IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability

Ruirui Huang and G. Edward Suh
Cornell University
Ithaca, NY 14853, USA
{huang,suh}@csl.cornell.edu

ABSTRACT
This paper proposes a unified off-chip memory integrity protection scheme, named IVEC. Today, a system needs two independent mechanisms in order to protect the memory integrity from both physical attacks and random errors. Integrity verification schemes detect malicious tampering of memory while error correcting codes (ECC) detect and correct random errors. IVEC enables both detection of malicious attacks for security and correction of random errors for reliability at the same time by extending the integrity verification techniques. Analytical and experimental studies show that IVEC can correct single-bit errors and even multi-bit errors from one DRAM chip within a cache block read without any additional ECC bits, when the integrity verification is also required for security, effectively removing the memory and bandwidth overheads (12.5%) of typical ECC schemes. Alternatively, with parity bits, IVEC can provide even stronger error correction capabilities comparable to the traditional chip-kill correct, still with less overheads. For both cases, IVEC can use standard non-ECC DIMMs.

Categories and Subject Descriptors
B.3.4 [Memory Structures]: Reliability, Testing and Fault Tolerance—Error-Checking

General Terms
Design, Reliability, Security

Keywords
fault tolerance, error correction, error detection, memory systems, security, reliability

1. INTRODUCTION
Recently, there have been significant efforts to build secure and reliable computing devices through architectural enhancements. For security, researchers have proposed "secure processor" techniques in order to protect against both software attacks and physical attacks. At the same time, many techniques have been developed for reliability as the technology scaling expects to increase both transient and permanent faults in hardware. In a high level, both security and reliability deal with unintended errors in computing devices. Yet, most of today’s techniques only target either security or reliability separately, perhaps due to differences in fault/attack models. This paper investigates the potential synergy between security and reliability and shows that it is indeed possible to benefit from handling the two types of errors together. For off-chip memory, integrity verification that is designed for security can be extended for reliability with minimal costs and replace traditional Error Correcting Code (ECC).

The integrity verification mechanism in secure processors is designed to detect any unauthorized change in off-chip memory content. As we expand the use of computing devices, many systems are physically exposed to potential attackers; users may not be able to fully trust data center operators in cloud computing, embedded devices often operate in an open field, mobile devices may be lost or stolen, and device owners may be motivated to break the system security in cases such as copyright protection violation. In such environments, an attacker can physically tamper with the off-chip memory by either changing critical data or reading confidential information. As an example, the security of Xbox was broken by reading a secret key from an off-chip bus [13]. The integrity verification checks that a value from off-chip memory is the most recent one that was written by the processor at the memory address. If any error is detected, a processor aborts the program in order to prevent further damages. As devices continue to expand in the physical world, such hardware protection techniques for off-chip memory are likely to become more popular. In fact, IBM recently introduced a technology named SecureBlue that includes off-chip encryption and integrity verification. SecureBlue is reported to be in production for one IBM client already [4].

In order to tolerate random transient errors, today’s computer systems often have Error Correcting Code (ECC) for both on-chip and off-chip memory. Systems typically use a Single-Error-Correction and Double-Error-Detection (SEC-DED) code, which requires 8 additional parity bits per 64-bit chunk. The code guarantees detection of all 1-bit or 2-bit errors and correction of all 1-bit errors. As a result, a system can continue its operation under a small number of bit flips in memory. On the other hand, the SEC-DED ECC provides no guarantee beyond 2-bit errors within a 64-bit chunk.
chunk; an error may be detected, correctly corrected, or incorrectly corrected. To handle multi-bit errors, a technique called chip-kill correct [5] is widely used. The chip-kill correct can correct up to 4 adjacent bits at once, and hence can handle a complete failure of a x4 DRAM chip [17].

This paper investigates whether protecting memory from hardware faults can be simplified in secure processors when the integrity verification is also required for security. Intuitively, the integrity verification mechanism provides a stronger error detection capability than the traditional ECC because security must consider any changes including malicious tampering when ECC only handles random errors. This suggests that the error detection aspect of ECC is redundant if the integrity verification already exists.

However, today’s integrity verification techniques cannot simply replace ECC. In general, there are a couple of inherent differences between today’s security techniques and hardware fault tolerance techniques. First, security protection techniques tend to protect a smaller portion of a system compared to corresponding reliability techniques. For example, the integrity verification is only concerned with off-chip memory because it is much more difficult to tamper with on-chip memory. On the other hand, ECC is often applied to both on-chip and off-chip memory as both are equally vulnerable to hardware errors. Therefore, security techniques can only help a subset of reliability techniques when both protect the same components. This paper focuses on combining the off-chip integrity verification with off-chip ECC.

Another important difference between security and reliability lies in their responses to an error. Today’s security mechanisms including the integrity verification often focus only on error detection and do not attempt to recover from errors. While availability is important for security as well, the recovery is extremely difficult, if not impossible, when an adversary physically controls a system. For example, there is no way to “correct” errors when an attack can arbitrarily change memory content. On the other hand, error correction is common for reliability mechanisms that enable a system to continue its operation under random errors. Therefore, the integrity verification must be augmented with error correction in order to replace ECC.

Unfortunately, the error correction in today’s integrity verification is not straightforward due to its use of complex cryptographic signatures, often called hash or MAC. Unlike ECC where error locations can be mathematically obtained from parity bits, cryptographic primitives for integrity verification do not reveal where errors may be. In fact, if they do, such mathematical properties will be a security vulnerability in cryptography. Therefore, in the integrity verification, we need a different way to identify error locations. At the same time, unlike parity bits in ECC, cryptographic primitives do not consider errors in hashes or MACs, which requires a way to correct errors in them. Finally, we must ensure that the integrity verification is still secure even after the addition of error correction.

In this paper, we extend the integrity verification to correct errors by searching for error locations when an integrity check fails. We call this enhanced integrity verification IVEC (Integrity Verification with Error Correction). For example, to correct a single-bit error, IVEC tries to flip one bit at a time until the integrity check passes. Obviously, the naïve brute-force search does not scale for multi-bit errors. To handle multi-bit errors more efficiently, this paper presents intelligent search algorithms based on typical error patterns and selective use of parity bits. Essentially, our approach converts the error correction into a search problem. To correct errors in hashes/MACs, IVEC exploits a tree structure in common integrity verification schemes.

This paper also shows that IVEC can still ensure a proper level of security against malicious changes in memory. To break the integrity verification scheme, an attack must find a different value that produces the same signature (hash/MAC) as the authentic one in memory - this is often called a collision. The IVEC scheme slightly lowers the security of the baseline integrity verification mechanism because an attack only needs to be close enough to a collision instead of being an exact collision. Fortunately, however, our analysis shows that the lowered security is still sufficient for typical hash/MAC lengths.

Overall, the IVEC scheme enables efficient protection of off-chip memory from both security attacks and random errors without paying unnecessary memory and bandwidth overheads twice. IVEC can handle single-bit upsets and simple multi-bit errors within a single read without any ECC bits. With a small number of additional parity bits, IVEC offers a correction capability that is comparable to the chip-kill correct even for wide DRAMs. Thanks to its security protection, IVEC also provides a much stronger error detection capability than ECC. While the error correction takes longer in our approach than traditional ECC, experimental results suggest that the overall performance overhead is negligible even under pessimistic error rates. The paper also shows that the parity bits can be efficiently managed in regular memory space without requiring expensive ECC DIMMs because the parity bits are only used for correcting errors. In addition to reducing the ECC overheads for secure processors, IVEC also enables dynamic trade-offs between the error correction strengths and the error correction latency.

The rest of the paper is organized as follows. Section 2 discusses attack and fault models that IVEC aims to cover. Section 3 describes current integrity verification techniques for security, and Section 4 proposes IVEC. Section 5 discusses the hardware implementation details of IVEC. Section 6 evaluates IVEC for its error detection and correction capabilities as well as the performance and memory bandwidth impacts. Section 7 discusses related work in the area, and Section 8 concludes the paper.

2. ATTACK AND FAULT MODELS

2.1 Security Model

Figure 1 illustrates a security model that is commonly assumed for secure processors. Put briefly, a secure processor...
can be used with various length read-block configurations. In recent chip-kill correct implementations such as reads from each DRAM chip are needed to read one cache once. If a cache block is larger than the read-block, multiple such data blocks where each DRAM chip contributes only a bit to the entire block. If not, the last three models are identical. Typical chip-kill correct techniques [29, 17] can correct errors in up to 4 adjacent bits and are comparable to the MCME model.

3. INTEGRITY VERIFICATION

This section describes integrity verification algorithms, which ensure that the values stored in off-chip memory have not been changed. When describing the algorithms, this paper uses a term block as the minimum memory block that is verified by the integrity checking. If a word within a block is read by a processor, the entire block is brought into the processor and its integrity is checked. In this paper, blocks for integrity verification are identical to the L2 cache blocks.

3.1 Cryptographic Primitives

Integrity verification techniques typically use one of the two cryptographic primitives as basic building blocks: a cryptographic hash function or a message authentication code (MAC). Here, we briefly introduce the two primitives. A cryptographic hash of a message is a fixed length fingerprint of the message. One of the key properties of a cryptographic hash function is that it is hard to find two distinct inputs that produce the same hash. This property is called collision-resistance. The collision-resistance enables a hash function to protect the integrity of data if the hash can be trusted. For example, a file that is downloaded from a website can be checked by recomputing its hash if the correct hash is known. Because it is infeasible to find another file that produces the same hash, an adversary cannot change the file without changing the hash.

A message authentication code (MAC) is a keyed hash function where the hash depends on a secret key so that only parties with the correct key can compute a valid MAC. Unlike hashes that must be trusted, the secret key enables MACs to be either sent over untrusted networks or stored in an untrusted storage. As the name suggests, MACs are often used to authenticate a message sent over networks. A sender computes the MAC of a message and sends it along with the message. Later, a receiver recomputes the MAC of the received message and compares it with the attached MAC. If the MACs match, the receiver knows that the message is from the sender with the valid secret key.

There are many instantiations of the cryptographic hash and MAC. The standard hash functions are SHA-1 [23] and SHA-256 [24]. Both hash functions process 512-bit blocks. SHA-1 produces 160-bit hashes and SHA-256 produces 256-bit hashes. However, both hash outputs can be truncated to produce a shorter hash. For the MAC, HMAC [14] is the most widely used implementation today and takes two hash-
ing operations. To avoid the long latency of the hash functions (80 rounds), GMAC [7] has recently been proposed as one of NIST recommendations. GMAC uses the Advanced Encryption Standard (AES), which has only 10 to 14 rounds, as the underlying building block. GMAC can also be easily combined with encryption.

### 3.2 Hash/MAC Trees

To protect the integrity of off-chip memory without requiring an excessive amount of trusted on-chip memory, virtually all integrity verification techniques rely on Hash trees (or Merkle trees), which were originally developed for untrusted storage [20]. Figure 2 illustrates this hash tree algorithm applied to the off-chip memory verification.

The protected data blocks are located at the leaves of a tree. Each internal hash is computed over a block below. The root hash of the tree is stored in secure on-chip memory where it cannot be tampered with while others are stored in off-chip memory.

![Figure 2: A 4-ary hash tree assuming that one block can contain four hashes.](image)

In the figure, one block can contain four hashes, which results in a 4-ary hash tree (one parent hash covers four children hashes or one data block). For example, a 64-B block can have four 128-bit hashes. The arity of a tree can be increased by either increasing the L2 cache block size or reducing the length of a hash. To check the integrity of a block in the tree, the processor (i) reads the block, (ii) computes the hash of the block, (iii) checks that the resultant hash matches the parent hash, and (iv) checks the integrity of the block containing the parent hash. The steps are repeated all the way to the root of the tree. The root does not need to be checked as it is stored in on-chip memory.

To update a block, the processor (i) reads the block and checks its integrity, (ii) modifies the block, (iii) recomputes the parent hash, and (iv) updates the parent with the new hash. Again, these steps are repeated to update the whole path from the block to the root.

With a balanced \( k \)-ary tree, the number of blocks to be checked on each memory access is \( \log_k(N) \), where \( N \) is the number of blocks in the verified memory space. Unfortunately, this logarithmic bandwidth overhead of the hash tree can be significant. For example, a previous study [8] showed that applying the hash tree to protect 4-GB memory can slow down the system by as much as a factor of ten.

### 3.3 Cached Hash Trees

The bandwidth overhead of using a hash tree can be dramatically reduced by caching the internal hash blocks on-chip with regular data. The processor trusts data stored in the cache, and can access it directly without any checking. Therefore, instead of checking the entire path from the block to the root of the tree, the processor checks the path from the block to the first hash it finds in the cache.

The internal hash blocks can be cached either in the existing L2 cache with other data blocks or in a separate cache dedicated to the hash blocks. For a high performance processor with a large L2 cache, sharing the existing cache is likely to be a better choice because it provides a large cache for hashes without significantly increasing the logic area. On the other hand, embedded processors with only L1 caches should have a dedicated hash cache because sharing an L1 cache will result in a significant performance degradation.

Previous work [8] showed that the cached hash tree reduces the performance overheads of the integrity verification down to around 25% on average. Additional optimizations have been proposed to further reduce the overheads. Using GMAC in place of a hash function lowers the overheads to around 5% even with encryption [33] and Bonsai Merkle Trees report the overhead of 2% using small trees [27].

### 3.4 Security

To break the integrity verification scheme, an adversary needs to change a protected block in a way that its hash or MAC does not change, essentially finding a collision in the hash/MAC. Note that even though an attacker can change the protected block and its parent hash/MAC together at the leaves of the Merkle tree, one must eventually find a collision for the root hash/MAC that is stored on-chip.

![Table 2: Security of the integrity verification schemes with \( m \)-bit hashes/MACs.](image)

As shown in Table 2, an attacker can try to find a collision in a few different ways depending on the cryptographic primitives used in the integrity verification. Note that we do not consider attempts to mathematically break the cryptographic functions here. To find a collision with a particular memory block, an attacker needs to try one possible value at a time (brute-force search) by either computing a hash by himself or changing the block in the target system. For such a brute-force attack, the collision probability is \( 1/2^m \) given an \( m \)-bit hash/MAC function. In this scenario, an attacker is expected to try \( 2^m \) different values before finding a collis-
...and even 64-bit hashes or MACs are considered strong enough. Even if an attacker can try a few billion values per second, finding a collision will take over a thousand years.

Alternatively, an attacker can try to find a collision by constructing a table with a large number of hashes/MACs and their input values. In this scenario, the birthday problem in the probability theory shows that one can expect a collision within the table if there are $2^{m/2}$ entries. In the on-line birthday attack, an attacker constructs the table by observing all hashes or MACs that a processor produces. For hashes, attackers can also construct a table by computing hashes off-line by themselves. Once $2^{m/2}$ different values are in the table, an attacker can expect to find a collision. If the hash or MAC includes an address as well as data, then attacker needs to see $2^{m/2}$ values per cache block. If the birthday attack is possible, the hash or MAC needs to be at least 128 bits because 64-bit hashes can be broken only after 4 billion table entries.

However, the birthday attack is not always possible. For example, a GMAC-based scheme uses a counter value, which is unique for each MAC and whose integrity is protected separately. Therefore, even if a collision for a MAC is known, an attacker must also replace the counter value without being detected. This proposition requires another collision in the counter values, whose probability is only $1/2^m$.

4. IVEC

In this section, we propose our approach of extending an existing off-chip memory integrity verification scheme to perform memory error correction. To explain the key ideas, this section describes the IVEC algorithms based on the simple cached hash tree. However, the algorithms are easily applicable to other integrity verification schemes as well. For example, the IVEC can be implemented with MACs by simply replacing hashes in the algorithm with MACs. Section 5 discusses the details of applying IVEC to a state-of-the-art integrity verification based on GMAC [33].

4.1 Error Detection

The basic idea of the integrity verification scheme is to compute and store the hash of a memory block on a write. When the block is read back, the hash of the loaded block is re-computed and compared with the stored hash value. If a block in memory is tampered by an adversary, the change gets detected by an inconsistent hash value. Conceptually, from the physical perspective, there is no difference between data corruptions from random errors and corruptions from malicious attacks. Therefore, the integrity verification for security can also detect random errors.

In the integrity verification, a hash value is shorter than the memory block that is protected by the hash. As an example, hashes are often 64 or 128 bits while cache blocks are 512 bits. Hence, a hash function may map two or more input values to the same hash value, which is referred to as hash collision. As a result, the error detection in integrity verification is probabilistic and the false negative rate - the probability of not detecting an error - is determined by the hash collision rate. Fortunately, the false negative rate is extremely low even with typical hash lengths. Strong hash or MAC functions such as SHA-1 [23] and GMAC [7] uniformly distribute hashes in its output space. Therefore, the expected collision rate is inverse exponentially related to the hash output length: $1/2^m$ for $m$-bit hashes or MACs.

4.2 Error Correction as a Search

While the integrity verification scheme can be used for memory error detection, the existing integrity verification schemes cannot correct errors in a corrupted memory block. Here, we describe the IVEC error correction strategy, which searches for possible error locations. The discussion assumes that hashes (or MACs) are error-free and only considers one memory block and its hash. The next subsection removes this assumption by extending the error correction to a tree structure. IVEC scheme can handle all four previously defined fault models, and offer the flexibility to dynamically correct the target fault model of the system.

4.2.1 Error Correction without Parity

The error correction can be viewed as finding erroneous bits within a cache block. Once the error locations are known, those bits can be simply flipped to correct errors. Therefore, if we assume that the hash can detect any errors with a very high probability, we can correct errors by performing a search. As an example, for a single-bit error (SBE), we can try to flip one bit at a time, and check if the new value passes the integrity verification. In theory, if we exhaustively search through all possibilities, we can correct any error. However, in practice, the search space quickly explodes as the number of bit flips increases. To keep the search manageable, the IVEC algorithm restricts the search space based on the fault model and hardware configurations.

Figure 4 illustrates how a cache block is read from DRAMs assuming the configuration used by typical chip-kill correct: 4 DIMMs with 16 DRAM chips each ignoring ECC bits. In this configuration, a 64-byte cache block corresponds to two 32-byte memory read-blocks and each DRAM chip produces 4 adjacent bits. For our multi-bit fault models based on chip-kill correct (Table 1), the error search can be restricted to bit positions from one DRAM chip; bits from different DRAM chips in a read-block cannot have errors simultaneously.

Figure 3 describes the baseline error correction algorithm. The algorithm takes a cache block, its hash and the desired fault model as inputs and returns either a corrected block, if a block that matches the hash can be found, or a correction fail signal. As shown in the figure, the algorithm searches the error space based on the fault model, starting from more likely cases. The algorithm tries single-bit errors (SBE), multi-bit errors from a single DRAM chip in one read-block (SCSE), multi-bit errors from a single DRAM chip in both read-blocks (SCME), and multi-bit errors in any one DRAM chip in each read-block (MCME) in order. Our analysis in Section 6 shows that this baseline algorithm is efficient enough for the SBE and SCSE models in x4 DRAMs, but becomes rather expensive for stronger fault models.

4.2.2 Error Correction with Parity

This subsection extends the baseline error correction using additional parity bits, which reduces the error correction...
Error Correction Operation without Parity

1. For (i=0; i<READ_BLOCK_SIZE; i++) (a) Flip a bit in Block[i]. (b) If hash(Block) == CorrectHash, return Block.

2. If FaultModel == SCSE then (a) Try possible errors in Block[i:i+CHIP_WIDTH-1]. (b) If hash(Block) == CorrectHash, return Block.

3. If FaultModel == SCME then (a) Try possible errors in Read_Block[i:i+CHIP_WIDTH-1] and Read_Block[j:j+CHIP_WIDTH-1]. (b) If hash(Block) == CorrectHash, return Block.

4. If FaultModel == MCME then (a) For each error pattern in Read_Block[i:i+CHIP_WIDTH-1] and Read_Block[j:j+CHIP_WIDTH-1] (i) Try errors in Read_Block[i:i+CHIP_WIDTH-1] or Read_Block[j:j+CHIP_WIDTH-1].

Figure 3: Baseline Error Correction Algorithm. The algorithm assumes that each block consists of two Read_Block. Block[n] indicates nth bit of Block, and Block[n:m] indicate nth to mth bit in Block. Read_Blockk represents Block[(k-1)·READ_BLOCK_SIZE:k·READ_BLOCK_SIZE-1].

search space and significantly improve multi-bit error correction. Figure 5 illustrates how the additional parity bits are used. As shown, there are 4 parity bits for each read-block, and each covers one bit from every DRAM chip. For example, P1 is a parity bit that covers every DRAM chips’ left most bit in the first read-block. We can correct up to a complete chip failure in each read-block while using parity bits to guide our search. We note that the corresponding parity bits from two read-blocks can be combined into one for the SBE and SCSE fault models where only one read-block may have errors.

Figure 5: Parity protection in DRAM reads.

Figure 6 describes the error correction algorithm with parity bits. In the algorithm, a sub-block is defined as bits in a read-block that are protected by one parity bit. A parity error indicates that a particular sub-block has an odd-bit error. In our fault models, at most one DRAM chip in each read-block would be erroneous. As each parity bit in our scheme only covers one bit from each chip, a parity error indicates a 1-bit flip within a sub-block. Also, the fault models imply that all errors in a read-block are from the same DRAM chip and thus the same location in each erroneous sub-block. For example, parity upsets in Parity[1][1] and Parity[2][1] (Parity[i][j] is defined in Figure 6) indicate that the first and second bits of one chip in the first read-block have errors. Therefore, the search algorithm simply needs to identify which DRAM chip is erroneous.

The algorithm for the first and second bits of each chip in the first read-block is as follows:

1. For (i=0; i<READ_BLOCK_SIZE; i++) (a) Flip a bit in Block[i]. (b) If hash(Block) == CorrectHash, return Block.

2. If FaultModel == SCSE then (a) Try possible errors in Block[i:i+CHIP_WIDTH-1]. (b) If hash(Block) == CorrectHash, return Block.

3. If FaultModel == SCME then (a) Try possible errors in Read_Block[i:i+CHIP_WIDTH-1] and Read_Block[j:j+CHIP_WIDTH-1]. (b) If hash(Block) == CorrectHash, return Block.

4. If FaultModel == MCME then (a) For each error pattern in Read_Block[i:i+CHIP_WIDTH-1] and Read_Block[j:j+CHIP_WIDTH-1] (i) Try errors in Read_Block[i:i+CHIP_WIDTH-1] or Read_Block[j:j+CHIP_WIDTH-1].

The write operation in IVEC is the same as the cached hash tree except for updating parity bits if they exist.

4.4 IVEC Parity Management in DRAMs

The baseline IVEC scheme completely eliminates the need for ECC bits when the integrity verification exists. Therefore, the baseline scheme can simply use non-ECC DIMMs. The parity scheme, on the other hand, requires a small number of parity bits on top of the integrity verification. In a naive approach, the parity bits can be stored in additional bits in ECC DIMMs. However, this approach requires special DIMMs with fewer ECC bits in order to benefit from the small number of parity bits. Instead, we propose to store parity bits in regular memory space in the same way that the integrity verification stores its meta-data. The parity bits are protected in the same way that data and hashes/MACs are. Therefore, before an error correction operation is performed on data or hashes, the associated parity bits are read, checked and possibly corrected first using the same algorithms shown in this section.

This configuration allows the IVEC schemes to utilize non-ECC DIMMS even with parity bits, and also exploits the inherent decoupling between error detection and correction. Because hashes/MACs detect errors, IVEC only uses the parity bits in a rare case when there is an error. Therefore, without an error, the parity bits do not need to be read from memory on a read. On an L2 write-back, the parity bits for the evicted block are updated. This optimization reduces...
Error Correction Operation with Parity

```plaintext
corrected = correct(dead, parity, CorrectHash, FaultModel);
1. Check parity bits.
   (a) SumParity = the number of parity errors in read-block j.
   (b) ParityUpset[j][1] = 1 if SubBlock[j] has a parity error.
2. If SumParity == 0 for all j.
   Indicates errors in multiple DRAM chips in a read-block, return unsuccessful.
3. If FaultModel == (SBE|SCSE|SCME) then
   For (n = 0; n < SB_SIZE; n++)
   (a) Flip SubBlock[n][m] if ParityUpset[m][n] == 1.
   (b) If hash(Block) == CorrectHash, return Block.
4. If FaultModel == SCME then
   For (n = 0; n < SB_SIZE; n++)
   (a) Flip SubBlock[n][m] if ParityUpset[m][n] == 1.
   (b) For (m = 0; m < SB_SIZE; m++)
      i. Flip SubBlock[m][n] if ParityUpset[m][n] == 1.
      ii. If hash(Block) == CorrectHash, return Block.

Parity and SubBlock Definition
1. SubBlock[j] = jth bit from each chip in read-block j.
2. Parity[i][j] = a parity bit for SubBlock[j].
SB_SIZE is the number of bits in a sub-block.
SB_SIZE = BLOCK_SIZE/PARITY_LENGTH.
```

![Figure 6: Error Correction Algorithm with Parity.](image)

To Read an IVEC Protected Block from Memory

```plaintext
tree_read(Address)
1. Check an on-chip cache. Return Block on a hit.
2. Read Block from Address in memory.
3. Compute Hash' = hash(Block).
4. Read the parent hash of the Block.
   (a) If the parent is the root, Hash = RootHash.
5. If MCME
   i. HashAddress = get_hash_address(Address).
   ii. Hash = tree_read(HashAddress).
6. Check hash == Hash', if so return Block.
7. If not, call correct(Block, Hash, FaultModel).
   (a) If correct() returns Unsuccessful,
      raise an exception.
   (b) else return Block.
```

![Figure 7: IVEC algorithm for a read.](image)

5. HARDWARE IMPLEMENTATION

This section discusses the implementation details of IVEC. Our implementation is based on a previously proposed integrity verification scheme [33], which uses GMAC as its MAC function. We refer to this baseline scheme as GMAC-Tree. The integrity verification aspect of our IVEC implementation is essentially identical to GMAC-Tree. IVEC only differs in its error correction operations when the integrity verification fails. Here, we briefly summarize the common GMAC-Tree operations and discuss the additional hardware that is required for IVEC compared to GMAC-Tree. We note that GMAC-Tree is chosen as a representative scheme. The IVEC implementation can be easily adapted to other integrity verification schemes.

In addition to a secret key and an input string to protect, GMAC requires another constant that is unique for each MAC. For this purpose, the GMAC-Tree scheme [33] maintains a counter per memory block, which is incremented on each write to the corresponding block. The counter concatenated with a block’s address provides the unique constant for GMAC. As an optimization, GMAC-Tree uses split-counters where an 7-bit counter is maintained per block to reduce overheads and a 64-bit counter is used per page to avoid overflows. The counters are cached on-chip for performance. Our IVEC implementation adopts the same split-counter design. The GMAC-Tree paper provides more details on the baseline integrity verification [33].

The use of a counter in GMAC enables efficient operations both for integrity verification and error correction. The GMAC computation consists of three operations: an AES encryption of a counter value, Galois field (GF) multiplications with XORs on a memory block, and an XOR of the two results. As shown in Figure 8, the AES operation can be performed in parallel to reading a memory block if a counter value exists in a cache, which is the common case. In this case, only the GF multiplication and XORs need to be done after the block arrives from memory, which adds only an 8 cycle latency for a 512-bit block even with conservative estimates [33]. Moreover, the AES result does not change when IVEC re-computes the MAC for error correction. As a result, each error correction try requires only the GF multiplication and XORs.

The integrity verification schemes commonly optimize performance by allowing a processor to use a block from memory before the integrity check completes (see Figure 8). To ensure security, the processor delays the commit (retire) of the load instruction until the check is done. If the check fails, the processor flushes the pipeline and re-executes from the failed load after an error correction. To avoid the hardware complexity, it is also possible to simply delay memory reads and wait for integrity checks. The previous study shows even this conservative approach only results in a 6% slowdown with GMAC-Tree [33].

The IVEC scheme needs to store meta-data such as MACs, counters, and optional parity bits in off-chip memory. As discussed in the previous section, we store parity bits in the standard memory space along with other meta-data. Therefore, the IVEC scheme can be implemented using non-ECC DIMMs without requiring special DIMMs with additional parity bits. The meta-data are stored in a part of the physical memory in a way that the IVEC unit can compute the addresses of the parent MAC, the counter, and the parity bits from a memory block’s physical address. This approach is the same as other integrity verification schemes. The MAC
tree protects the counters, data, and parity bits (if used) as leaf nodes to ensure their integrity. The root MAC and counter are stored on-chip.

Figure 9: The hardware block diagram. The dark (blue) modules indicate additional modules necessary for IVEC.

Figure 9 shows the high-level block diagram of the IVEC implementation. The light (yellow) blocks indicate the ones that already exist in the baseline GMAC-Tree scheme. The dark (blue) blocks indicate the new blocks for IVEC. On an L2 read miss, the access is queued in a load queue (LDQ), which initiates a read from memory. At the same time, the parent MAC gets loaded from the L2 cache to a MAC queue (MACQ), and the counter is loaded from the counter cache for an AES encryption. When a memory block arrives, the GF multiply result is computed and queued in the IV queue along with the AES result for the integrity check.

IVEC only adds a few small modules to GMAC-Tree. First, a data queue stores a block from memory while its integrity is being checked. Second, the correction buffer stores information for memory blocks whose integrity check failed. The buffer entry includes the AES result, the memory block, the parent MAC, parity bits, and an identifier or an address. The depth of the buffer needs to match the height of the MAC tree to correct errors from the root to a leaf in the worst case. A parity cache is used to load parity bits on an error detection, and to store parity bits on an L2 write-back operation. Alternatively, the parity bits can share the L2 cache similar to the way MACs are stored in L2. Section 6 studies both design options and the impact of different parity cache sizes. Finally, the IVEC control unit is a simple FSM to try the error correction. Note that the GMAC computation can simply reuse the existing GF multiply unit. Given that the area overhead of the integrity verification is dominated by the AES unit and the counter cache, the overhead of IVEC is quite small. Even with 128-bit MACs and 512-bit blocks, the additional buffer space for IVEC excluding the parity cache will be less than 16KB.

6. EVALUATION

This section evaluates various aspects of the IVEC scheme including its effectiveness such as the security and the error detection and correction capabilities, and the overheads in memory space, performance, and memory bandwidth usage. The evaluation focuses on five IVEC configurations, IVEC with no parity (IVEC-NP), IVEC with 4-bit parity per block (IVEC-P4), IVEC with 8-bit parity per block (IVEC-P8), IVEC with 16-bit parity per block (IVEC-P16) and IVEC with 32-bit parity per block (IVEC-P32). IVEC-NP represents the baseline that can handle the SBE and SCSE fault models with minimal overheads, while IVEC-P4 can handle the SBE and SCSE fault models more efficiently than IVEC-NP. IVEC-P8, IVEC-P16, and IVEC-P32 present the schemes to handle the worst-case fault model (MCME) for x4, x8, and x16 DRAMs, respectively.

### 6.1 Analytical Study

This subsection presents analytical studies on various aspects of the IVEC scheme, including the detection and correction rates, mis-correction rate, correction latency, and security. Table 3 summarizes the analysis results using the following parameters: $n$ for the memory block size in bits, $m$ for the hash length in bits, $h$ for the hash latency in cycles, $p$ for the number of parity bits per block, and $k$ for the worst-case number of correction attempts in the IVEC scheme.

<table>
<thead>
<tr>
<th>Detection Rate</th>
<th>Miscorrection Rate</th>
<th>Correction Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 - \frac{1}{2^m}$</td>
<td>$1 - (1 - \frac{1}{2^m})^k$</td>
<td>$k \cdot h$</td>
</tr>
</tbody>
</table>

Table 3: IVEC error detection and correction
Table 4: The maximum number of correction attempts under typical IVEC configurations. The table assumes 64-byte cache blocks and 32-byte read-blocks.

<table>
<thead>
<tr>
<th>Parity</th>
<th>SBE</th>
<th>SCSE</th>
<th>SCME</th>
<th>MCME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x4</td>
<td>x8</td>
<td>x16</td>
<td>x4</td>
</tr>
<tr>
<td>None</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>2432</td>
</tr>
<tr>
<td>4 bits</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>8 bits</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>16 bits</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>32 bits</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5: Memory space overhead of various protection schemes assuming 64-bit MACs and 512-bit blocks.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>ECC</th>
<th>IV (64b/54b)</th>
<th>IV+ECC (64b/54b)</th>
<th>IVEC-NP</th>
<th>IVEC-P4</th>
<th>IVEC-P8</th>
<th>IVEC-P16</th>
<th>IVEC-P32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overheads</td>
<td>12.50%</td>
<td>16.32% / 13.72%</td>
<td>25.82% / 26.22%</td>
<td>16.36%</td>
<td>17.44%</td>
<td>18.48%</td>
<td>20.61%</td>
<td>24.88%</td>
</tr>
</tbody>
</table>

6.3 Performance Simulation Methods

To evaluate the performance and off-chip bandwidth usage of IVEC, we used detailed simulations. The simulation infrastructure is built with the Pin binary instrumentation tool [18] and the TAXI performance simulator [32]. The Pin tool runs a program and sends execution traces so that the TAXI simulator performs a cycle-by-cycle micro-architecture simulation that models the processing core and the memory hierarchy. The performance simulations use benchmarks from the SPEC CPU2000 suite [11]. For each benchmark, we use a reference input set and its early single simulation points respectively [26]. Table 6 summarizes the simulation parameters. The size of the parity cache was selected as a fraction of the L2 cache size based on the ratio between the parity bits and a L2 cache block.

Table 6: Simulation parameters.

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>3 GHz, 4-issue out-of-order</td>
</tr>
<tr>
<td>Caches</td>
<td>L1:16b: 64KB/64KB 2-ways 2 cycle latency, L2: 2-MB unified, 8-ways, 14 cycle latency</td>
</tr>
<tr>
<td>DRAM</td>
<td>4GB, Latency: 50 ns, Bandwidth: 8 GB/s</td>
</tr>
<tr>
<td>IV Checks</td>
<td>Split Counter GMAC: Merkle Tree, 64-bit GMAC</td>
</tr>
<tr>
<td>AFS</td>
<td>128-bit, 80 cycle latency</td>
</tr>
<tr>
<td>GF Multiply</td>
<td>8 cycle latency</td>
</tr>
<tr>
<td>Counter cache</td>
<td>32KB (IVEC-P4), 64KB (IVEC-P8), 128KB (IVEC-P16), 256KB (IVEC-P32), 8 ways, 2 cycle latency</td>
</tr>
<tr>
<td>Parity cache</td>
<td>32KB (IVEC-P4), 64KB (IVEC-P8), 128KB (IVEC-P16), 256KB (IVEC-P32), 8 ways, 2 cycle latency</td>
</tr>
</tbody>
</table>
the baseline configuration, and focuses on evaluating the overheads of the IVEC scheme compared to the conventional integrity verification. The average performance overhead of the baseline is reported to be less than 5% when compared with a system without any protection [33].

6.4 Performance

There are two potential sources of the performance overheads for IVEC compared to the baseline integrity verification scheme: error correction latency and off-chip bandwidth usage from parity updates. However, the correction latencies only come into effect when there is an error on a memory read. During the normal operation, the error correction does not add any additional overhead. The soft-error rate (SER) for modern DRAM cells is reported to be 100 to 1000FIT/MB [30]. For 4GB memory, this SER translates to about an error every 10^{11} seconds. Given that the worst-case error correction takes tens of thousands of cycles in the IVEC configurations that we study, the performance overhead from error correction will be negligible. We note that in a case where the SER is several orders of magnitude higher than the current SER, the slower error correction of IVEC will become more visible.

Because our IVEC implementation maps parity bits to regular memory space without using special DIMMs, the additional parity accesses may interfere with processor's memory accesses. Figure 10(a) shows the performance of various IVEC configurations normalized to the baseline integrity verification scheme. Here, simulations assume that there is no memory error. Note that the performance of IVEC-NP is not shown in the graph because it is identical to the baseline. As shown in the figure, the performance overheads from additional parity accesses are less than 0.5% in the worst case. Figure 10(b) shows the IVEC-P16 scheme with various parity cache sizes, and also the alternative hardware design where parity bits share the L2 cache similar to the way MACs are stored in L2. Overall, the performance overhead increases with smaller parity cache sizes, most apparent in the memory intensive benchmarks such as equake. The shared L2 configuration also increases the performance overhead on average.

6.5 Memory Bandwidth Usage

This subsection studies the bandwidth usage of IVEC compared to the baseline integrity verification. While the performance impact of additional bandwidth consumption for parity bits has been studied in the previous subsection, the bandwidth usage is also a measure of the DRAM power consumption. Again, the off-chip bandwidth usage of IVEC-NP is almost identical to that of the baseline integrity verification. Therefore, IVEC-NP saves roughly 12.5% of off-chip traffic compared to the case where a traditional ECC is simply added to the integrity verification.

Table 7 shows the baseline memory bandwidth and Figure 11 shows the bandwidth usage overhead for different IVEC configurations. Overall, as the number of parity bits increases, the overhead of bandwidth traffic increases accordingly. In Figure 11(a), the bandwidth overheads are always less than the traditional ECC’s 12.5%. The average bandwidth overhead is at most 3.2% even for the IVEC-P32 case, which is considerably lower than ECC’s. In Figure 11(b), a larger parity cache helps to reduce the bandwidth traffic, since it would improve the miss rate of the cache. The shared L2 configuration shows comparable overhead to the default 128KB parity cache size for IVEC-P16.

6.6 Strengths and Limitations

Overall, the experiments illustrate two key advantages of IVEC compared to implementing the integrity verification and ECC separately. For simple fault models such as single-bit errors (SBE) or multi-bit errors from one chip in one read-block (SCSE), IVEC can provide error correction capabilities with negligible memory and bandwidth overheads beyond the baseline integrity verification (IVEC-NP). Alternatively, with parity bits, IVEC can handle a fault model (MCME) that is comparable to the chip-kil correct coverage, still with significant memory and bandwidth savings. For x4 DRAMs, IVEC-P8 has 2.1% memory overheads for parity bits and 0.9% bandwidth overheads. In addition, IVEC provides the flexibility to dynamically change the desired error protection level by controlling the search space.

IVEC also has a couple of limitations. First, the scheme is mainly useful for systems where both physical security and reliability are important. IVEC is more expensive than a simple ECC if only 1-bit correction is necessary without security. However, we believe that both physical security and reliability will become increasingly important for future computing devices. Second, while the overall performance overhead is negligible, each error correction is much slower in IVEC compared to ECC. Therefore, the current IVEC design is not well suited for real-time systems with tight deadlines.

7. RELATED WORK

This section summarizes the existing integrity verification and ECC schemes. The section also covers an existing technique that combine security and reliability concerns.

Memory Integrity Verification: To protect the integrity of off-chip memory without relying on an excessive amount of trusted on-chip memory, almost all integrity verification techniques rely on the tree structure. The original Merkle tree algorithm was proposed for untrusted storage in general [20]. Gassend et al. first proposed to apply the Merkle tree to protect off-chip memory with caching [8]. A split counter mode memory encryption and integrity verification is proposed using GMAC to further improve the integrity verification latency and overheads [33]. Bonsai Merkle tree [27] improves the performance and makes the integrity verification more OS friendly by using smaller Merkle trees to only authenticate counters. The IVEC scheme can be applied to any of the previous integrity verification schemes based on trees.

ECC: The SEC-DED ECC is originally introduced by Hamming [10]. Hsiao [12] proposed a class of optimal SEC-DED codes that simplifies the decoding logic. SEC-DED codes has also been further extended to detect single byte errors [3, 9] or correct 2-bit errors in adjacent bits [6] with the same number of bits. To handle multi-bit errors that SEC-DEC code cannot cover, researchers have proposed double-error-correcting triple-error-detecting (DEC-TED) codes [15, 2]. However, DEC-TED codes need 15 bits for each 64-bit data block with more complex encoding and decoding, almost doubling the memory overhead of SEC-DEC codes. The encoding and decoding latencies both double while the area increases by 3x for encoder and 5x for decoder [22]. Overall, although there are numerous extensions which im-
prove upon the SEC-DED ECC, the SEC-DED code still remains the most popular choice in the industry due to its simple encoding and decoding and low overhead in terms of speed, area, and power consumption.

Chip-kill Correct: Physical bit interleaving can also be used to address multi-bit error protection [25, 19]. A widely used technique is called chip-kill correct [5, 17], which can correct up to 4 adjacent bits at once, and hence can handle a complete failure within a x4 DRAM chip. The chip-kill correct can also handle a complete failure within a x8 DRAM chip when using a different SEC-DED code that handles 4 adjacent bit errors [5]. The chip-kill correct mechanism is observed to reduce the uncorrectable error rate by up to 10 times when compared to the traditional ECC in a real-world study [29]. IVEC with a small number of parity bits can handle a fault model (MCME) that is comparable to chip-kill correct, still with much less memory and bandwidth overheads than the traditional chip-kill correct single-bit errors or multi-bit errors from one DRAM chip in one read-block without any additional ECC bits. Also, IVEC’s “chip-kill” protection can be applied even for wide DRAM chips (x8, x16) where traditional chip-kill correct cannot be efficiently applied.

Tiered ECC: To improve the power and latency overheads of ECC, Sadler et al. proposed to decouple error-detection from error-correction so that a low-cost code can be used for detection while traditional ECC is used only for infrequent correction [28]. A similar idea is also used to reduce the power consumption of on-chip ECC by power-gating the ECC portion of clean cache lines [16], and to store expensive error correction code in memory while only maintaining detection code on-chip for caches [34]. A recent study uses virtualized two-tiered error protection code to provide chip-kil correct like protection using existing DRAM and packaging technology [35]. IVEC uses the same idea to optimize its parity bits. In our case, the hashes/MACs for integrity verification serve as a detection code, and the parity bits are used for correction.

Reliability and Security Engine: The Reliability and Security Engine (RSE) aims to solve security and reliability concerns together [21]. RSE is a common processor-level framework that embeds hardware-implemented modules which monitors the processor pipeline for both security and reliability checks. While the idea of combining security and reliability is the same with IVEC, RSE is applicable to a completely different domain.

8. CONCLUSION

This paper proposes to extend the integrity verification with error correction capability. By removing the error detection aspect of ECC, which is redundant when the integrity verification exists, the proposed IVEC scheme enables efficient protection of off-chip memory from both security attacks and random errors. For example, IVEC can correct single-bit errors or multi-bit errors from one DRAM chip in one read-block without any additional ECC bits. Moreover, with parity bits, IVEC can handle a stronger fault model (MCME) that is comparable to chip-kill correct, still with much less memory and bandwidth overheads than the traditional SEC-DED ECC. The analytical and experimental studies show that IVEC can be applied with minimal overheads and without compromising the security. Overall, we believe that the IVEC scheme can eliminate the overheads.
of traditional SEC-DED ECC for off-chip memory when a system requires integrity verification for physical security.

9. ACKNOWLEDGMENT

We thank anonymous reviewers for their feedback. This work was partially supported by the National Science Foundation under grants CNS-0746913, CNS-0708788 and SA4897-10808PG, by the Air Force Office of Scientific Research under Grant FA9550-09-1-0131, and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, AFOSR, or Intel.

10. REFERENCES