

Architectures for Secure Processing

Matt DeVuyst

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404
mdevuyst@cs.ucsd.edu

Abstract

Recently, secure processors have been proposed to solve a number of problems including: preventing unauthorized program execution or duplication, protecting user data from unauthorized access, ensuring the trustworthy execution of a program on a potentially compromised system, and building a foundation for secure distributed computing environments. This paper explores the recent research in this field and examines how state-of-the-art secure processors could be built. Focus is given to general-purpose architectures that guard the execution of processes not just from one another, but from a compromised operating system or a physical attack. Upon such a framework a number of secure computing paradigms can be built. This paper will also touch on some open questions in the field of secure processing architectures.

1 Introduction

Every domain in computer security has its own solution customized to meet specific needs. However, virtually all of these solutions can benefit from a general purpose security architecture. Many researchers focus on preventing specific security violations. For example, recent attention has gone into preventing buffer overflow attacks. This work is orthogonal to the more generalized approach taken in this paper. A secure computing architecture provides a solid foundation for secure software applications.

Hardware structures built with secure computing in mind can add significantly to the performance of secure computation. All domains of computer security share a common set of primitives—like encryption and hashing—and the performance of secure computing solutions are greatly enhanced if these primitives can be implemented in hardware instead of software.

Architectural support for secure computing also adds a level of strength that cannot be achieved in software alone

because software relies directly on the hardware on which it runs; if the hardware is the point of attack, then software is usually powerless to stop it. A key advantage of a secure architecture is that cryptographic keys never need to leave the processor as they would for a software-only approach, limiting the exposure of the keys and the potential points of attack. Adding the necessary security functionality to processors, enabling them to carry out security routines or bootstrap software-based security routines, shrinks the trusted computing base (TCB) down to the processor. Having a small TCB is desirable because there is less to verify and, therefore, fewer points of vulnerability.

Threat Model In this paper we will make some assumptions to clarify what threats we wish to address.

1. We assume that the user trusts the computation of the secure processor. That is to say, we assume that an attacker cannot directly interfere with any computation done inside the processor package. We assume that the secure processor is the only piece of hardware trusted by the user. All computer components outside the processor are not trusted. This includes main memory, disks, network, etc.
2. We assume that an attacker may both inspect and modify any data stored off-chip. In particular we consider the possibility that an attacker has physical access to the machine and is able to probe the memory bus and change the contents of memory, read from and write to the disk, eavesdrop on all network traffic and inject his own packets.
3. We assume that the user trusts the correctness of the programs he chooses to execute. We also assume that these programs do what is in their power to safely execute in a hostile environment including, but not limited to, running secure networking protocols (IPsec, SSL, etc), encrypting file contents, and authenticating users.

4. We assume that all other running processes (those executed by other users) including operating system processes are potentially malicious. In particular, other processes may be reading or writing in the memory space of the trusted process.

Expectations of a Secure Processor The above set of assumptions that define our threat model lead us to now define what expectations we have of a secure processor.

Execution Privacy A secure processor should ensure that the execution of the trusted application be kept private from other processes and from an attacker with physical access to the computer. By this we mean that an attacker should not be able to inspect application code or data; application state should be unreadable to an attacker with the capabilities we have specified.

Execution Integrity A secure processor should ensure that the integrity of the execution of the trusted application be maintained. An attacker should not be able to modify application state (code or data).

The above set of expectations of a secure processor affect how the processor is designed at many levels. But the most challenging design decisions relate to how the processor interfaces with memory. External memory contains application state, whether it be code, data, or even the register state of swapped out processes. It is crucial that execution privacy and integrity be maintained in the processor's interactions with memory. This will be the focus of the paper—to explore how this can be accomplished.

In summary, architectural support for secure computing is attractive for its generality, performance characteristics, and added strength. The two foundations of secure computing architectures, execution privacy and execution integrity, are discussed in Sections 2 and 3, respectively. We present several notable architectures that provide general-purpose secure computing platforms in Section 4. Finally, we explore future research directions in Section 5 and draw conclusions in Section 6.

2 Execution Privacy

One of the two foundations of all secure computing architectures is execution privacy (the other foundation, execution integrity, is discussed in Section 3). Execution privacy is simply the non-disclosure of run-time data to unauthorized parties. Run-time data includes program variables on the stack and in the heap and the code itself. The boundary that separates trusted space from untrusted space is the processor package. Everything outside the processor package is untrusted and vulnerable to attack; everything inside

the processor package is considered trusted and invulnerable to attacks during run-time. The most common way for a process' run-time data to leave the boundary of the processor package is through transactions with main memory or off-chip caches (if they exist). Therefore, to protect against exposure of sensitive run-time data, all data sent to memory external to the processor package must be encrypted and all encrypted data retrieved from memory must be decrypted. In this section we will discuss some encryption techniques to achieve the necessary privacy of memory traffic.

2.1 Naïve Encryption

In the simplest case, a hardware encryption/decryption unit (EDU) resides in the processor on the path between the on-chip cache and external main memory. The EDU encrypts all data evicted from the cache to memory and decrypts all data loaded from main memory into the cache. The encryption algorithm must be sufficiently strong because an attacker may probe the memory bus, log the traffic, and launch a cryptanalysis attack offline. The Data Encryption Standard (DES) algorithm [14] is too weak to be used; while its successor, the Advanced Encryption Standard (AES) algorithm [15], is the algorithm of choice in most of the literature.

The cipher block chaining mode of AES is often used (AES-CBC) because it prevents patterns among repeating blocks in the plaintext from showing up as patterns in the ciphertext. It accomplishes this by causing each ciphertext block after the first to be dependent on all plaintext blocks up to that point. To ensure that the first plaintext block is sufficiently difficult for an attacker to decrypt, the plaintext of that block is first XORed with a random number called the initialization vector (IV). As a result of AES-CBC being used, encryption takes longer than decryption because most of the work in the decryption process is done in parallel while the encryption process is almost entirely serial. The relative latency of encryption to decryption depends on the size of the data to be encrypted/decrypted, which is usually equivalent to the length of a cache line. AES typically operates on data chunks 128 bits in size because smaller blocks are less secure and larger blocks are computationally more intensive; so, if the cache line length is 64 bytes, 4 AES operations must be performed to encrypt or decrypt the data. In this case, encryption would take 4 times as long as decryption. To illustrate why, see Figure 1. During encryption, each chunk of plaintext (except the first) must be XORed with the ciphertext data of the previous chunk before it can be encrypted. Thus, the AES operation for each chunk depends on the AES operation on the previous chunk. During decryption, each chunk of ciphertext data is decrypted in parallel, and then each chunk (except the first) is XORed with the ciphertext of the previous chunk to produce the plaintext.

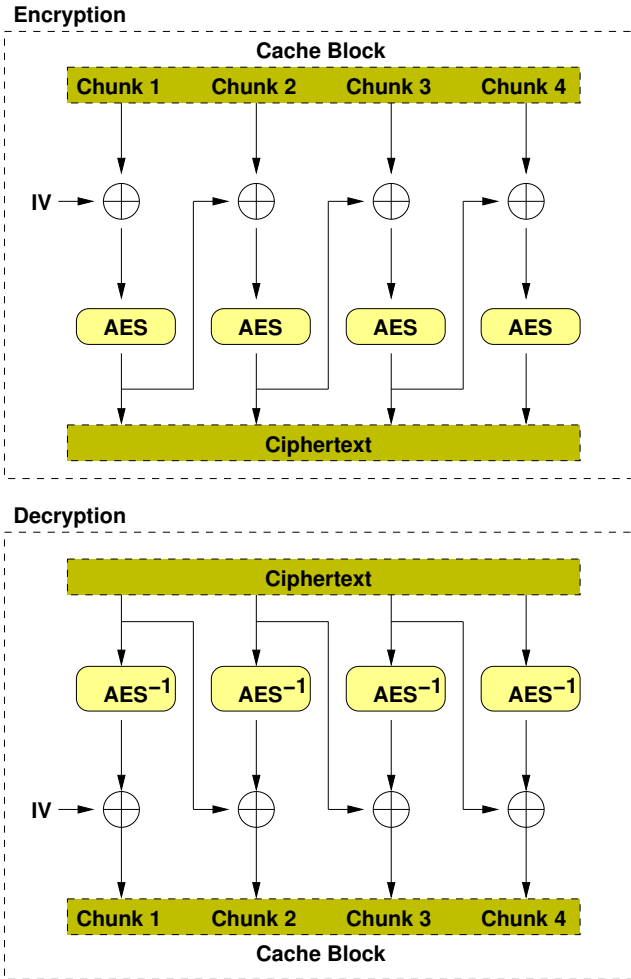


Figure 1. AES in cipher block chaining mode

Throughput To prevent the buffers holding memory traffic from overflowing in the worst case memory access patterns (resulting in stalls), the throughput of the EDU should match or beat the maximum processor-memory bandwidth. Recently, there has been an emphasis in the VLSI community to create AES units with very high throughputs, capable of keeping up with optical data network speeds. Hodjat, *et al.* [8] have designed AES components with throughputs between 3.75 and 8.75 GBytes/s. This is certainly comparable to the memory bandwidth of recent general purpose processors.

Area A high performance EDU should not consume too much die area. Indeed, modern, high performance AES units are being shrunk down to very reasonable sizes. For example, Hodjat, *et al.* [7] have design a high performance AES unit in 0.18 μ process technology that takes only 0.79 mm² of space. Considering that the total die area of a Pentium 4 Northwood processor is 145 mm² and it was devel-

oped in a 0.13 μ process, an EDU would take less than 0.5% of the die area. As another example, the AES unit with a throughput of 3.75 GBytes/s designed by Hodjat, *et al.* [8] has a gate count of approximately 200,000 (a small number considering the Pentium 4 Prescott has 125 million transistors).

Latency Suh, *et al.* [18] estimates that an AES operation implemented in hardware on a modern processor in the 0.13 μ process will take 20-64 ns (20-64 cycles if the processor is clocked at 1 GHz). Kuo, *et al.* [9] and Schaumont, *et al.* [16] indicate that it will take 60-96 ns on 0.18 μ technology. The key insight to take away from this is that the encryption/decryption latency is on the critical path and will add significantly to the time it takes to carry out memory operations. While the encryption latency is the worst, it is also the easiest to hide. Data evicted from the cache can be buffered (in a write buffer) to await encryption—the processor does not need to stall until the operation completes before proceeding. But loads, which require decryption, are typically followed by dependent instructions that consume the loaded data. Thus, the latency of decryption is more critical. Despite the parallelization of the AES operations in decryption, latency is still bounded by the latency of a single AES operation, of which the latency is on the order of a memory operation. The next section will address this problem and offer a solution.

2.2 OTP Encryption

To remedy the latency problem associated with decrypting data brought into the processor from memory, a different mode of AES encryption is used—one time pad (OTP), also called counter mode [18, 20]. This mode of encryption has not been found to be any weaker than CBC mode encryption. The goal is to hide the decryption latency under the memory load latency by breaking the dependency between the ciphertext data and the bulk of the decryption routine. In OTP mode the bulk of the decryption routine does not require the ciphertext; it can run in parallel with the memory load. Only at the last step is the ciphertext needed.

OTP encryption works by encrypting the address of the data and a unique sequence number. Since this input is usually not 128 bits wide (the block length that AES typically operates on), a pad of random bits is also included in the input to each AES unit. An encrypted bit stream that is the length of a cache block is desired, so multiple AES units are used. The input to each AES unit is (*sequence number, address, pad, i*), where *i* is the number of the AES unit. The result of the encryption (called a pad), which is the size of a cache block, is then XORed with the plaintext to yield the ciphertext. The ciphertext can then be stored in memory and the sequence number can be stored in a special region of

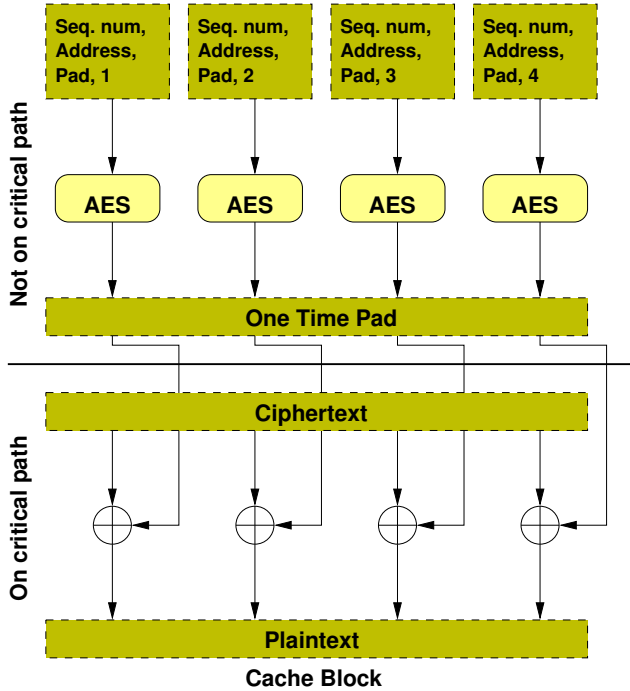


Figure 2. AES OTP decryption

memory indexed by the block address of the corresponding ciphertext. A new sequence number is generated for each cache block encryption typically by incrementing the last used sequence number; the initial sequence number is chosen randomly. The length of the sequence number should be small enough to store compactly but long enough to avoid frequent wrap-around. A 64-bit sequence number is typically used.

OTP decryption (see Figure 2) works by first generating the same pad that was generated in the encryption process (i.e. encrypting the *(sequence number, address, pad, i)* that was associated with the data during the store operation). Then the pad is XORed with the ciphertext to produce the plaintext. Note that the costliest part of the decryption process (the AES operation on the address and sequence number) happens without the need for the ciphertext; it happens in parallel with the loading of the ciphertext from memory. Assuming the latency of the decryption is less than the latency of the load, the only additional latency on the critical path that decryption introduces is the latency of the XOR operation, which is negligible.

The reason for using sequence numbers in the encryption and decryption process, as opposed to just memory addresses, is to prevent a type of known-plaintext attack from occurring. Such an attack works as follows: if an attacker knows the plaintext data value to be stored at a particular memory location, he can XOR the ciphertext stored there with the known plaintext to obtain the pad associated

with that memory location. When subsequent ciphertext is stored at that memory location, he can discover the plaintext by XORing the ciphertext with the pad found from the previous step. Using sequence numbers ensures that the same pad is never used twice, preventing this kind of attack.

It is safe for the sequence numbers to be stored unencrypted because an attacker can not generate a pad without the encryption key, which is kept secret. If an attacker changes a sequence number in memory, the processor will recognize the change when a memory verification is done (more on this in the section on execution integrity, Section 3).

For OTP encryption to be effective in eliminating the decryption latency, the sequence number associated with each cache line must be immediately available to the pad generation hardware. The simplest solution is to cache recently used sequence numbers. To avoid polluting the common on-chip caches with sequence numbers, a special sequence number cache may be employed. While this eliminates decryption latency in many cases, misses in the sequence number cache result in added latency because the sequence number must be fetched from main memory before pad generation can begin. It also places additional area demands on the processor. To be effective, the sequence number cache must be sufficiently large; and Shi, *et al.* [17] note that the hit rate of sequence number caches do not scale well with cache size (probably because they contain multiple large working sets). The services of a sequence number cache are only needed when the normal caches have failed (when the processor has to go to main memory). Thus, it is hard to exploit temporal locality in sequence numbers. In the next section we will look at proposed techniques that eliminate the need for sequence number caches, while eliminating decryption latency in most cases.

2.3 Improved OTP Encryption

Shi, *et al.* [17] tackle the problem of further reducing decryption latency by taking the fetching of the sequence number off the critical path. They propose a series of techniques to predict sequence numbers and pre-compute OTP pads such that the correct OTP pad is available by the time the ciphertext data comes in from main memory.

First, a root sequence number is associated with every page of memory. The need for this will be evident later. When a page is mapped into physical memory for the first time, it is assigned a (random) root sequence number, which is recorded in a reserved column in the TLB. The initial sequence number of every cache block in that page is the root sequence number. On every dirty cache block eviction, the sequence number associated with that block address is incremented. On memory loads, the data is loaded along with the sequence number stored in memory. It is not a security

risk for these sequence numbers to be exposed since, without the key, an attacker cannot encrypt the sequence number and address to produce the same pad that the processor can.

Because the AES unit can be pipelined, it is possible to pre-compute multiple OTP pads under a single memory load. While a memory load is in flight, multiple sequence number predictions can be made, and multiple pads can be computed. The number of pads that can be computed before a memory load completes is a function of the memory load latency, the decryption latency, and the degree of pipelining possible and is referred to as the prediction depth. When the load is finally complete, the correct sequence number (loaded from memory along with the cache block), is compared to the predicted sequence numbers. If a match is found, the pad generated from the matching sequence number is used—that is, it is XORed with the ciphertext to produce the plaintext. If a match is not found, the processor will have to continue to wait for the loaded data as the AES unit will have to generate a pad based on the correct sequence number.

In the simplest prediction scheme, the root sequence number and the next few consecutive numbers are predicted. This simple prediction scheme works well for infrequently updated data because the sequence numbers do not grow very far beyond the root sequence numbers.

The prediction accuracy for frequently updated data can be improved by employing what is called adaptive prediction. It builds on the simple prediction scheme by first identifying frequently updated data and then adapting by resetting the root sequence number. It identifies frequently updated data by keeping a prediction history (represented by a bit vector) with each line of page meta-data kept in the TLB. When the misprediction rate for a page exceeds a threshold, a new (pseudo-random) root sequence number is associated with that page and any subsequent blocks written out to memory will start with the new sequence number. The authors of [17] found adaptive prediction to work surprisingly well—with a prediction accuracy of around 80% (1.4 times the hit rate of a large sequence number cache).

Another way to improve prediction accuracy is to increase the effective prediction depth (the number of predictions made and pads generated under the memory latency). Two-level prediction does this by adding a few bits to each cache line that encode the range the sequence number predictor should predict in (based on the previous known sequence number associated with that cache line). Predictions are made starting at a sequence number obtained by summing the root sequence number with the product of the range number and the prediction depth. For example, if the prediction depth is 8 and two bits are added to each cache line, then if the root sequence number of the page is 1000 and if the range number is 2 (0–3 inclusive are possible), then the following sequence number predictions will be made: 1016,

1017, 1018, 1019, 1020, 1021, 1022, 1023. The effective prediction depth is quadrupled with only two bits added to each cache line. The authors of [17] found two-level prediction to have an average prediction accuracy of 96%.

To even further improve the prediction accuracy, context based prediction is used. An LOR (latest offset register) is added that records the difference in the most recent memory access between the root sequence number and the correct sequence number. A context based predictor makes two sets of predictions. The first set of predictions comes from the regular adaptive prediction rules (consecutive sequence numbers after the root sequence number). The second set of predictions are a consecutive set centered around the sum of the root sequence number and the LOR. Context based prediction has been shown to be nearly 100% accurate [17]. Making more predictions burdens the decryption unit, but only having to add a single register (LOR) to the architecture to reap the benefits of accurate sequence number prediction makes it worthwhile.

3 Execution Integrity

While memory encryption is useful for ensuring the privacy of program execution, it does little to prevent tampering with program execution. In fact, if a process is vulnerable to tampering, the privacy of its execution is at stake even if memory is encrypted because an attacker may alter a running program in such a way that it reveals its secrets. For example, a process that has been tampered with may be instructed to copy all its data from an encrypted region of memory to an unencrypted region, defeating its privacy. Thus, integrity is a requirement for privacy. Note that the converse is not true—privacy is not a requirement for integrity. The executing code and the data belonging to a process may be protected against tampering (tamper-evident) but not be hidden from public view. The asymmetric duality between integrity and privacy has prompted some researchers to classify process security into only two categories: integrity without privacy, and integrity with privacy. Process privacy cannot be guaranteed without integrity.

Integrity can mean different things depending on the context. Some might claim that a system is tamper-proof (or tamper-resistant). Tamper-proof means that it is impossible for tampering to take place. A tamper-proof processor is one in which computation cannot be subverted—the intended execution and only the intended execution is carried out. However, because processors are not entirely self sufficient and hardened against all possible physical threats, they are not tamper-proof. A physical attack may tamper with execution by halting it, restarting it, or introducing errors into it (perhaps by cutting power to the processor or bombarding with radiation). A more practical integrity goal of secure processors is tamper-evidence. In a tamper-evident system,

tampering is not prevented, but it is clear if tampering has taken place. Computation in which tampering has been detected is distrusted, and the results of the computation are prevented from being used.

Tamper-evident execution is a quality that can be provided by cryptographic keyed hash functions. In our model, since everything outside the processor boundary is distrusted, all data read in from memory must undergo an integrity check that determines if the data has been tampered with or not. This is accomplished by signing (i.e., hashing) all data that leaves the on-chip caches bound for main memory with a key only known by the processor (usually a symmetric key for performance reasons). When data is loading into the processor from main memory, the integrity check routine computes a keyed hash of the data with the same key and compares the resulting signature with the signature taken during the store. If the hash values match, the data can be trusted; if they do not match, the data has either been maliciously tampered with or it has been corrupted by a hardware error (either way, the data is not trustworthy and is unusable). This assertion relies on the collision resistance of the hash function. A hash function is collision resistant if an attacker cannot produce data that hashes to the same hash value even if the attacker is given the data that produced the hash.

There is a fundamental trade-off between the latency of hash computation, the degree of integrity that hashing provides, and the storage requirements of the hashes. Hashing large chunks of data results in a higher latency for each write to memory. For example, computing a hash over a whole page of memory each time a cache block from that page is written out to main memory is a costly proposition. On the other hand, hashing small chunks of data results in more hashes that must be kept around. The size of a hash value is directly related to the integrity it provides; longer signatures are harder to forge than smaller ones. So, if we optimize for performance and strong integrity, the storage requirement becomes burdensome and it becomes infeasible to store all hashes on-chip. Typically, hashes are 128 bits wide and are taken over data the size of cache blocks.

The obvious solution to the storage problem of hashes is to store hashes in main memory. Because only the processor has the key to compute valid hashes, spoofing attacks are prevented even if attackers are able to alter data and their hashes in memory; an attacker cannot introduce arbitrary data of his choosing on the memory bus that will be trusted by the processor because he cannot compute a valid keyed hash of the arbitrary data without the processor's key.

Another kind of attack is made possible, though, by storing hashes in an insecure manner in main memory: a replay attack. In a replay attack on a secure processor, the attacker stores valid data (though signed and possibly encrypted) and later feeds this data back to the processor (over the memory

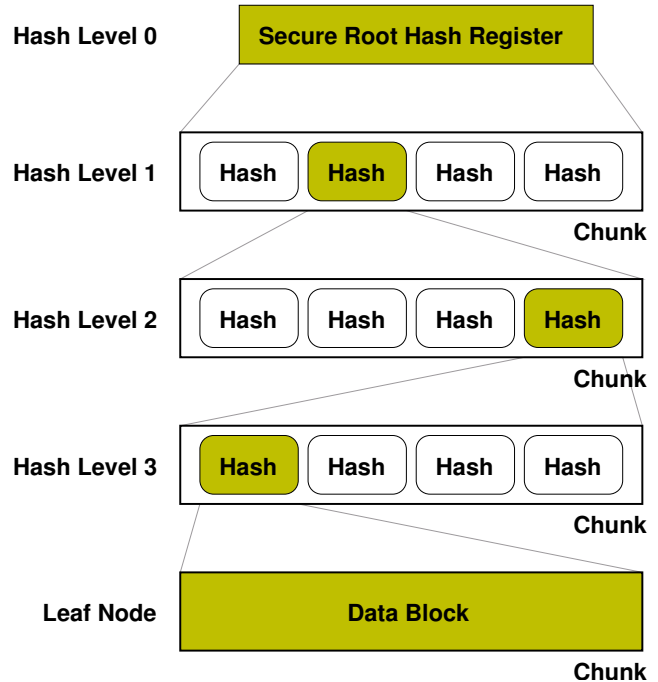


Figure 3. Cached Hash Tree

bus). The stale replayed data is then accepted as valid, current data by the processor. To prevent replay attacks, the integrity of hashes (which attest to the integrity of data) needs to be maintained.

3.1 Cached Hash Trees

One way to accomplish this is to build a hash tree [13]. A hash tree is made up of leaves containing data and intermediate levels of hashes all the way up to a root hash [2]. Each hash either signs a chunk of data (a leaf) or a small group of hashes. The root hash can be maintained in a register on-chip, guarding it against attack. If any data is modified off-chip, that data's hash will tell of the change. If the hash is modified as well in such a way that it is consistent with the changed data (as in a replay attack), the next hash up in the hierarchy will tell of that change. The chain of hashes continues all the way up to the root hash, which can never be compromised because it never leaves the chip. Figure 3 shows one branch of such a tree, tracing a chain of integrity from data up to the root hash.

Operations on a hash tree rooted in the processor but stored off-chip are potentially expensive. To keep the tree in a consistent state, each store operation would result in multiple hash operations. The number of hashes that must be taken is $O(\log(N))$ of the memory footprint. The radix of the log is the number of nodes over which each hash is taken. Similarly, each load instruction would result in $O(\log(N))$ hash operations. So, for example, if an appli-

cation uses 256 MB of memory, the cache block size is 64 bytes, and the hash size is 128 bits, the tree would have 11 levels (11 hashes to compute and check) and the intermediate hash nodes would take up about 85 MB of memory. The frequency of loads and stores and relative latency of cryptographic hash operations makes the scheme impractical.

Gassend, *et al.* [6] conceived of a technique to remedy the latency issue of maintaining a hash tree in memory. They leverage the on-chip caches to make the common case fast. The key insight is that the entire tree does not need to be kept in a consistent state all the time. Only the portions of the tree that reside in memory need to be consistent because all nodes (hashes or data) already on-chip have already been verified. When a chunk is read into the cache from memory, a hash is taken over it and compared against the parent hash. If the parent hash is in the cache, that is as far as we need to check; we do not need to check the parent of that hash because the hash value has not been tampered with since it has been residing on-chip. If the parent hash of the chunk read in is not in the cache it is brought in and verified in the same way. The process is recursive until a hash is found in the cache (or the root hash is reached). When data is written, if the data is found in the cache, it can be written without modifying any hashes. If the data is not found in the cache, a write-allocate policy invokes a mechanism to bring the data to be written into the cache. When a dirty cache block is evicted, a hash is computed over it and, atomically, the chunk is written to memory while the parent chunk's hash is written using the write mechanism just described. In the common case, the parent node will be in the cache and only a single write to the parent node will be all that is necessary to achieve sufficient hash tree consistency.

3.2 Log Hashes

The performance of integrity verification can be significantly improved by relaxing the constraint that integrity must be maintained at all times. In most secure applications, the integrity of all computation only needs to be ensured at one point: at the end of computation before the results are “presented”. The cached hash tree technique verifies integrity at each memory read. Postponing verification to the end of a computation (and then verifying a series of memory events) delays the detection of an integrity violation; but as long as an integrity violation is detected before the results of the computation are consumed by a user or another application, no harm can be done. It is important to note that verification must still be done over all memory instructions in the computation, even though the verification check is performed later.

The log hash technique [18] was designed to take advantage of the relaxed constraint on when verification must occur. But to understand the log hash technique it is necessary to first understand a unique cryptographic function called an

incremental multiset hash function [3]. A multiset is just an unordered set in which duplicate items are allowed. All sets are multisets, but not all multisets are sets because sets do not allow duplicates. An incremental hash function [1] is one in which the change to the hash required by a change to the string over which the hash was taken is proportional to the amount of change in the string. A small change in the string does not require a new hash to be recomputed over the whole string; rather, the hash is incrementally updated using the changed part of the string. This makes small updates very efficient. An incremental multiset hash function has the property that an incremental hash H over a sequence S with key K is equivalent to an incremental hash H' over a sequence S' with key K if S' contains exactly the same elements as S does with the same multiplicities (the multiplicity of an item is the number of times it is duplicated in a multiset). For example, the incremental multiset hash of items A,B,B is the same as the hash of items B,A,B.

Log hashing has three phases: initialization, run-time hashing, and verification. The following components are added to the processor: a register to hold an incremental hash value called the READHASH, a register to hold an incremental hash value called the WRITEHASH, and a register to hold a counter. Additionally, a portion of main memory is set aside to hold counter values—a counter value is associated with each block of data. Counter values (which are usually 32 bits) use a relatively small fraction of memory since only one counter value is associated with each cache block. For example, if the cache block size is 64 bytes and the counter is 32 bits wide, the amount of memory required to hold the counters is only 6.25% of the application memory footprint.

Initialization During initialization, every chunk belonging to the process whose integrity must be later verified is hashed (one at a time) along with the current counter value and the memory address, and the hash is stored in memory along with the counter value. The hashes are added to WRITEHASH and the counter is incremented with each chunk written.

Run-time During run-time, data read in from memory is hashed (along with the associated address and counter value) and the hashes are added to READHASH. If the counter value associated with the chunk is equal to the current counter value, then the current counter is incremented. Any chunks evicted from the on-chip caches and sent to memory are hashed along with the current counter value and the memory address, and the hashes are added to WRITEHASH. Figure 4 depicts log hashing's run-time behavior.

Verification When verification is requested, all chunks that belong to the process (whose integrity must be veri-

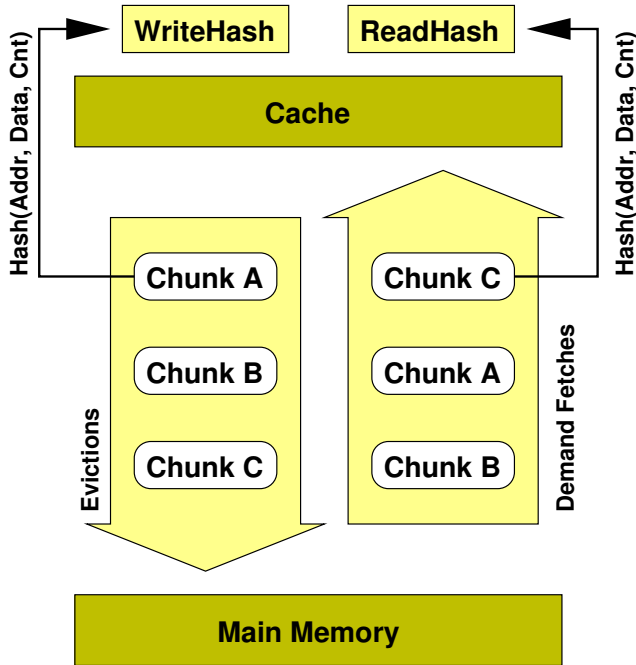


Figure 4. Log Hashing: The hashes of all data chunks written to memory are incrementally added to the WriteHash register; the hashes of all data chunks read from memory are incrementally added to the ReadHash register.

fied) that are not in the cache are read in from memory and their hashes (of data, counter value, and address) are computed and added to READHASH. At this point, if none of the memory traffic has been tampered with, everything incrementally hashed into WRITEHASH has been incrementally hashed into READHASH the same number of times (though, perhaps in a different order)—the multiset of writes is equivalent to the multiset of reads and, thus, their multiset incremental hash values are equivalent. If WRITEHASH is equal to READHASH, integrity has been maintained throughout all the previous computations. Otherwise, integrity has been compromised and the result of the computation should be distrusted.

To see why it is true that, if integrity has been maintained, READHASH should equal WRITEHASH, consider the possible ways in which memory might be used. If a value is part of the memory contents of the application but is never read or written during execution, its hash will have been added to WRITEHASH exactly once during initialization and added to READHASH exactly once during verification. Every value brought into the cache for the first time during run-time mirrors the write of that value during initialization. Every time a value is evicted from the cache (and its hash added to WRITEHASH), it is guaranteed that value will ei-

ther be read in again during run-time (and its hash added to READHASH then) or will be read in during verification (and its hash added to READHASH then). If data is ever brought into the cache and then evicted without ever being modified, WRITEHASH is still updated and the new counter value is stored in memory. In this way, if the data is brought back into the cache on a subsequent read, the hash added to READHASH will mirror the hash added to WRITEHASH on the eviction, and the symmetry of memory traffic will be maintained.

An attacker cannot replay old data without it being evident because that would change the balance of reads versus writes of the replayed data (the data would be read more times than written) and READHASH would not match WRITEHASH. The sequence numbers prevent an attack where the attacker predicts a future read and presents it to the processor on a load. Then the attacker might replay the stale data at a later time, effectively re-ordering the data sent to the processor on two read operations.

The performance of log hashing is very good. Because integrity is not verified at every data read, the hashing function can be taken off the critical path; as soon as data reaches the cache from main memory, it can be consumed while the incremental hash function is done in parallel. As long as the hash unit can match the required memory bandwidth and verification points are infrequent, no performance penalty will have to be paid for memory integrity except during the initialization and verification phases, which run for only a fraction of the total execution time of most applications. The on-chip storage requirement is very small—just a few registers (no special caches need to be built). Since counter values are small compared to cache block size, the extra space they occupy in memory will be relatively small. The main performance cost is in the extra memory bandwidth required to transport counter values. This is still considerably less additional memory bandwidth than is required to implement cached hash trees. Also log hashing does not pollute the cache as cached hash trees do. The authors of [18] show that log hashing reduces IPC by less than 5% on average (15% in the worst case), while cached hash trees reduce IPC by 20–30% on average (50% in the worst case).

4 Summary of Architectures

Summaries of the major secure architectures are given here. All of these proposed architectures have memory privacy and integrity at their core. Their designs and contributions are briefly discussed. To conclude this section, the shortcomings of each architecture are presented and briefly analyzed. The architectures introduced here include XOM (eXecute Only Memory), SP (Secret-Protected), AEGIS (Architectural EnGINes for Information Security), and SENSS (Security ENhancement to SMP Sys-

tems). For a more detailed description and analysis of any of these architectures, please refer to the original work.

4.1 XOM

XOM [11] was the first of its kind. Most of the recent research in secure computing architecture is inspired by XOM. XOM trumpeted the idea that software alone was not sufficient to provide protection, that some security infrastructure is needed at the hardware level—a part of the very architecture. XOM drew a perimeter of trust around the CPU and grew trust out from there. It promised privacy and integrity in the face of the most extreme types of attack including a compromised operating system or a physical attack on memory or the system buses.

A program to be run on XOM is encrypted with a unique symmetric key that is, in turn, encrypted with the public key of the XOM processor. This key and its private key counterpart are embedded in the processor when it is manufactured and cannot be changed. When the program is run on XOM, the processor first decrypts the symmetric key (session key) and uses the symmetric key to decrypt the running program. Caches and registers are tagged with XOM IDs that correspond to secure processes executing in the machine. A table associates XOM IDs with XOM session keys. All data written to memory is encrypted and hashed using the session key for the running process and all data read from memory is decrypted and its hash is verified. The privacy and integrity techniques are simple, but naïve. Consequently, the performance of XOM is dismal.

4.2 SP

Created by researchers at Princeton, the secret-protected architecture [10] (formerly called virtual secure coprocessor [12]), shares the same foundation as XOM and the other secure computing architectures; the core of the SP architecture is memory encryption and integrity verification. SP uses the simplest form of symmetric encryption, AES-CBC, to provide execution privacy. SP uses a naïve hashing technique to provide execution integrity; the details of the mechanism are not given, but it is inferred that a large fraction of memory is set aside for hashes.

SP does not focus on the mechanisms of execution privacy and integrity, but on how to keep the keys used for these mechanisms secure. The novel idea introduced by this architecture is the decoupling of the secret from the device. Associating a user with a particular device via the device secret (i.e., processor private key) is incorrect because mutually distrusting users may share the same device or the ownership of the device may be transferred to another user at some point. The SP architecture allows users to securely access their keys from any location. All the user's keys for var-

ious secure applications (e.g., PGP key, encrypted file system key, banking PIN) are organized in a hierarchical fashion where each key is encrypted by a higher key; the top of the keychain is the user master key. The encrypted keychain can be safely transmitted over unsafe networks because it is encrypted with the user master key which is only known to the user. On an SP-enabled device, the user presents some credentials, like a password or a biometric scan, and the system hashes this data to obtain the user master key, which can then be used to access keys in the user keychain. All of this computation is protected within the confines of the processor package.

4.3 AEGIS

AEGIS [19] is a project at MIT that aims to build a secure processor that is secure against physical attacks and suited for secure distributed computing. The goal is to be able to trust execution done by various remote processors.

AEGIS allows four modes of operation: standard (no guarantees of integrity or privacy), tamper evident (integrity guaranteed but not privacy), private tamper evident (integrity and privacy guaranteed) and suspended secure processing. The first three modes of operation recognize the fact that a software developer may not require full security all the time. If privacy is not needed but the integrity of execution needs to be verified, the software developer may choose to have his application run in tamper evident mode, eliminating the overhead incurred to ensure privacy. The fourth mode of operation (suspended secure processing) allows for transitions into standard mode from secure modes, giving the software developer the flexibility to interrupt the secure execution flow to execute insecure or untrusted (perhaps third-party) code. All security meta-data is safely stored away during the period of suspended secure processing in such a way that inspection and tampering are virtually impossible.

AEGIS uses a simple form of OTP encryption to ensure that memory contents are private. It does not incorporate any prediction or precomputation methods, though such methods would greatly improve the performance. It is not clear which integrity technique AEGIS currently utilizes. Both cached hash trees and log hashing were introduced by the authors of AEGIS.

AEGIS includes a security kernel that runs at a higher privilege than the rest of the kernel and handles security critical operations like memory management and interrupt handling. The operation of the security kernel is enabled by the same execution privacy and integrity methods available to all running applications. To prevent an unauthorized security kernel from being introduced into the system without detection, computation results may be signed by the processor with a hash of a program's output concatenated with a hash of the program's input concatenated with a hash taken

over the security kernel code concatenated with a hash over the application code. Hashes of well-known security kernels and user applications are published so that users may then verify the signature made by a particular processor (with a public key/private key pair) of the execution of a particular application (with particular inputs) facilitated by a particular security kernel.

The researchers who are developing AEGIS proposed the use of what they call Physical Random Functions (sometimes called PUFs) to protect processor secrets [19, 4, 5]. Processor secrets include the keys used in OTP generation and hashing. PUFs take advantage of the slight differences that exist between individual processing cores. These differences are inherent in the silicon chip manufacturing process and are manifested in slightly different timing characteristics of the circuits. Racing two signals down two wires of the same length will result in different winners depending on the particular chip. Physical Random Functions work on a challenge/response system where a given challenge (just a bit sequence) receives a different response (another sequence of bits) from each core (with identical design). Because the environment (heat, electrical fields in close proximity, etc) changes the timing delay characteristics of circuits to some degree, BCH error correcting codes are used by PUFs to eliminate this noise, which is much smaller than the variance between two chips.

4.4 SENSS

SENSS [21] extends the security measures already proposed for uniprocessors with new security measures for multiprocessor systems. The memory encryption and authentication scheme remains essentially unchanged in the multiprocessor domain. The new communication channel that must be guarded is the shared bus that connects the caches of each processing node. The researchers who designed SENSS focus on this problem.

To bootstrap the execution of a multi-threaded application on a secure multiprocessor system, the application's code is first signed by its distributor before being shipped to the client. It's signed with all the public keys of all the processors in the multiprocessor system on which it is intended to run. That way, the software vendor is given complete control over which processors to allow his program to run on. When cache to cache data transfers need to take place during execution, the data is encrypted by the sender and decrypted by the receiver.

Many of the techniques to protect memory from attack are used to protect the communication channel between processor caches in the multiprocessor environment. A combination of cipher block chaining (CBC) mode encryption and one time pad (OTP) encryption is used to improve the performance of the data transfers; it works as follows. The sender and receiver share a mask (or initial vector to begin

with) and a symmetric session key (that is unique to each program execution). The plain-text is XORed with the mask by the sender and sent on its way to the receiver. The sender then computes a new mask by encrypting the cipher-text just sent. The receiver XORs the cipher-text it receives from the sender with the mask it has and passes the resulting plain-text on to the receiving thread. The receiver then computes a new mask in the same way as the sender, by encrypting the latest cipher-text message. The latency problem of back-to-back transmission is handled by having a vector of masks (instead of a single mask) that are used in a round robin fashion.

SENSS uses the same simple OTP encryption and hash tree schemes proposed by the authors of AEGIS. A standard cache coherency protocol (like write update or write invalidate) ensures that cached hashes remain consistent between caches located on separate processors.

4.5 Critical Analysis of Current Architectures

XOM Despite its novelty, XOM falls short in a couple of areas. First, performance is not a consideration. Most glaring, memory encryption and verification are on the critical path and are not optimized. Second, while XOM is immune to most kinds of attacks, like spoofing attacks (in which an attacker tries to pass off invalid data as valid data), XOM is still vulnerable to replay attacks (an attacker replays old stale data when newer data is requested).

SP The SP architecture has some very attractive features regarding key management. However, it also has these weaknesses:

- Tampering with the secure I/O channel is not considered. The user presents his security credentials and checks the security status of the device over this channel.
- Only the software module that carries out cryptographic routines on behalf of user applications (called the TSM) runs in Concealed Execution Mode (the mode of operation that provides privacy and integrity). It would be ideal if all program execution was protected, not just the critical parts that deal with user secrets.
- The researchers claim that the small code size of the TSM makes it more trustworthy because it is easier to check for bugs. However, they also stress the need for static compilation of library dependencies (to guard against library corruption as a possible point of attack), bloating the code size and making the TSM harder to verify by simple inspection.

AEGIS Despite all its merits, AEGIS has a couple of shortcomings: Dynamic memory verification is accomplished using cached hash trees, which, though it is one of the best memory integrity verification techniques, still imposes a significant performance penalty (cached hash trees are explained in Section 3). The other shortcoming of AEGIS is its dependence on a trusted compiler. The compiler must be able to work with different memory regions corresponding to the different protection levels, understand more complex variable scoping rules, manage separate stacks and heaps, and insert mode transitioning instructions in the appropriate places.

SENSS SENSS is a clever architecture that achieves its goal of being a secure multiprocessor architecture, but it has a couple of deficiencies. First, it is not clear what happens if an alarm signal (generated if an integrity check fails) is lost in transit to the root node. Such a signal may be blocked by an attacker who wishes to prevent the user or the rest of the system from discovering a problem. A secure I/O mechanism is needed. But SENSS does not address this or even acknowledge the need for it. Second, SENSS assumes that the cache coherence protocol ensures a global ordering of messages. However, not all cache coherence protocols make such a guarantee (weak consistency). It is not clear how SENSS could be adapted to such a system.

5 Future Work

This section explores areas of potential future work. It poses unanswered questions in the field—questions that researchers are either trying now to answer or questions that are not being addressed but that ought to be addressed.

5.1 Fault Handling

When a process has been tampered with a fault is generated by the security hardware (e.g., the integrity verification unit). Faults must be raised before any damage can result from the fault. For example, an attacker replays stale data from memory to the processor that it uses in a bank transaction computation. It would be hazardous if the process updates the bank’s account database before the integrity violation is caught and another process reads the incorrect data out of the database before the error is corrected. Integrity verification schemes like cached hash trees could prevent this kind of scenario, but more efficient integrity verification algorithms like log hashing rely on the programmer or compiler to insert integrity verification checkpoints at key locations in execution to prevent the spread of corrupt information. Can the insertion of integrity verification checkpoints be automated? Should it be done at run-time or should it be the responsibility of the compiler?

Once a fault has been generated in a timely manner, the next step is to handle the fault. Unfortunately, security faults cannot be handled the same way as other kinds of faults. Normally, an operating system fault handling routine is invoked; however, in our security model, the operating system is untrusted. If an attacker has compromised a process, it is quite possible that the operating system is compromised as well. On a compromised system, the simplest way a secure processor can reliably signal to the user that a violation has occurred is to signal the event over a secure I/O channel—but it may not always be a feasible means to inform the user, or a more automated fault recover mechanism may be required. In summary, future research needs to be done on how to detect security faults in a timely manner without sacrificing performance or placing a burden on the programmer and how to handle security faults when they are detected.

5.2 Secure I/O

Users must be able to securely communicate with secure processors. A user must be able to enter a password without it being exposed to an attacker, even if the attacker has physical access to the machine. Also, users must be able to trust the output of the computer even if the video card, network card, or printer has been tampered with. If nothing is done to secure vulnerable communication channels, the security of the processor will be irrelevant.

Realizing the importance of securing the I/O of the processor, some researchers (particularly Lee, *et al.* [10]) have advocated a secure I/O mechanism, whereby users can be given some kind of assurance that processor results have not been tampered with and can securely authenticate themselves or perform key management operations on the processor. For example, to prevent a keystroke logger from intercepting passwords as they are typed, the authors of [10] propose redirecting keyboard input directly to the processor and encrypting the communication channel with some form of public key encryption. This would require the keyboard and the processor to both contain key pairs, and it is unclear how such a system would be re-keyed.

While it is usually considered peripheral to the field of computer architecture, the issue of securing I/O to the processor is, nevertheless, a critical related issue and is largely an open question.

5.3 Side Channel Attacks

The authors of HIDE [22] investigate ways to prevent side channel (sometimes referred to as out-of-band) attacks on secure processors. In particular, they look at how an attacker can exploit information about the control flow of a program, obtained by monitoring memory access patterns and comparing them against memory access patterns collected from common library routines. They take advantage

of the fact that most code is reused and that executing reused code results in signature control flow patterns that can lead an attacker to discovering the algorithms employed by a supposedly secure program. Or it may lead to the discovery of the data being operated on by those algorithms. The authors of HIDE employ a technique of obfuscating the memory access pattern to prevent this kind of attack. However, the obfuscation comes with a high performance cost. Future work might investigate less costly ways of preventing such attacks. The authors of HIDE make the case that a typical cache is not effective at hiding memory access patterns. Can caching mechanisms designed with access pattern obfuscation in mind be more effective while minimizing performance loss due to the obfuscation? Can dynamically remapping memory locations effectively combat the problem?

Also, several other kinds of side-channel attacks have not been addressed. For example, can an attacker mount an effective attack by observing data memory access patterns instead of instruction memory access patterns? Can code size leak useful information about a program? Can an attacker glean useful information by timing execution on varying inputs?

5.4 Other Issues

How to go about encrypting software for distribution to a specified set of computers only is still an open question and a central one to researchers investigating digital rights management (DRM)—a topic that deals with how to assign to people rights on digital content distribution and use. The software industry is facing a huge problem—how to prevent the proliferation of pirated software in the market. Many have proposed DRM as a solution, with secure processors as the backbone to that solution. The idea is that, if every processor has a public key/private key pair that was sealed in such that no one could ever extract it, then software could be encrypted with the public keys of the computers of paying customers and the protection mechanisms of the secure processor would allow that software to be run but prevent it from being copied. However, it is not clear how a software manufacturer can come to trust certain public keys. If the processor manufacturer maintains a list of factory-installed keys, then the processor could never be re-keyed. Not only is it more likely that an attacker could extract a factory-installed secret key without destroying it versus a dynamically-generated one, but it may be desirable to re-key the system if it is ever sold to someone else.

Finally, while secure processing for multiprocessor systems has been explored [21], shared memory in such systems has not. The issue is as much an operating system one as an architecture one. Without hardware support for memory sharing in a secure multiprocessor system, it would either be impossible or highly inefficient. The alternative so-

lutions involve using message passing for interprocess communication and statically linking shared libraries instead of dynamically linking them. This may be acceptable for some applications, but it would devastate the performance of others.

6 Conclusion

General-purpose secure processors are capable of solving a variety of problems—from helping prevent software piracy to protecting user data from unauthorized access to ensuring trustworthy execution on a potentially compromised system. In this paper we have looked at recent proposals on how to build secure processor to meet these demands. We explored the two most important mechanisms in a secure processor architecture: keeping a process' memory private and making it trustworthy; and we briefly covered some exemplary architectures. Finally, we raised some unanswered questions regarding secure processing architectures.

References

- [1] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. *Lecture Notes in Computer Science*, 839:216–233, 1994.
- [2] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.
- [3] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Proceedings of the 9th International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt 2003)*, 2003.
- [4] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled physical random functions. In *Proceedings of the 18th Annual Computer Security Conference*, Dec. 2002.
- [5] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Nov. 2002.
- [6] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The 9th International Symposium on High Performance Computer Architecture*, 2003.
- [7] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede. A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18- μ CMOS technology. In *GLSVLSI '05: Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, pages 60–63, New York, NY, USA, 2005. ACM Press.
- [8] A. Hodjat and I. Verbauwhede. Minimum area cost for a 30 to 70 gbits/s AES processor. In *IEEE Computer Annual Symposium on VLSI*, pages 83–88, 2004.
- [9] H. Kuo and I. M. Verbauwhede. Architectural optimization for a 1.82 gb/s VLSI implementation of the AES Rijndael algorithm. In *Cryptographic Hardware and Embedded Systems 2001*, 2001.
- [10] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society.

- [11] D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November 2000.
- [12] J. P. McGregor and R. B. Lee. Protecting cryptographic keys and computations via virtual secure coprocessing. In *Proceedings of the Workshop on Architectural Support for Security and Antivirus (WASSA) held in conjunction with ASPLOS-XI*, 2004.
- [13] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.
- [14] National Bureau of Standards. *Data Encryption Standard*. FIPS Publication 46. NTIS, Apr. 1977.
- [15] National Institute of Standards and Technology. *Advanced Encryption Standard*. FIPS Publication 197. NTIS, Nov. 2001.
- [16] P. R. Schaumont, H. Kuo, and I. M. Verbauwhede. Unlocking the design secrets of a 2.29 gb/s Rijndael processor. In *Design Automation Conference 2002*, June 2002.
- [17] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 14–24, Washington, DC, USA, 2005. IEEE Computer Society.
- [18] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339, Washington, DC, USA, 2003. IEEE Computer Society.
- [19] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. *SIGARCH Comput. Archit. News*, 33(2):25–36, 2005.
- [20] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 351, Washington, DC, USA, 2003. IEEE Computer Society.
- [21] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENS: Security enhancement to symmetric shared memory multiprocessors. In *HPCA11*. IEEE Computer Society, 2005.
- [22] X. Zhuang, T. Zhang, and S. Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 72–84, New York, NY, USA, 2004. ACM Press.