First, a cryptographic hash function is used to map the large memory partitions into much smaller fixed-size hash values which satisfy the above-mentioned cryptographic properties. The cryptographic hash function $H_{K_i}()$, parameterized on key K_i, is defined as H: $\{0,1\}^{l_1} \times \{0,1\}^{|K_i|} \rightarrow \{0,1\}^{l_2}$, $l_1 > l_2$, which maps l_1-bit contents of the jth partition M_j^i to an l_2-bit hash value $h_j^i = H_{K_i}(M_j^i)$, $1 \leq j \leq n$.

2. In the second step, the cryptographic hashes are mapped to smaller-sized fingerprints. This is done primarily to reduce the size of the state information of set M^i which is generated by combining the n partition fingerprints using CRT. Also, the smaller-sized fingerprints reduce the complexity of the CRT computation. The Rabin fingerprinting function $F_{P_i}()$, parameterized on an l_3-degree irreducible polynomial P_i, is defined as F: $\{0,1\}^{l_2} \times \{0,1\}^{|P_i|} \rightarrow \{0,1\}^{l_3}$, where $l_2 > l_3$, which reduces an l_2-bit cryptographic hash h_j^i to an l_3-bit fingerprint $f_j^i = F_{P_i}(h_j^i)$, $1 \leq j \leq n$.

3. Finally, the partition fingerprints f_j^i ($1 \leq j \leq n$) are combined by CRT to generate the state information of set M^i. To enable CRT computation, a distinct prime is associated with each fingerprint. Let $P = \{p_1, p_2, \ldots, p_q\}$ be a set of q primes where $q > n$ and $|p_j| = (l_3+1)$ ($1 \leq j \leq q$). For set M^i, the 1-1 mapping π^i: $\{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, q\}$ is used to select n distinct primes from set P (the π^i mapping has a security implication which is explained in the next section). The $CRT_{p_{\pi^i(1)}, p_{\pi^i(2)}, \ldots, p_{\pi^i(n)}}()$ function defined as CRT: $\{0,1\}^{n \cdot l_3} \times \{0,1\}^{n \cdot (l_3+1)} \rightarrow \{0,1\}^{l_4}$ where $l_4 < n \cdot (l_3 + 1)$, generates an l_4-bit output X^i such that $X^i \equiv f_j^i \bmod p_{\pi^i(j)}$. X^i is the state information of set M^i which is stored securely in a trusted memory, and $H_{K_i}(X^i)$ is the hash value that is made public.

In a conventional scheme, where the hashes of the sets are stored, the contents of set M^i would be hashed, and the hash value (length l_2) stored securely. In order to keep the space complexity of the proposed scheme comparable, it is desirable that l_4 (size of state X^i) is within a factor of l_2 (hash value length), *i.e.*, $l_4 = \sigma \cdot l_2$, $1 \leq \sigma < 2$. Since $l_4 < n \cdot (l_3 + 1)$, we have $n \cdot (l_3 + 1) > \sigma \cdot l_2$, and $l_3 > \lfloor \frac{\sigma \cdot l_2}{n} - 1 \rfloor$. Thus, the value of l_3 (fingerprint size) is chosen so that $l_4 = \sigma \cdot l_2$ for a given σ (alternately, for given values of σ and l_3, the number of partitions n can be determined by $n > \lfloor \frac{\sigma \cdot l_2}{l_3+1} \rfloor \approx \lfloor \frac{\sigma \cdot l_2}{l_3} \rfloor$).

The verification process is much simpler than state information generation. In order to verify the integrity of a block belonging to partition $M_j^i \in M^i$, only partition M_j^i is read from memory, and its contents are hashed and fingerprinted to yield value f, *i.e.*, $f = F_{P_i}(H_{K_i}(M_j^i))$. The integrity is verified by checking whether f is equal to $X^i \bmod p_{\pi^i(j)}$ (where state X^i is stored securely in a trusted memory). Thus, we see that the communication complexity of integrity verification is equal to a constant, *i.e.*, the size of partition M_j^i, and is independent of the size of set M^i. Also, only the state information of the set

is needed for integrity verification. The hash value of the set is computed so that the proposed technique can be incorporated into memory integrity verification schemes which require such a value.

3.3 Security intuition

In this section, we explain the $\pi^i()$ mapping, and the intuition behind employing it for improving the security of the proposed scheme. A cryptographic hash function $H_{K_i}()$ has very low collision probability. However, this collision probability is increased by composing it with the fingerprint function, *i.e.*, $F_{P_i}(H_{K_i}())$, which is motivated by the need to keep the space complexity overhead of the proposed scheme low. Thus, given a partition $M_j^i \in M^i$, it might be possible to modify it to $M_j^{i'}$ such that $H_{K_i}(M^i) \neq H_{K_i}(M_j^{i'})$, but $f_j^i = f_j^{i'}$, where $f_j^i = F_{P_i}(H_{K_i}(M_j^i))$ and $f_j^{i'} = F_{P_i}(H_{K_i}(M_j^{i'}))$. In such a case, $CRT_{(p_1,p_2,\ldots,p_n)}()$ (where p_j, $1 \leq j \leq n$, is statically assigned) would give the same result X^i for sets M^i and $M^{i'}$ (where set $M^{i'}$ is set M^i with the jth partition M_j^i modified to $M_j^{i'}$). In the proposed method, the p_j's used in the CRT computation are selected by a Δ-universal hash function-based mapping $\pi^i()$ which takes as its input a *random* subset of bits from the partition hash value $H_{K_i}(M_j^i)$. We use a random subset (rather than a pre-determined one) to make it difficult for the adversary to have any influence on the output of the $\pi^i()$ mapping. Thus, if partition M_j^i is modified to $M_j^{i'}$, then there is a high probability that $\pi^i()$ will map j to different values in both the cases (thereby assigning different $p_{\pi^i(j)}$'s). Then, function $CRT_{(p_{\pi^i(1)},p_{\pi^i(2)},\ldots,p_{\pi^i(n)}}()$ will map sets M^i and $M^{i'}$ to different values, thereby reducing the adversary's chances of fooling the memory checker with malicious inputs like $M_j^{i'}$ (this probability is analyzed in a later section). For a set M^i having n partitions, the mapping $\pi^i(j)$ for partition j ($1 \leq j \leq n$) is given by $h_x(m_j) = \sum_{i=1}^{k}(m_{ji}+x_i)^2 \bmod q$, where m_j is a $k.|q|$-bit chunk chosen randomly from the partition cryptographic hash value $h_j^i = H_{K_i}(M_j^i)$, and $m_{ji}, x_i \in \mathcal{Z}_q^k$. There are two optimizations possible to make the implementation of the $\pi^i()$ mapping more efficient with respect to space and time:

- If prime q is chosen to be of the form $2^l - 1$ (for example, $l = 5$), then the mod q operation can be performed using a shift, add, compare, and, if needed, a subtract.
- A prime p_i is called a Sophie Germain prime if $(2p_i + 1)$ is also prime. There exist many such primes, and they have been used in various cryptographic protocols [19]. Thus, instead of storing q primes, we can store only $\lfloor q/2 \rfloor + 1$ Sophie Germain primes, and the remaining $\lfloor q/2 \rfloor$ primes can be generated by a simple computation $(2p_i + 1)$, where $1 \leq i \leq \lfloor q/2 \rfloor$.

3.4 `UPDATE()` and `CHECK()` implementation details

`UPDATE()` is responsible for generating the hash value and the state corresponding to a set of partitions and recomputing the hash and state information when contents of one or more of the partitions is modified. `CHECK()` is responsible for verifying whether contents of any of the partitions in the set have been illegally modified by an adversary. The routine does this with the aid of state information computed for the set by `UPDATE()`).

Figure 2 shows the pseudo-code of `UPDATE()`. This routine takes as its input the set M^i and outputs the corresponding state information X^i and hash value $H_{K_i}(X^i)$. Initially, the hash values and fingerprints of all the partitions in the set are generated (steps $1-4$). Next, the mapping $\pi^i(j)$ ($1 \leq j \leq n$) is computed (steps $5-14$). In order to do this, initially a random index is generated (step 8), and a $k.|q|$-bit chunk beginning at that random index is extracted (circularly, if needed) from the hash value h_j^i (step 9). This value is input to the strongly universal hash function $h_x()$ (step 10) to generate the mapped value, *i.e.*, $\pi^i(j)$. If this value is different from all the values generated for indices 1 to $(j-1)$ (step 11), then the prime $p_{\pi^i(j)} \in P$ is stored in set P^i (step 13), else the computation is repeated. Next, the CRT computation is used to calculate state X^i (step 15), and its hash is computed (step 16). The state is securely stored on the processor while the hash is stored in the untrusted memory. The CRT computation

```
UPDATE (Primes P = {p_1, ..., p_q}, Set M^i = {M^i_1, ..., M^i_n},
        Key K_i, Polynomial P_i)
 1 : for (j = 1; j ≤ n; j = j + 1)
 2 :  h^i_j ← H_{K_i}(M^i_j);
 3 :  f^i_j ← F_{P_i}(h^i_j);
 4 : endfor
 5 : P^i = {∅};
 6 : for (j = 1; j ≤ n; j = j + 1)
 7 :  do{
 8 :    index^i[j] ← rand();
 9 :    seed ← h^i_j[index^i[j] ··· (index^i[j] + k.|q| − 1)];
10 :    π^i(j) ← h_x(seed);
11 :  while (π^i(j) ∈ {π^i(1), π^i(2), ..., π^i(j − 1)})
12 :  end do
13 :  P^i = P^i ⋃ p_{π^i(j)} ;
14 : endfor
15 : X^i ← CRT(P^i, f^i_1, f^i_2, ..., f^i_n);
16 : Z^i ← H_{K_i}(X^i);
17 : return (Z^i, X^i);
```

Figure 2: Pseudo-code for `UPDATE()`

allows incremental updates to state X^i. If a partition M^i_j is modified to $M^{i'}_j$, then updated state $X^{i'}$ can be computed from old state (X^i), old partition (M^i_j) and updated partition ($M^{i'}_j$) rather than by repeating the entire CRT computation. Figure 3 shows the pseudo-code for `CHECK()`. This routine takes as inputs, the contents of a partition and the state corresponding to the set which the partition belongs to (along with the hash value of the state, keys, indices, and list of primes), and verifies whether the partition was tampered with since the last time it was legally modified. Initially, an optional sanity check is performed by checking whether hash of the state value equals the set hash value, else it aborts (steps $1-4$). Then the hash value and the fingerprint of the partition are calculated (steps $5-6$). Next, the *index* associated with partition j of set i is obtained from the array filled in by `UPDATE()` (step 7), and $k.|q|$ bits beginning at *index* are read (circularly, if needed) from the hash value as the *seed* (step 8). The mapping corresponding to partition j of set i, $\pi^i(j)$, is obtained by giving *seed* as input to the hash function $h_x()$ (step 9). Finally, the state information is modular-reduced with prime $p_{\pi^i(j)}$, and checked to see whether the result is equal to the fingerprint calculated (steps $10-14$). Equality guarantees non-corruption of the contents of partition M. Illegal tampering of the partition will be detected with a high probability since it will either produce a wrong fingerprint or a wrong mapping for choosing the prime associated with the partition.

We can see that `UPDATE()` is more expensive than `CHECK()`. However, since in a program execution, loads are more frequent than writes, and the `CHECK()` routine is called on every load operation, it should have the minimum possible latency to perform quick verification of data integrity. Also, the higher performance penalty of `CHECK()` can be amortized by the lower cost incurred for updating the state for subsequent modifications to the data in the set.

3.5 Memory integrity verification using `UPDATE()` and `CHECK()`

In this section, we illustrate the working of our proposed memory integrity verification scheme which uses routines `UPDATE()` and `CHECK()` as its building blocks. As mentioned in Section 2.1, the processor issues `LOAD()` and `STORE()` instructions to the memory integrity checker which, in turn uses the `CHECK()` and `UPDATE()` routines to perform the required integrity verification on the addressed data, respectively. If the integrity verification is successful, the `LOAD()` and `STORE()` routines are handled in the normal way.

```
CHECK (Primes P = {p_1, ..., p_q}, Hash Z^i, State X^i,
       Set index i, Partition M^i_j, Partition index j,
       Key K_i, Polynomial P_i)
 1 : Z' ← H_{K_i}(X^i);
 2 : if (Z^i ≠ Z')
 3 :  exit("error");
 4 : endif
 5 : r1 ← H_{K_i}(M^i_j);
 6 : r2 ← F_{P_i}(r1);
 7 : index ← index^i[j];
 8 : seed ← r1[index ··· (index + k.|q| − 1)];
 9 : π^i(j) ← h_x(seed);
10 : s ← X^i mod p_{π^i(j)};
11 : if (r2 == s)
12 :  return 1;
13 : else
14 :  return 0;
15 : endif
```

Figure 3: Pseudo-code for `CHECK()`

Let $\mathcal{M}$ be the untrusted (external) memory which is divided into m sets, $\{M^1, M^2, \ldots, M^m\}$ where each set M^i is further divided into n equal-sized partitions, $\{M^i_1, M^i_2, \ldots, M^i_n\}$. If each partition has p memory blocks, then the size of untrusted memory $\mathcal{M} = m * n * p$ blocks. A set of q distinct primes (only $\lfloor q/2 \rfloor$ Sophie Germain primes are used), $P = \{p_1, p_2, \ldots, p_q\}$, where $q > n$, is pre-computed and stored in the trusted memory. A subset of n primes, $P^i \subset P$, is obtained using the $\pi^i()$ mapping and assigned to every set M^i. To reduce the space overhead, the same hashing key (K_i) and irreducible polynomial (P_i) can be used for all sets M^i, $1 \leq i \leq m$. Thus, the space overhead incurred for each set M^i is its state information X^i.

```
LOAD (Memory block x)
 1 : Determine set (i) and its constituent partition (j) to which block x belongs
 2 : Read partition M^i_j from untrusted memory M
 3 : Read public hash value HASH_{M^i} from untrusted memory M
 4 : Read secret state STATE_{M^i} from trusted memory
 5 : Run CHECK(P,HASH_{M^i}, STATE_{M^i},i,M^i_j,j,K_i,P_i)
 6 : If (CHECK(.) returned 1)
 7 :   Pass contents of block x to processor
 8 : Else
 9 :   Abort with error message
```

Figure 4: `LOAD()` instruction with integrity checking

Figure 4 shows the steps executed during the `LOAD()` instruction which requests memory block x while Figure 5 enumerates the operations executed for the `STORE()` instruction which writes data y to memory block x. `STORE()` is more complicated than `LOAD()` due to the extra overhead incurred in constructing and maintaining the secret state associated with a set. In line 15 in Figure 5, even though it is not indicated, `UPDATE()` is run in an incremental fashion (as described in the previous section).

STORE (Memory block x, Data y)
1 : Determine set (i) and its constituent partition (j) to which block x belongs
2 : If (partition M_j^i is being written to for the first time)
3 : Read in the non-empty partitions of set M^i from untrusted memory M
4 : Write data y into block x in M_j^i
5 : Run UPDATE(P, M^i, K_i, P_i)
6 : Write $STATE_{M^i}$ in trusted memory
7 : Write $HASH_{M^i}$ and M_j^i in external memory
8 : Else
9 : Read partition M_j^i from external memory M
10 : Read hash value $HASH_{M^i}$ from untrusted memory M
11 : Read state $STATE_{M^i}$ from trusted memory
12 : Run CHECK(P, $HASH_{M^i}$, $STATE_{M^i}$, i, M_j^i, j, K_i, P_i)
13 : If (CHECK(.) returned 1)
14 : Write data y into block x in M_j^i
15 : Run UPDATE (P, M^i, K_i, P_i)
16 : Write updated $STATE_{M^i}$ to trusted memory
17 : Write updated $HASH_{M^i}$ and partition M_j^i to untrusted memory M
18 : Else
19 : Abort with error message

Figure 5: STORE() instruction with integrity checking

3.6 Security analysis

In this section, we present a discussion related to the security properties of the proposed method. We begin by presenting some lemmas related to the properties of different components used for realizing the proposed method. Then we present an analysis related to the completeness and soundness of the proposed method, and its resistance to adversarial attacks.

Lemma 1 [16]: *Given that the adversary chooses a set of n distinct binary strings each of length m, and then a degree-l irreducible polynomial is chosen, the probability of collision in Rabin fingerprinting scheme is* $< \frac{m \cdot n^2}{2^l}$

Proof: Let $H = \{h_1, h_2, \ldots, h_n\}$ be a set of n distinct m-bit binary strings where each string is represented as an $(m-1)$-degree polynomial, *i.e.*, $h_i(x) = h_{i_1}x^{m-1} + h_{i_2}x^{m-2} + \ldots + h_{i_{m-1}}x + h_{i_m}$, $1 \leq i \leq n$, and $P(x)$ be an l-degree irreducible polynomial. Define $A(x) = \prod_{i,j=1, i \neq j}^{n}(h_i(x) - h_j(x))$. Then

$$degree(A(x)) \leq (m-1) \cdot \frac{n(n-1)}{2} < m \cdot n^2$$

Without loss of generality, let us assume two distinct strings $h_i(x)$ and $h_j(x)$ map to the same fingerprint $f(x)$. We have
$(h_i(x) - h_j(x))$
$= (q_i(x)P(x) + f(x)) - (q_j(x)P(x) + f(x))$, for some $q_i(x)$ and $q_j(x)$
$= (q_i(x) - q_j(x))P(x)$
Thus, for any $h_i(x), h_j(x) \in H$, $P(x)|(h_i(x) - h_j(x)) \Rightarrow P(x)|A(x)$, if $h_i(x)$ and $h_j(x)$ map to the same fingerprint. The maximum number of l-degree irreducible factors in $A(x)$ is $degree(A(x))/l < (m \cdot n^2)/l$, and the total number of l-degree irreducible polynomials with coefficients in $\mathcal{Z}_2$ is $(2^l - 2^{l/2})/l$. Thus, we have
$Pr[$ Distinct strings in H map to the same fingerprint $] =$
$Pr[$ Randomly chosen l-degree irreducible polynomial divides $A(x)$ $]$

$< \frac{m \cdot n^2}{l} \ / \ \frac{2^l - 2^{l/2}}{l} = \frac{m \cdot n^2}{l} \cdot \frac{l}{2^l - 2^{l/2}} = \frac{m \cdot n^2}{2^l - 2^{l/2}} = \frac{m \cdot n^2}{2^{l/2}(2^{l/2}-1)} \approx \frac{m \cdot n^2}{2^l}$ (for large l)

Lemma 2: [18] *Hash function family $h_{b,x}$: $\mathcal{Z}_q^k \rightarrow \mathcal{Z}_q$, where q is prime, $b, x \in \mathcal{Z}_q$, where $h_x(m) = \sum_{i=1}^{k}(m_i + x_i)^2 \ mod \ q$, is Δ-universal.*

Proof: Let $m, n \in \mathcal{Z}_q^k$, $m \neq n$, with $m = (m_1, \ldots, m_k)$ and $n = (n_1, \ldots, n_k)$ and $a \in \mathcal{Z}_q$. Since $m \neq n$, without loss of generality assume that $m_1 \neq n_1$.
$Pr[h_x(m) - h_x(n)]$
$= Pr[\sum_{i=1}^{k}(m_i + x_i)^2 - \sum_{i=1}^{k}(n_i + x_i)^2 \equiv a \,(\text{mod } q)]$
$= Pr[2(m_1 - n_1)x_1 \equiv a - (m_1^2 + m_2^2) - \sum_{i=2}^{k}((m_i + x_i)^2 + (n_i + x_i)^2) \,(\text{mod } q)]$
$= 1/q$ (since $m_1 \neq n_1$ and there is a unique x which satisfies the equation)

Theorem 1: *The function composition $CRT_{(\pi(p_1),\pi(p_2),\ldots,\pi(p_n))} \circ F_P \circ H_K$ is complete and sound.*

Proof: For any set $M^i = \{M_1^i, M_2^i, \ldots, M_n^i\}$, the function composition $CRT_{(\pi(p_1),\pi(p_2),\ldots,\pi(p_n))} \circ F_P \circ H_K(M^i)$ generates a value X^i such that $X_j^i \equiv F_P(H_K(M_j^i))(mod\, p_{\pi(j)})$. The completeness property says that the function accepts legitimate inputs with very high probability. Since CRT is a bijection from $(\mathcal{Z}_{p_1} \times \mathcal{Z}_{p_2} \ldots \times \mathcal{Z}_{p_n})$ to $\mathcal{Z}_{\prod_{i=1}^{n} p_i}$, the function composition uniquely maps set M^i to value X^i such that $X^i(mod\, p_{\pi(j)}) = F_P(H_K(M_j^i))$ for any legitimate partition $M_j^i \in M^i$. Thus, the function is deterministically complete. The soundness property says that invalid input values are accepted successfully with a negligible probability. Without loss of generality, assume that set $M^{i'}$ is set M^i with the jth partition modified. We bound the probability of M^i and $M^{i'}$ colliding under function composition, *i.e.*, resulting in the same value X^i. Assuming l_1 is the hash value length, l_2 the signature length, and q the total number of primes such that $q = 2^l - 1$ for some l, we have
$Pr_{M^i \neq M^{i'}}[CRT_{(p_{\pi^i(1)},p_{\pi^i(2)},\ldots,p_{\pi^i(n)})} \circ F_P \circ H_K(M^i) = CRT_{(p_{\pi^i(1)},p_{\pi^i(2)},\ldots,p_{\pi^i(n)})} \circ F_P \circ H_K(M^{i'})]$
$= Pr_{M^i \neq M^{i'}}[F_P \circ H_K(M^i) = F_P \circ H_K(M^{i'})] * Pr_{M^i \neq M^{i'}}[(\pi^i(j) \text{ for } M_j^i) = (\pi^i(j) \text{ for } M_j^{i'})]$ (since CRT is a bijection, and for fixed moduli a collision of CRT output values occurs only when input values to CRT are the same).
$= (Pr_{M^i \neq M^{i'}}[H_K(M^i) = H_K(M^{i'})] + Pr_{M^i \neq M^{i'}}[(H_K(M^i) \neq H_K(M^{i'})) \bigwedge (F_P \circ H_K(M^i) = F_P \circ H_K(M^{i'}))])$
$\quad * Pr_{M^i \neq M^{i'}}[(\pi^i(j) \text{ for } M_j^i) = (\pi^i(j) \text{ for } M_j^{i'})]$
$\leq (\frac{1}{2^{l_1/2}}) + ((1 - \frac{1}{2^{l_1/2}})(\frac{n^2 l_1}{2^{l_2}})) * (\frac{1}{2^l - 1})$ (follows from Lemmas 1 and 2)
$\leq (\frac{n^2 l_1}{2^{l_2}})) * (\frac{1}{2^l - 1})$ (assuming l_1 is large enough).

Thus, the collision probability increases from $\frac{1}{2^{l_1/2}}$ to $(\frac{n^2 l_1}{2^{l_2}})) * (\frac{1}{2^l - 1})$. By making l_2 and l sufficiently large, we can make the collision probability of the function composition acceptably small. However, increasing the sizes of l_2 and l increases the space requirement.

Theorem 2: *Routines UPDATE() and CHECK() are (p,k)-secure against passive and active adversaries.*

Proof: Figure 6 shows the experiments carried out by active adversary A_{active} (on the right) and passive adversary $A_{passive}$ (on the left). $O^{\texttt{UPDATE}(K_i,P_i)}()$ indicates black-box access to routine `UPDATE()` with keys (K_i, P_i) embedded (and not divulged to the adversary). Similarly, $O^{\texttt{CHECK}(X_{M^i},K_i,P_i)}()$ represents oracle access to the `CHECK()` routine where the secret state X_{M_i} is embedded along with the keys and unobservable to the adversary. $A_{passive}$ observes the `UPDATE()` and `CHECK()` routines on uniformly random inputs (set M^i and partition number j) while the A_{active} generates its inputs according to a distribution of its choice and gives them to the routines. The secret state value X_{M_i} which is generated by the `UPDATE()` routine is hidden from the adversary. α and β represents information about the state gained by $A^{passive}$ and A^{active}, respectively. The experiment is repeated k times. Finally, based on the observations from the k runs, each adversary generates a hitherto unseen pair, partition and its hash value, with the intention of successfully fooling the `CHECK()` routine.

Experiment $\mathsf{EXP}^{A_{passive}}$ ()	Experiment $\mathsf{EXP}^{A_{active}}$ ()
$M^i = \{M_1^i, \ldots, M_n^i\} \stackrel{R}{\leftarrow} \{0,1\}^{n \cdot l}$	$M^i = \{M_1^i, \ldots, M_n^i\} \leftarrow A_{active}(1^{n \cdot l})$
$H_{M^i} \leftarrow O^{\texttt{UPDATE}(K_i, P_i)}(M^i)$	$H_{M^i} \leftarrow O^{\texttt{UPDATE}(K_i, P_i)}(M^i)$
$j \stackrel{R}{\leftarrow} \{0,1\}^{\lvert j \rvert}$	$j \leftarrow A_{active}(1^{\lvert j \rvert})$
$\alpha \leftarrow O^{\texttt{CHECK}(X_{M^i}, K_i, P_i)}(M_j^i)$	$\beta \leftarrow O^{\texttt{CHECK}(X_{M^i}, K_i, P_i)}(M_j^i)$
$(N, H_N) \leftarrow A_{passive}((\alpha, M^i, H_{M^i}, j)^k)$	$(N, H_N) \leftarrow A_{active}((\beta, (M^i, H_{M^i}, j)^k)$

Figure 6: Passive and active adversarial experiments

We show that if k (the number of queries by the adversaries to the black-box routines) is polynomially-bounded, then the probability of generating a tuple which successfully fools the CHECK() routine is negligible. First, let us consider $\texttt{EXP}^{A_{passive}}$. It has been proven that sets $\Omega_P \subset \mathcal{Z}_P$ created from sets $\Omega_{p_i} \subset \mathcal{Z}_{p_i}$ (where $P = \prod_{i=1}^{n} p_i$) by CRT appear to be randomly (Poisson) distributed for large primes [20]. Thus, for uniformly random sets M^i, state value X_{M_i} is uniformly distributed. In $\texttt{EXP}^{A_{passive}}$, adversary $A_{passive}$ can only observe the hash of uniformly distributed state value X_{M_i} which in turn is uniformly distributed (due to the property of hash functions). Thus, the information revealed to $A_{passive}$ is negligible, and so is the probability of $A_{passive}$ successfully fooling the CHECK() routine. Now, let us consider $\texttt{EXP}^{A_{active}}$. A_{active} can not only observe the hash of the state value, but has the additional capability of generating its own input values. Given $X \equiv r_i \, (mod \, p_i)$, $1 \leq i \leq n$, A_{active} has to either guess the r_i's with respect to (wrt) primes p_i, $1 \leq i \leq n$, or find r_i''s wrt a different set of primes p_i''s, $1 \leq i \leq n$, such that $X \equiv r_i' \, (mod \, p_i')$. Recall that there are q primes out of which n primes are chosen by the $\pi()$ mapping. Thus, there are $\left(\begin{array}{c} q \\ n \end{array} \right)$ ways of choosing n primes. Also, since CRT is a bijection from $(\mathcal{Z}_{p_1} \times \mathcal{Z}_{p_2} \ldots \times \mathcal{Z}_{p_n})$ to $\mathcal{Z}_{\prod_{i=1}^{n} p_i}$, for each X there is exactly one tuple $(r_1, \ldots, r_n)$ wrt to primes p_i, $1 \leq i \leq n$, such that $X \equiv r_i (mod \, p_i)$. We have

$p = Pr[\text{Adversary guessing } r_i \text{ wrt } p_i \text{ such that } X \equiv r_i (mod \, p_i), 1 \leq i \leq n \,]$

$= \sum_{(i_1, \ldots, i_n) \in [1..q]} \frac{1}{\left(\begin{array}{c} q \\ n \end{array} \right)} \cdot \frac{1}{(p_{i_1} \cdot p_{i_2} \cdots p_{i_n})}$

$= \left(\begin{array}{c} q \\ n \end{array} \right) \cdot \frac{1}{\left(\begin{array}{c} q \\ n \end{array} \right)} \cdot \frac{1}{(p_{i_1} \cdot p_{i_2} \cdots p_{i_n})}$

$= \frac{1}{(p_{i_1} \cdot p_{i_2} \cdots p_{i_n})}$

$< \frac{1}{2^{32 \cdot n}}$ (since $\lvert p_{i_j} \rvert \geq 32$, $1 \leq j \leq n$)

Thus, the probability p of active adversaries breaking the proposed scheme with k polynomially-bounded queries is negligible (which is also true for the less powerful passive adversaries).

4 Results

In this section, we present results comparing communication complexity of the proposed method with two existing methods, *store hash* and *Merkle tree*. We analytically derive the communication complexity of the three schemes for a typical memory configuration. Usually, memory integrity verification is applied only to the portion of memory storing secure code and data (termed *secure memory*) rather than to the entire memory. Based on typical secure applications, *e.g.*, digital rights management programs, we set the size of secure memory to $N = 1$ MB (2^{20} B). For memory block size of 64 B (2^6 B), $N = 2^{14}$ blocks. Assuming that secure memory is divided into sets of size 512 B (2^3 blocks), the number of sets $S = 2^{11}$ ($2^{14}/2^3$). With the above system parameter values, the working of three memory integrity schemes to be compared can be summarized as:

- *Store hash*: The S sets of secure memory are hashed, and the hash values are stored in on-chip storage. Hence, when a block belonging to a set is read, the entire set has to be read in order to verify its integrity.

- *Merkle tree*: A Merkle tree of height $log_2(2^{11}) = 11$ is constructed whose leaf nodes hold the sets while the internal nodes hold the hash values. Only the root hash is stored in on-chip storage. When a block is read from a set, the set and its sibling set along with $2 * 11 = 22$ hash values are read in order to verify the integrity of the block (by computing the root hash, and comparing it with the stored value).

- *Proposed*: We assume $\sigma = 1$ (σ is defined in Section 3.2), and fingerprint length to be 32 bits. Then, a set is divided into $n = (L/32)$ partitions where L is the hash length in bits, and the number of blocks per partition $b = \lceil 2^3/n \rceil$. In this scheme, the state values (each of length L since $\sigma = 1$) are computed for all the S sets and stored in on-chip storage. When a block is read, only the partition holding it is read from memory. Thus, the communication complexity is b blocks.

Table 2 shows the communication complexity results of the three schemes for three hash value lengths (16 Bytes, 20 Bytes, and 32 Bytes). The *store hash* scheme imposes an overhead which is independent of the hash size, and equal to the set size. The *Merkle tree* scheme has very high communication complexity which increases with hash size. In contrast, the proposed scheme imposes a much lower communication overhead compared to the other two, and moreover, the overhead deceases with increasing hash size.

Table 2: Communication complexity of the three schemes

Scheme	Communication complexity (Bytes)		
	16 Byte hash	20 Byte hash	32 Byte hash
Store hash	512	512	512
Merkle tree	2*(512+11*16) = 1376	2*(512+11*20) = 1464	2*(512+11*32) = 1728
Proposed	⌈(8/((16*8)/32))⌉*64 = 128	⌈(8/((20*8)/32)/4)⌉*64 = 128	⌈(8/((32*8)/32)/4)⌉*64 = 64

Both *store hash* and *proposed* schemes impose a space overhead of $2^{11} \cdot L$ to store the hash and state values in an on-chip storage, respectively, while the *Merkle tree* scheme only stores the root hash of length L on-chip. Hence, it would be fair to examine the factor by which the communication complexity of the *Merkle tree* scheme would improve if it is provided with on-chip storage for storing all the hash values held in its internal nodes. A Merkle tree with 2^{11} leaf nodes has $1 + 2 + 2^2 + \ldots + 2^{10} = 2^{11} - 1$ internal nodes. Thus, an on-chip storage of $(2^{11} - 1) \cdot L$ bits is enough to store all the hash values. With this storage, the communication overhead in a Merkle tree pertaining to reading in 11 hash values vanishes, *i.e.*, the factors 2*(11*16), 2*(11*20) and 2*(11*32) vanish from columns, corresponding to 16 B, 20 B and 32 B hash lengths in Table 2. However, this still leaves an overhead of 2*512 B related to reading in the sets. Thus, even when the storage complexity of Merkle tree is made comparable to that of the proposed scheme, the proposed scheme has much lower communication complexity.

5 Conclusion

In this paper, we proposed a novel memory integrity verification technique which achieves a constant communication complexity overhead while incurring a modest cost in space complexity. The scheme is based on CRT used in conjunction with standard hashing tools: cryptographic, universal and polynomial hashing. Our security analysis shows the proposed scheme is sound and complete and resist computationally-bounded adversarial attacks.

References

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. Intl. Conf. Arch. Support Prog. Lang. and Operating Sys. (ASPLOS)*, pp. 169–177, Nov. 2000.

[2] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. USENIX Operating Sys. Design and Impl. Symp.*, pp. 135–150, Oct. 2000.

[3] R. B. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proc. Intl. Symp. Comp. Arch.*, pp. 2–13, May 2005.

[4] R. Merkle, "A certified digital signature," in *Proc. Crypto '89*, pp. 218–238, Aug. 1989.

[5] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[6] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *Proc. Symp. Foundations of Comp. Science*, pp. 90–99, Oct. 1991.

[7] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to build a trusted database system on untrusted storage," in *Proc. USENIX Operating Sys. Design and Impl. Symp.*, pp. 135–150, Oct. 2000.

[8] W. Shapiro and R. Vingralek, "How to build a trusted database system on untrusted storage," in *Proc. Digital Rights Management Wkshp.*, pp. 176–191, Jan. 2001.

[9] P. T. Devanbu and S. G. Stubbleline, "Stack and queue integrity on hostile platforms," *Software Engineering*, vol. 28, pp. 100–108, Jan 2002.

[10] B. Chen and R. Morris, "Certifying program execution with secure processors," in *Proc. USENIX HotOS Wkshp.*, May 2003.

[11] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and merkle trees for efficient memory authentication," in *Proc. Intl. Symp. High Perf. Comp. Arch.*, pp. 295–306, Feb. 2003.

[12] D. Williams and E. G. Sirer, "Optimal parameter selection for efficient memory integrity verification using merkle hash trees," in *Proc. Intl. Symp. Network Comp. and Appl.*, pp. 383–388, July 2004.

[13] M. Bellare and P. Rogaway, "Collision-resistant hashing: Towards making UOWHFs practical," in *Proc. Crypto '97*, pp. 470–484, Aug. 1997.

[14] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: Incrementality at reduced cost," in *Proc. EuroCrypt '97*, pp. 163–192, May 1997.

[15] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their applications to memory integrity checking," in *Proc. AsiaCrypt '03*, pp. 188–207, Nov. 2003.

[16] A. Z. Broder, "Some applications of rabin's fingerprinting method," *Sequences II: Methods in Communications, Security, and Computer Science*, pp. 143–152, 1993.

[17] M. O. Rabin, "Fingerprinting by random polynomials," Tech. Rep. TR 15-81, Harvard University, 1981.

[18] M. Etzel, S. Patel, and Z. Ramzan, "Square hash: Fast message authentication via optimized universal hash functions," in *Proc. Crypto '99*, pp. 234–251, Aug. 1999.

[19] H. Riesel, *Prime Numbers and Computer Methods for Factorization*. Birkhauser, 1985.

[20] A. Granville and P. Kurlberg, "Poisson statistics via the Chinese remainder theorem," Tech. Rep. http://arxiv.org/abs/math.NT/0412135, 2005.