The AEGIS Processor Architecture for Tamper-Evident and Tamper-Resistant Processing

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, Srinivas Devadas
MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
{suh,declarke,gassend,marten,devadas}@mit.edu

Abstract

We describe the architecture of the AEGIS processor which can be used to build computing systems secure against both physical and software attacks. AEGIS assumes that the operating system and all components external to it, such as memory, are untrusted. AEGIS provides tamper-evident, authenticated environments in which any physical or software tampering by the adversary is guaranteed to be detected, and private and authenticated, tamper-resistant environments where additionally the adversary is unable to obtain any information about software or data by tampering with, or otherwise observing, system operation. AEGIS enables many applications, such as commercial grid computing, software licensing, and digital rights management.

We present a new encryption/decryption method that successfully hides a significant portion of encryption/decryption latency, in comparison to a conventional direct encryption scheme. Efficient memory encryption and integrity verification enable the implementation of a secure computing system with the only trusted component being a single-chip AEGIS CPU.

Detailed simulation results indicate that the performance overhead of security mechanisms in AEGIS is reasonable.

1 Introduction and Motivation

It is becoming common to use a multitude of computing devices that are highly interconnected to access public as well as private or sensitive data. On the one hand, users desire open systems for ease-of-use and interoperability, but on the other hand, they require privacy mechanisms that restrict access to sensitive data, and authentication mechanisms that ensure data integrity. With the proliferation and increasing usage of embedded, portable and wearable devices, in addition to protecting against attacks from malignant software, we also have to be concerned with physical attacks that corrupt data, discover private data or violate copy-protection, as well as combinations of physical and software attacks.

Given these trends, computing systems have to achieve several goals in order to be secure. Systems should provide *tamper-evident environments* where software processes can run in an authenticated environment, such that any physical tampering or software tampering by an adversary is guaranteed to be detected. In *private and authenticated tamper-resistant environments*[1], an additional requirement is that an adversary should be unable to obtain any information about software and data within the environment by tampering with, or otherwise observing, system operation. Ideally, a computing platform should provide a multiplicity of private and authenticated environments wherein each process (or each user) is protected from all other users and potential adversaries.

In this paper we describe the AEGIS processor architecture, which provides multiple mistrusting processes with environments such as those described above, assuming an untrusted operating system and untrusted external memory. We believe that these environments will enable a new set of applications. For example, grid computing is a popular way of solving computationally-hard problems (e.g., `SETI@home`, `distributed.net`) in a distributed manner on a huge number of machines with different volunteer owners connected via the Internet. However, maintaining reliability in the presence of malicious volunteers requires significant additional computation to check the results produced by volunteers. The tamper-evident and tamper-resistant environments provided by AEGIS can enable commercial grid computing on multitasking server farms, where computation power can be sold with the guarantee of a compute environment that processes data correctly and privately.

[1] In the remainder of this paper, we may refer to these environments as private tamper-resistant environments for brevity.

Private tamper-resistant environments can also enable applications where a compute server is used as a trusted third party. For example, a proprietary algorithm owned by party A can be applied to an proprietary instance of a problem owned by party B to produce a certifiable result, ensuring that no information regarding either the algorithm or the problem instance is leaked, and ensuring that the data was processed by the code correctly.[2] Private tamper-resistant environments also enable the copy-protection of software and media content in a wide range of computing systems in a manner that is resistant to software or physical attacks. This will enable strong forms of software licensing and intellectual property protection on portable as well as desktop computing systems.

The key architectural mechanisms required in AEGIS are memory integrity verification, encryption/decryption of off-chip memory accesses and a secure context manager. In this paper, we describe how these mechanisms are integrated into the AEGIS microarchitecture, and evaluate the performance overheads of these mechanisms. We also present a new encryption/decryption method that generates one-time pads using the AES encryption algorithm, successfully hiding a significant portion of encryption/decryption latency, and improving performance relative to a conventional, direct encryption scheme. A companion paper (also submitted to this conference) describes new and efficient integrity verification mechanisms. Detailed simulation results indicate that the performance overhead of security mechanisms in AEGIS is reasonable. These mechanisms therefore enable the implementation of a secure computing system with the only trusted component being a single-chip AEGIS CPU.

We present our security model in Section 2. The AEGIS processor architecture is described in Section 3. We describe how the architecture can be used for a certified execution application and a simple Digital Rights Management (DRM) application in Section 4. The integration of integrity verification techniques described in [18] is the subject of Section 5. A new encryption/decryption scheme is presented in Section 6. Simulation experiments to evaluate the performance overheads of the various mechanisms are presented in Section 7. Related work is described in Section 8, and we conclude the paper in Section 9.

[2]By correctly, we do not mean that the code does not have any bugs, but that the code was not tampered with and was correctly executed.

2 Secure Computing Model

We consider systems that are built around a processing subsystem with external memory and peripherals. Figure 1 illustrates the model. The processor is assumed to be trusted and protected from physical attack, so that its internal state cannot be tampered with or observed directly by physical means. The processor can contain secret information that identifies it and allows it to communicate securely with the outside world. This information could be the secret part of a public key pair protected by a tamper-sensing environment [17], or a Physical Random Function [6].

In the model of Figure 1, external memory and peripherals are assumed to be untrusted. They may be observed and tampered with at will by an adversary. The operating system (OS) is also untrusted. Software attacks by the operating system or from other malicious software are therefore possible. The processor is used in a multitasking environment, which uses virtual memory, and runs mutually mistrusting processes within tamper-evident or private tamper-resistant environments. Given that the OS is untrusted, the processor needs support for secure context switching.

The adversary can attack off-chip memory, and the processor needs to check that it behaves like valid memory. *Memory behaves like valid memory if the value the processor loads from a particular address is the most recent value that it has stored to that address.* If the contents of the off-chip memory have been altered by an adversary, the memory may not behave correctly (like valid memory). We therefore require memory integrity verification [18].

In the case of private tamper-resistant environments, we have to encrypt data values stored in off-chip memory.

Thus, the three main processor mechanisms required for security are memory integrity verification, secure context management, and memory encryption.

3 The AEGIS Processor Architecture

3.1 Tamper-Evident Architecture

This section describes a processor architecture that a tamper-evident execution environment can be built around. We first define the tamper-evident environment. Then, we discuss the protections required for security and additional mechanisms required for usefulness. Finally, the secure processor architecture as-

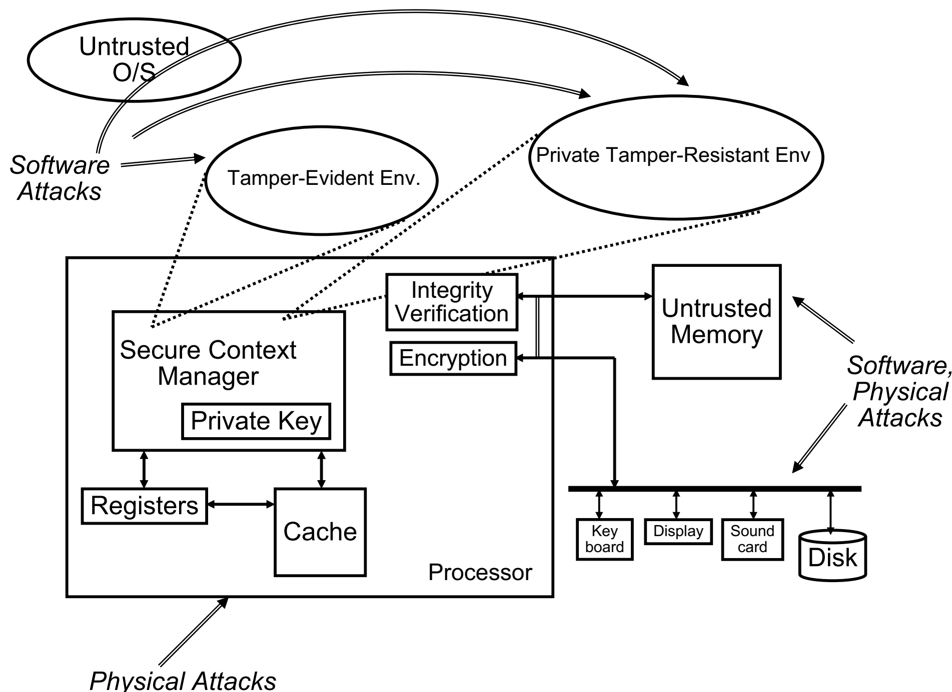


Figure 1: Our secure computing model.

suming an untrusted operating system is described.

The tamper-evident execution environment guarantees that any physical or software tampering that can alter the behavior of a program is detected or prevented. In other words, the integrity of a program execution is guaranteed. Tamper-evident execution does not provide any privacy for code or data; a private tamper-resistant environment is required for privacy.

To enter tamper-evident execution mode, an application program executes an `enter_tee` (enter tamper-evident environment) instruction, specifying regions of code and data that should be verified. Similarly, a program exits the tamper-evident mode using an `exit_tee` instruction.

A valid execution of a program on a general-purpose time-shared processor can be guaranteed by securing three potential sources of attacks: attacks on *initial state*, *state on a context switch*, and *on-chip/off-chip memory*.

- *initial state*: For a correct execution, the initial state of a program must be properly set up. In particular, the program's code and data must be unmodified, execution must start at the instruction that follows `enter_tee`, and in some architectures, the stack pointer must not overlap with the program's code or data.

- *state on a context switch*: The program state can be tampered with on a context switch when control transfers to either an operating system or another program. Therefore, the processor needs to verify the program state whenever it resumes execution.

- *on-chip/off-chip memory*: Finally, the integrity of program instructions and data in the on-chip/off-chip memories should be protected. On-chip registers and caches are secure from physical attacks but can be tampered with by malicious or buggy software. Off-chip memory including pages swapped out to the disk is vulnerable to both physical and software attacks.

For the tamper-evident execution to be useful in practice, a user should be able to trust the result provided by a system when all communication channels from a processor are untrusted. In order to trust the result, a user first needs to authenticate that the system has a valid tamper-evident execution environment (*system authentication*). Then, the program executed by a processor should be verified to be the one that is sent by a user (*program authentication*). Finally, a processor should have an authenticated communication channel with a user (*message authentication*).

3.2 The Secure Context Manager

To have a tamper-evident environment without trusting the operating system, the processor needs to keep

Vulnerability	Solutions
Initial state	PC dependent program hash, Check the stack pointer
Context switching	Store the hash of the state on an interrupt and verify the hash on a resume
On-chip caches	Tagged with secure process IDs (SPID)
Off-chip memory	Virtual memory verified by a processor: Memory integrity checking mechanisms
Capabilities	**Mechanisms**
System authentication	Processor: Private/public keys in a processor
Program authentication	Hash of a program by a processor
Message authentication	Out: Sign with a private key
	In: Public keys in a program

Table 1: The implementation of the tamper-evident environment with an untrusted operating system.

track of the processes that it is running in tamper evident mode, so that it can securely keep track of their state. We introduce a *secure context manager (SCM)*, which is specialized hardware in the processor that ensures proper protection for each process.

The SCM maintains a table that holds various protection information for each process executing in the tamper-evident mode. The table entry for a process consists of a secure process ID (SPID), the program hash ($H(Prog)$), the hash of registers ($H(Regs)$), and a hash used for memory integrity verification. We refer to the table as the SCM table. A new entry is created by the `enter_tee` instruction, and deleted by the `exit_tee` instruction. The operating system can also delete an entry as it has to be able to kill processes; this feature is not a security issue, as it does not allow the operating system to impersonate the application that it killed.

The SCM table can be entirely stored on the processor as in [9], however, this severely restricts the number of tamper-evident processes. Instead, we store the table in a virtual memory space that is managed by the operating system. Memory integrity verification mechanisms prevent the operating system from tampering with the data in the SCM table. An on-chip cache similar in structure to a TLB is used to store on-chip the SCM table entries for recent processes.

3.2.1 Protection Mechanisms

To ensure a valid initial state, the SCM computes a hash of essential program code and data (and checks the initial stack pointer on architectures such as x86 to avoid a stack overflow if an interrupt occurs) when the `enter_tee` instruction is executed. The instruction specifies the code and data segments to be hashed for authentication purposes. The trusted code segment is assumed to include the `enter_tee` instruction. Therefore, if the entry point changes, the program hash also changes and the user will be able to detect the tampering.

Given that context switching is a rather complicated task, we let the untrusted operating system manage all aspects of multitasking. The processor nevertheless has to verify that a tamper-evident process' state is correctly preserved when it is not executing. For that reason, the SCM stores all the process' register values in the SCM table when the interrupt occurs, and restores them at the end of the interrupt.

Finally, on-chip/off-chip memory integrity should be checked by the processor. First, on-chip caches are protected using tags. When a tamper-evident process accesses instructions or data, the corresponding cache block is tagged with the process' SPID. For a cache hit, a request is serviced with the existing cache block only if the tag matches the active SPID. Otherwise, the block is invalidated and reloaded from the off-chip memory, which invokes integrity verification by the off-chip mechanism.

For off-chip memory, including pages swapped to disk, we use memory integrity verification algorithms (see Section 5). The memory verification algorithms are applied to each tamper-evident process' virtual memory space. Each tamper-evident process uses different hash trees or multiset hashes to protect its own virtual memory space. Changes made by a different process are detected as tampering. Because we are protecting virtual memory, pages are protected both when they are in RAM and when they are swapped to disk.

The integrity verification mechanism described above does not allow any sharing of memory space among processes. In order for a process to read

data from another process without an error, we provide an explicit instruction `load_nocheck`. The `load_nocheck` instruction reads data without any integrity verification.

3.2.2 Authentication Mechanisms

To send authenticated messages to a user, a program can use the `sign_msg` instruction. It returns $\{H(Prog), M\}_{SKp}$, where $H(Prog)$ is the hash of the program that was computed when the `enter_tee` instruction was executed, and which was stored in the SCM table. The processor only signs the message if the program is in tamper-evident mode, and always includes the program hash in the signature. That way, when the user receives a message signed by the processor's secret key SKp, he knows that the message is from a particular processor (system authentication). The program hash is used to identify the program that sent the message (program authentication), and the message is also signed (message authentication).

3.2.3 Performance Implication

Most mechanisms that are required for tamper-evident processing have marginal overhead on the processor performance. The `enter_tee` instruction and the `sign_msg` instruction incur cryptographic hash computation and private/public key signing, respectively, which are rather expensive operations. However, these instructions are only used very infrequently; the `enter_tee` instruction is only for the beginning of a program, and the `sign_msg` instruction is only for exporting trusted results. Thus, the overhead will be amortized over a long execution period.

There are three mechanisms that are frequently used at run-time: register protection on an interrupt, on-chip cache tagging, and off-chip integrity verification. Fortunately, the performance overhead of register protection and cache tagging is negligible. Register protection simply requires storing the register in the SCM table, and the cache tags do not increase cache access time although they occupy additional on-chip storage.

The only significant performance overhead comes from off-chip integrity verification. The integrity verification consumes additional memory bandwidth to access meta-data such as hashes or time stamps on every memory access, and may also cause additional latency for the `sign_msg` instruction. Our companion paper [18] indicates that the integrity verification incurs less than 5% performance degradation in most cases, and 15% degradation in the worst case when the signing operation is infrequent. Therefore, even if we include the amortized overhead of the start-up and signing costs, our tamper-evident processing can be done with less than 10% performance overhead in most cases.

3.3 Private Tamper-Resistant Architecture

This section extends the tamper-evident processor architecture presented in the previous section to support a private tamper-resistant execution. Additional mechanisms are needed to ensure the privacy of registers, on-chip caches, and off-chip memory. Two new instructions are needed to provide privacy:

- `set_key`: Sets the key K_{static} that is used to decrypt *static data. That is, data that is encrypted in the program binary.*
- `store_private`*: Private version of the regular store instructions. If data is stored using the* `store_private` *instruction, it is to be encrypted in memory.*

3.3.1 Ensuring Privacy

The main mechanism that we use to protect the privacy is symmetric key encryption. Whenever data that needs to remain private goes off-chip, the processor encrypts it. Fast symmetric encryption/decryption can be used because the data only needs to be read by the processor that wrote it in the first place. The processor holds a master key K_M, and each process uses a pair of keys, K_{static} and $K_{dynamic}$. K_{static} is obtained from the `set_key` instruction, and $K_{dynamic}$ is randomly chosen by the processor when `enter_tee` is called. The static key is used to decrypt data from the program binary. The dynamic key is used to encrypt and decrypt data that is generated during the program's execution. Section 6 details the encryption mechanism. To determine when data should be decrypted using the static key, the processor checks for the special encryption time stamp value of zero. The keys for each process are stored in the SCM table, the private parts of which are protected by the master key K_M when they are stored in untrusted memory.

Simply encrypting memory does not provide complete opacity of program operation. Much information can be leaked via memory access patterns or other covert channels [1]. Here we will assume that programs are well-written and do not leak their secrets via those channels. Techniques exist (e.g., [10]) which can check programs for information leaks.

Whether data should be encrypted is determined on a page granularity using an extra bit per page in

the page table. However, the page table is under the control of the untrusted operating system, so it cannot be relied on. Therefore, the program additionally explicitly indicates whether it wants to use the `store` or `store_private` instruction when writing to memory. If the bit in the page table does not match the store instruction, the secure process aborts. One bit per cache block (the 'private bit' is additionally used to remember whether the block is private, to prevent the operating system from changing the page table entry between the store and the write-back. The bit in the page table is used when reading data from memory, to decide whether to decrypt it or not. If the bit is tampered with, incorrect data gets put into the cache, and the memory integrity checker aborts the program. The encryption/decryption is done by a hardware engine placed between the on-chip L2 cache and the off-chip memory bus.

The privacy of registers and on-chip caches must also be protected by the processor. Indeed, when an interrupt occurs, the interrupt handler, which in general cannot be trusted, must be prevented from reading the private tamper-resistant process' register values. Therefore, once the register values are stored in the SCM, the working copy of the registers is cleared so that the interrupt handler cannot see their previous values.

The privacy of on-chip caches is protected using the same tags that are used for integrity verification. If the private bit is set, the processor only allows access to a block in the cache when the SPID of the block matches the active SPID.

3.3.2 Secret Sharing

Programs in the tamper-evident architecture could only use private/public key cryptography to authenticate a message from a user, because privacy was not provided. In the private tamper-resistant environment, privacy is available, so programs can use symmetric key cryptography to interact with the outside world.

To share a secret with a program, a user sends an encrypted message $E_{PKp}(H(Prog), M)$. To decrypt the message, the program uses a special `decrypt_msg` instruction. The processor decrypts the message with its private key, and returns M only if $H(Prog)$ matches the current program hash. Therefore, the message can only be read by the program with the hash $H(Prog)$ running on a valid private tamper-resistant processor.

3.3.3 Performance Implication

The only additional run-time mechanism that is required for private tamper-resistant processing over tamper-evident processing is off-chip memory encryption. Therefore, the performance overhead of our private tamper-resistant architecture can be estimated by considering only memory integrity verification and memory encryption given that the memory integrity verification is the only major performance overhead in the tamper-evident architecture. We study this performance impact quantitatively through detailed simulations in Section 7.

4 Applications

We describe two representative applications enabled by the AEGIS processor, Certified Execution and Digital Rights Management.

4.1 Certified Execution

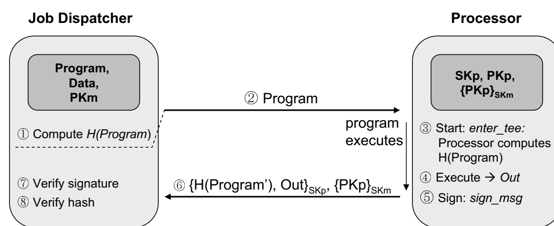


Figure 2: Certified execution for distributed computation.

A typical example of certified execution is grid computing. At present, computation power is a commodity that undergoes massive waste. Most computer users only use a fraction of their computer's processing power, though they use it in a bursty way, which justifies the constant demand for higher performance. A number of organizations, such as `SETI@home` and `distributed.net`, are trying to tap that wasted computing power to carry out large computations in a highly distributed way. This style of computation is unreliable as the person requesting the computation has no way of knowing that it was executed without any tampering. In order to obtain correctness guarantees, redundant computations are performed, resulting in a loss of efficiency. Moreover, to detect malicious volunteers, it is assumed that these volunteers do not collude and are continuously malicious [14].

Using a tamper-evident environment as described in Section 3, a certificate can be produced that proves that a specific computation was carried out on a specific processor chip. The person requesting the computation can then rely on the trustworthiness of the chip manufacturer who can vouch that he produced the processor chip, instead of relying on the owner of the chip.

Figure 2 outlines a protocol that could be used by a job dispatcher to do certified execution of a program on a remote computer. First (1) the job dispatcher needs to know the hash of the program that it is sending out. For simplicity, we assume that the program encompasses all the necessary code and data for the run. The program is sent to the secure processor (2), which proceeds to run it. The program enters tamper-resistant mode by using the `enter_tee` instruction (3), at that time, a hash of the program gets computed for later use. The program executes and produces a result (4). The result gets concatenated with the program's hash and signed (5). The processor returns the signed result to the job dispatcher along with a certificate from the manufacturer that proves certifies the processor's public key as belonging to a correct processor (6). The job dispatcher checks the signature (7) and the program hash (8) before accepting the program's output is correct.

4.2 Application: Digital Rights Management

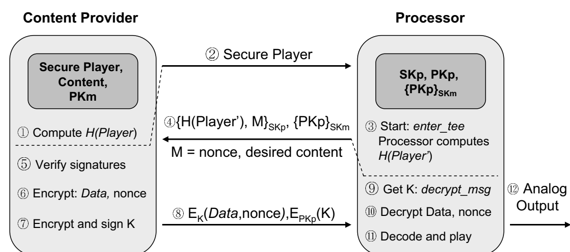


Figure 3: Digital rights management with private tamper-resistant architecture.

Digital Rights Management (DRM) is a hot topic since the advent of large scale sharing of copyrighted media over the internet. We are starting to see applications that attempt to enforce simple DRM policies [16]. A typical scenario is for an individual to buy a media file that can only be played once. This type of policy is enforced by encrypting the the media file so that it can only be decoded by an authorized reader, which enforces the single use policy. Unfortunately, a determined attacker can use debugging tools to get the player to provide him with a decrypted version of the media file, thus breaking the DRM scheme.

Figure 3 shows the main elements of a DRM implementation. The content provider develops a player application, and identifies it by its program hash (1). The player is then distributed to customers (2). When customers run the player, it uses the `enter_tee` instruction to enter secure mode (3). The player allows the user to pick the content he desires. It then sends the choice to the content provider, along with a nonce (4); the message is signed as in the certified execution application so that the content provider can check its authenticity (5). The content provider then encrypts the desired data and the nonce (6) with a randomly selected key. The key is encrypted with the processor's public key (7). All the encrypted data is then sent to the processor (8), which decrypts it (9, 10). Once the nonce is checked to make sure that stale data isn't being replayed (we assume that the data and nonce were encrypted in a non mutable way, i.e., that it is hard replace the nonce by a chosen nonce), the data can be decoded (11) and played over an analog channel (12). As with any DRM scheme, this scheme can be broken by tapping into the final analog stream, but in that case, the ripped data is of lower quality than the original media file.

5 Memory Integrity Verification Mechanisms

Memory integrity checking mechanisms [18] can be added as a layer between the L2 cache and the encryption mechanisms that we describe in section 6. For example, a hash tree can be maintained with the hashes computed over the plaintext data, the data being encrypted when it is stored in memory (the root of the tree is kept in the processor where it can be used to verify the integrity of the processor's operations on the memory). Memory integrity checking mechanisms check that the value the processor loads from a particular address is the most recent value that it stored to that address [18]. Thus, if an adversary tampers with the data, he will be detected by these mechanisms. The advantage of having the integrity checking mechanisms protect the plaintext data, instead of the encrypted data, is that the page table does not have to be trusted (the page table decides which blocks have to be encrypted/decrypted, and which do not).

We assume that the processor speculatively uses data fetched from off-chip memory while integrity checking is performed in the background. Thus, integrity checking latency does not directly add to data access latency seen by a processor. There are exceptions to this rule, however. In a tamper-evident environment, the processor must wait for integrity checking if there is an instruction that signs a message (`sign_msg`) to be exported outside of the processor. In a private, tamper-resistant environment, besides waiting if there is a signing instruction, the processor must also wait for the integrity checking if there is an instruction that stores plaintext data to off-chip

memory, for example, the `store` instruction.

For untrusted disk, as virtual memory is being protected, pages will already be protected by the memory checking when they are stored on disk. For Direct Memory Access (DMA), an unprotected area for use in DMA transfers can be set aside in virtual memory. When the transfer is done, the process can authenticate the data, using some scheme of its choosing, and decrypt the data if necessary. If the data is verified successfully, the process can then copy it to authenticated memory.

6 Encryption Mechanisms

Encryption of off-chip memory is essential for providing privacy to programs. Without encryption, physical attackers can simply read confidential information from off-chip memory. On the other hand, encrypting off-chip memory directly impacts the memory latency because encrypted data can be used only after decryption is done. This section discusses issues with conventional encryption mechanisms and proposes a new mechanism that can hide the encryption latency by *decoupling computations for decryption from off-chip data accesses.*

6.1 Advanced Encryption Standard (AES)

For off-chip memory encryption, we use a symmetric key encryption algorithm rather than public/private key algorithms. In our case, it is easy to use symmetric keys because the same processor performs both encryption and decryption. Also, symmetric encryption algorithms are simpler to implement than public key algorithms, they consume less power, and are significantly faster.

The National Institute of Standards and Technology (NIST) specifies Rijndael as the Advanced Encryption Standard (AES), which is an approved symmetric encryption algorithm [12]. AES is one of the most advanced symmetric encryption algorithms in terms of both security and performance. We base our subsequent discussions on AES as a representative symmetric algorithm.

AES can process data blocks of *128 bits* using cipher keys with lengths of *128*, *192*, and *256 bits*. The critical path of one round consists of one S-box look-up, two shifts, 6-7 XOR operations, and one 2-to-1 MUX. This critical path will take 2-4 ns with the current 0.13μ technology depending on the implementation of the S-box look-up table. Therefore, encrypting or decrypting one 128-bit data block will take about 20-64 ns depending on the implementation and the key length.

When the difference in technology is considered, this latency in good agreement with one custom ASIC implementation of the Rijndael in 0.18μ technology [8, 15]. It reported that the critical path of encryption is 6 ns and the critical path of key expansion is 10 ns per round with 1.89 ns latency for the S-box. Their key expansion is identical to two rounds of the AES key expansion because they support 256-bit data blocks. Therefore, the AES implementation will take 5 ns per round for key expansion, which results in a 6 ns cycle per round, for a total of 60-96 ns, depending on the number of rounds.

Given the gate counts in [15], a 128-bit block encryption using AES without pipelining costs approximately 75,000 gates. If we implement AES fully in parallel for the four 128-bit blocks in an L2 cache block to match off-chip memory bandwidth, the module should be duplicated four times. Therefore, the AES implementation will result in the order of 300,000 gates.

6.2 Direct Block Encryption

We encrypt and decrypt off-chip memory on a L2 cache block granularity because memory accesses are carried out for each cache block. Encrypting multiple cache blocks together will require accessing all those multiple blocks for decrypting any part of them.

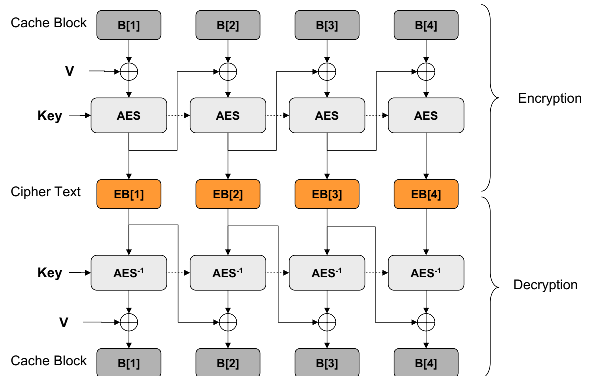


Figure 4: Encryption mechanism that directly encrypt cache blocks with the AES algorithm.

The most straightforward approach is to use a L2 cache block as an input data block of the AES algorithm. For example, a 64-B cache block `B` is broken into 128-bit chunks (`B[1]`, `B[2]`, `B[3]` and `B[4]`), and encrypted by the AES algorithm. Figure 4 illustrates this mechanism with Cipher Block Chaining (CBC) mode. In this case the encrypted cache block `EB = (EB[1], EB[2], EB[3], EB[4])` is generated by `EB[i]` $= AES_K($`B[i]` $\oplus$ `EB[i-1]`$)$, where `EB[0]` is an initial vector `I`. To prevent adversaries from comparing whether two cache blocks are the same or not,

I can be randomized and stored in off-chip memory along with data.

This scheme perfectly serves our purpose in terms of security, however, it has a major disadvantage for performance. On a L2 cache miss, an encrypted cache block is read from memory. Since decryption can only start after reading data from off-chip memory, the decryption latency is directly added to the memory latency and delays the processing (See Figure 7 (a)). For example, if the memory latency is 120 ns and the decryption latency is 40 ns, the processor will see a load latency of 160 ns.

6.3 One-Time-Pad Encryption

The main problem of the direct encryption scheme is that most of the AES decryption latency cannot be overlapped with the memory access. We therefore adopt a different encryption mechanism that decouples the AES computation from the corresponding data access using one-time-pad encryption [2] and time stamps.

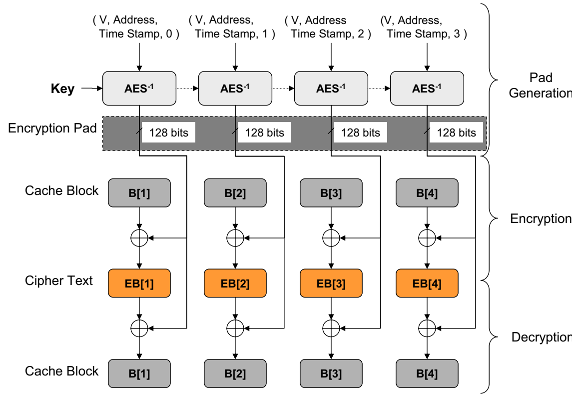


Figure 5: Encryption mechanism that uses one-time-pads from the AES algorithm with time stamps.

Figure 5 illustrates the scheme. A cache block, B, consists of four chunks, B[1], B[2], B[3], and B[4]. Each chunk is XOR'ed with an encryption pad, and the resulting encrypted cache block, EB, is stored in off-chip memory. To decrypt the block, the encrypted cache block, EB, is XOR'ed with the same encryption pad.

To obtain encryption pads, the AES algorithm is used with a time stamp. To generate an encryption pad for the 128-bit chunk, B[i], of a cache block, B, the processor decrypts $(\texttt{V}, \texttt{Address}, \texttt{TS}, i)$ with a secret key K.[3] V is a fixed bit vector that makes the input 128 bits, and can be randomly selected by the processor at the start of program execution. TS is

[3] In general K is the dynamic key $K_{dynamic}$, except when TS is zero, in which K_{static} is used (see section 3.3).

- For an L2 cache write-back
  write-back-block(Address, B):
  1. Increment TIMER. TS = TIMER.
  2. For each $0 \le i \le 3$
     (a) OTP[i] = AES_K^{-1}(V, Address, TS, i).
     (b) EB[i] = B[i] $\oplus$ OTP[i].
  3. Write TS and EB to in memory.
  4. Cache TS. (Note: This step was used for the simulations, but better performance can be expected if we omit it)

- For an L2 cache miss
  read-block(Address):
  1. Check the cache for the time stamp for Address. If the time stamp is not in the cache, read it from in memory. Denote the time stamp as TS.
  2. For each $0 \le i \le 3$
     (a) Start OTP[i] = AES_K^{-1}(V, Address, TS, i).
  3. Read EB from Address in memory.
  4. For each $0 \le i \le 3$
     (a) B[i] = EB[i] $\oplus$ OTP[i].
  5. Cache TS and B.

Figure 6: One-Time-Pad Encryption Algorithm.

a time stamp that is the current value of TIMER. TIMER is a counter that the processor increments for every write-back of a cache block. The processor maintains TIMER on-chip where it cannot be tampered with. TS is stored in the clear in the off-chip memory with the cache block. As (Address, TS) is unique for each write-back to memory, the encryption pads are used only once.

Figure 6 details the scheme. write-back-block is used to write dirty cache blocks to memory[4]. In steps 1 to 3, the TIMER is increased, and the block is encrypted using a one-time pad. The time stamp used to create the pad is cached in a special time stamp cache in step 4.

read-block is used to read cache blocks from memory. The first step is to check if Address's time stamp is in the cache. If not, the time stamp is fetched from memory. In either case, once the time stamp is retrieved, we immediately start with the generation of the OTP using AES in step 2. *The pad is generated*

[4] If the block that is being evicted is clean, it is simply evicted from the cache, and not written back to memory. This avoids incrementing TIMER in the processor and updating TS in memory; this implies that we do not need to update EB by decrypting and re-encrypting with a new time stamp.

while `EB` *is fetched from memory in step 3.* Once the pad has been generated and `EB` has been retrieved from memory, `EB` is decrypted in step 4. In step 5, `EB` is cached in L2, and `TS` is cached in the special time stamp cache. We are assuming that time stamps are stored separately from data in memory, and that a cache-block sized set of time stamps is loaded along with `TS`. Therefore, when we cache `TS`, we are also caching neighboring time stamps, which are likely to be used in subsequent accesses because of spatial locality.

When the TIMER reaches its maximum value, the processor changes the secret key and re-encrypts blocks in the memory. The re-encryption is very infrequent given an appropriate size for the time stamp (32 bits for example), and given that the timer is only incremented when dirty cache blocks are evicted from the cache. We do not need to increment `TS` during re-encryption, because `Address` is included as an argument to AES_K^-, thus guaranteeing the unicity of the one-time-pads. This trick allows us to extend the period between re-encryptions. During re-encryption, the processor uses page table bits to determine which data has to be re-encrypted.

Also, we note that, instead of a global TIMER, we could use per-address time stamps, `TS`. In this case, `TS` for `Address` should be fetched from the cache and incremented in step 1 of write-back-block. We need a slightly more complicated cache replacement policy for time stamps to ensure that `TS` for a cache block `B` is not evicted from the cache before `B` is evicted from the cache. An intermediate variant of this scheme is to only apply this improved scheme when `TS` happens to be present in the cache. For these schemes to work, we have to ensure that the memory integrity checking mechanisms detect any altering of the time stamps by the adversary before the block gets written-back. The advantage of per-address time stamps is that they can be smaller in size.

Security of the Scheme The conventional one-time-pad scheme is proven to be secure [2]. Therefore, to prove the security of our scheme, we only need to prove that it is infeasible for an adversary to find the encryption pad for a cache block with a particular time stamp. We assume that the adversary knows `V`, TIMER, and possibly many encryption pads for different time stamps.

The security of our encryption pad can be easily proven by the property of a good symmetric encryption algorithm. First, an adversary cannot find the encryption pad given the input `(V, Address, TS, i)` because it implies decrypting a cipher text. Also, an adversary cannot find the encryption pad for one time stamp from the encryption pad for another time stamp because it breaks the non-malleability of the encryption algorithm.

Hiding Latency Unlike the direct encryption scheme, the data access and the AES computation are independent in our new scheme. Therefore, the encryption latency can be hidden from the processor by overlapping AES computations with data accesses.

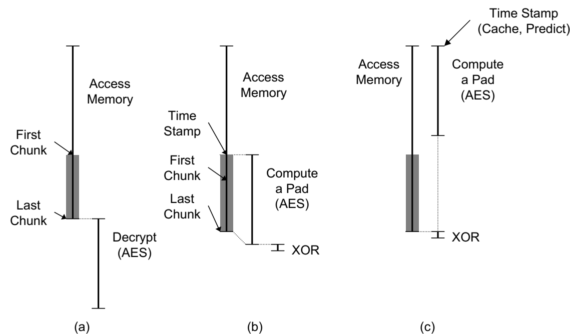


Figure 7: Impact of encryption mechanisms on memory latency.

Computing an encryption pad requires the time stamp for the cache block. Without caching or speculation, the AES computation for decryption starts after the time stamp comes back from the off-chip memory as shown in Figure 7 (b). This computation is overlapped with the following bus accesses for the cache block. Once the entire cache block is read and the pad computation is done, an XOR operation is performed for decryption. Although we may not hide the entire AES latency, our scheme can hide significant portion of the latency even in the worst case. For example, if it takes 80 ns for readying the first chunk and 40 ns for the rest of the chunks in a cache block, we can hide 40 ns of the AES latency.

When overlapping the AES computation with data bus accesses is not sufficient to hide the entire latency, the time stamp can be cached on-chip or speculated based on recent accesses. In this case, the AES computation can start as soon as the memory access is requested as in Figure 7 (c), and completely hides the encryption latency.

The ability to hide the encryption latency obviously benefits the processor performance. It also enables a less aggressive implementation of the AES algorithm. If our scheme can always hide up to 40 ns latency by overlapping with data accesses, the generation of the 4 one-time-pad blocks no longer has to be done in parallel.

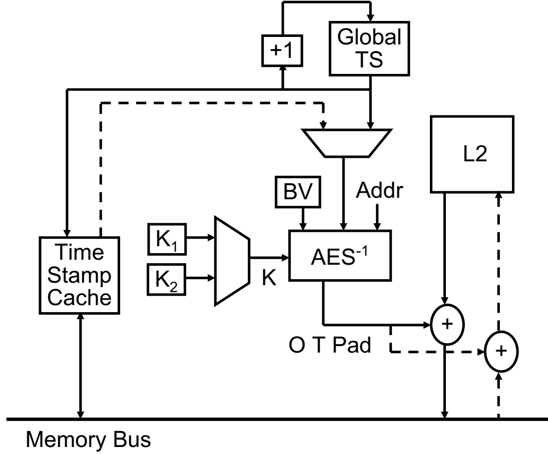


Figure 8: The implementation of the one-time-pad encryption algorithm.

Architectural parameters	Specifications
Clock frequency	1 GHz
L1 I-caches	64KB, 2-way, 32B line
L1 D-caches	64KB, 2-way, 32B line
L2 caches	Unified, 1MB, 4-way, 64B line
L1 latency	2 cycles
L2 latency	10 cycles
Memory latency (first chunk)	80 cycles
I/D TLBs	4-way, 128-entries
Memory bus	200 MHz, 8-B wide (1.6 GB/s)
Fetch/decode width	4 / 4 per cycle
issue/commit width	4 / 4 per cycle
Load/store queue size	64
Register update unit size	128
AES latency	40 cycles
AES throughput	3.2 GB/s
Hash latency	160 cycles
Hash throughput	3.2 GB/s
Hash buffer	32
Hash length	128 bits
Time stamps	32 bits
Time stamp cache	32 64-B entry

Table 2: Architectural parameters used in simulations.

6.4 Implementation

Figure 8 illustrates the implementation of the one-time-pad algorithm. The dotted lines differentiate the data flow of cache misses from the flow of cache write-backs. When there is a write-back, the global time stamp is used to compute the one-time-pad, and incremented. The one-time-pad is XOR'ed with the evicted data, and written back to memory. The time stamp is stored in a time stamp cache.

The same mechanisms are used to find the physical address of the corresponding time stamp for each evicted cache block, as in memory integrity verification. The L2 cache holds the virtual address of each block. In the virtual memory space, time stamps are laid out linearly starting at TS_{Base}. Therefore, the address of a time stamp can be simply computed by

$$TimeStampAddr = TS_{Base} + \frac{Addr}{B_{L2}} \times B_{TS}.$$

B_{L2} is the L2 block size, and B_{TS} is the size of a time stamp. Once the virtual address of a time stamp is found, it is translated into the physical address using the secure PID and the second TLB next to the encryption module.

Similarly, when there is a cache miss for an encrypted block, the module reads a time stamp for the time stamp cache and computes the one-time-pad, which may access the memory on a miss. Then, the data block is read from the memory, XOR'ed with the pad, and returned to the L2 cache.

7 Experiments

This section evaluates the performance overhead of our new encryption scheme and the AEGIS processor architectures through detailed simulations.

7.1 Simulation Framework

Our simulation framework is based on the SimpleScalar tool set [3]. The simulator models speculative out-of-order processors. To model the memory bandwidth usage more accurately, separate address and data buses were implemented.

The architectural parameters used in the simulations are shown in Table 2. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance.

To capture the characteristics of benchmarks in the middle of computation, each benchmark is simulated for 100 million instructions after skipping the first 1.5 billion instructions. In the simulations, we ignore the initialization overhead of the integrity checking schemes. Given the fact that benchmarks run for a long time, the overhead should be negligible compared to the steady-state performance.

For all the experiments in this section, nine SPEC2000 CPU benchmarks [7] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`.

7.2 Encryption Performance

Figure 9 compares the direct encryption mechanism with the one-time-pad encryption mechanism. The instructions per cycle (IPC) of each benchmark is normalized by the IPC of standard processor without encryption. In the exeriments, we simulated the worst case when *all instructions and data are encrypted in the memory.* Both encryption mechanisms degrade the processor performance by consuming additional memory bandwidth for either time stamps or

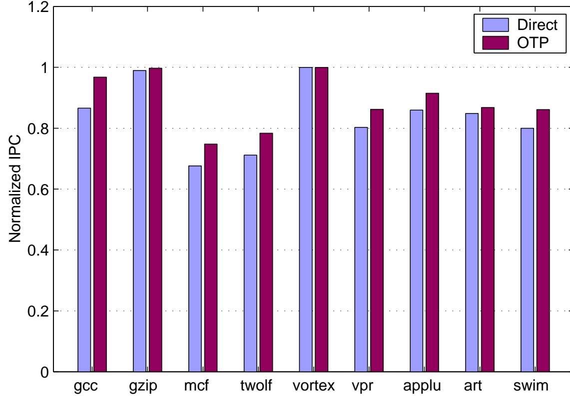


Figure 9: The performance overhead of direct encryption and one-time-pad encryption.

initial vectors, and by delaying the data delivery for decryption.

As shown in the figure, the memory encryption for this configuration results in up to 25% performance degradation for the one-time-pad encrytion, and 32% degradation for the direct encryption. The one-time-pad scheme outperforms the direct encryption by 6% on average, and 12% in the best case.

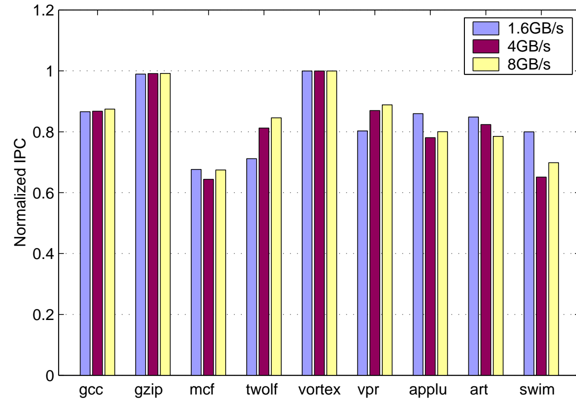
As we increase the L2 cache size or L2 block size, the overhead of encryption decreases (not shown in the figure). For larger L2 cache, there are fewer off-chip memory accesses , thus the decryption latency is less important. Larger L2 blocks reduces the bandwidth overhead of the encryption scheme.

7.2.1 Impact of the High Memory Bandwidth

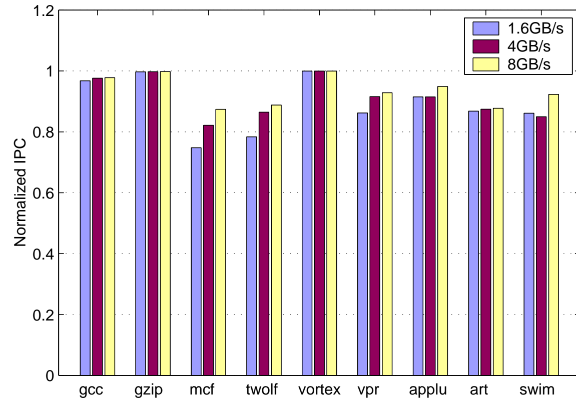
Our base configuration assumes the memory bandwidth of 1.6GB/s, which corresponds to 5 processor cycles per 8-B memory transfer in our case. Modern microprocessors are beginning to have higher bandwidth with the development of new memory and interconnect technologies. Figure 10 shows the impact of this higher memory bandwidth on the memory encryption overhead.

With high bandwidth, the performance is more sensitive to the memory latency because it is not limited by the bandwidth anymore. At the same time, the memory latency without encryption decreases as we can transfer a cache block faster, which means that the decryption latency becomes more significant in comparison to the original memory latency. On the other hand, higher bandwidth mitigates the effect of the bandwidth overhead for accessing time stamps and initial vectors.

Direct encryption cannot hide any latency and the impact of relatively increased decryption latency often outweighs the mitigated bandwidth overhead. As a result, the relative performance degradation due to decryption is often larger for higher memory bandwidth in the direct encryption scheme.



(a) Direct Encryption

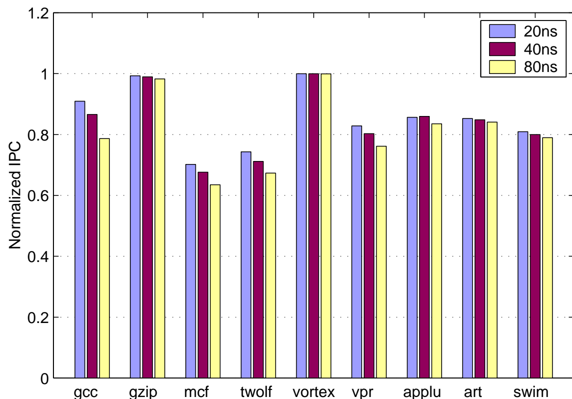


(b) One-Time-Pad Encryption

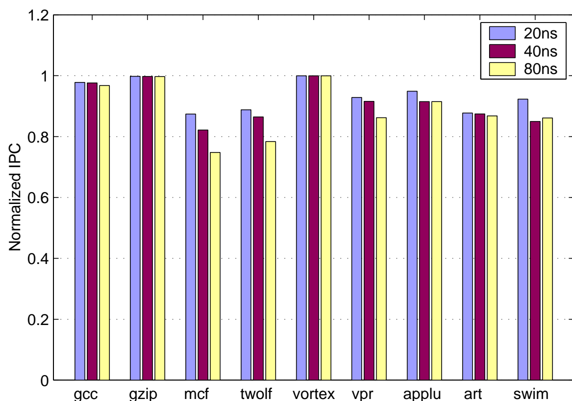
Figure 10: The impact of memory bandwidth on the memory encryption overhead.

The one-time-pad encryption scheme benefits when higher bandwidth is available. Because the time stamp can often be found in the time stamp cache (on average, the hit rate is 73% in our experiments), the AES latency can be hidden by the long memory access latency. Therefore, the overhead of the one-time-pad scheme usually decreases as the memory bandwidth increases. Thus, the advantage of the one-time-pad scheme over the direct encryption is greater as memory bandwidth increases. For example, the one-time pad scheme is 20% better than the direct scheme for `swim` in the 8 GB/s bandwidth case.

7.2.2 Impact of the AES Latency



(a) Direct Encryption



(b) One-Time-Pad Encryption

Figure 11: The impact of AES latency on the encryption overhead.

Our experiments assume 40ns latency of the AES computation. However, more advanced VLSI technology can reduce this latency. On the other hand, an implementation with longer latency is desirable to reduce the logic overhead. Figure 11 illustrates the impact of the AES computation latency on the encryption overhead. Obviously, longer latency degrades the encryption performance in both schemes.

The performance of the direct encryption is a little more sensitive to the AES latency as compared to the one-time-pad encryption scheme. This is because a portion of the latency is hidden in the latter scheme.

7.2.3 Re-Encryption Period

As noted in Section 6, the one-time-pad encryption mechanism requires re-encrypting the memory when the global time stamp reaches its maximum value. Because the re-encryption operation is rather expensive, the time stamp should be large enough to either amortize the re-encryption overhead or avoid the re-encryption itself.

Fortunately, the simulation results for the SPEC benchmarks indicates that even 32-bit time stamps are large enough. In our experiments, the processor writes back to memory every 4800 cycles when averaged over all the benchmarks, and 131 cycles in the worst case of `swim`. Given the maximum time stamp size of 4 million, this indicates the re-encryption needs to be done on every 5.35 hours (in our 1 GHz processor) on average, or 35 minutes for `swim`. For our benchmarks, the re-encryption takes less than 300 million cycles even for `swim` that has the largest working set. Therefore, the re-encryption overhead is negligible in practice. If the 32-bit time stamps are not large enough, the encryption period can increased by having larger time stamps or per-page time stamps.

7.3 Tamper-Evident Processing

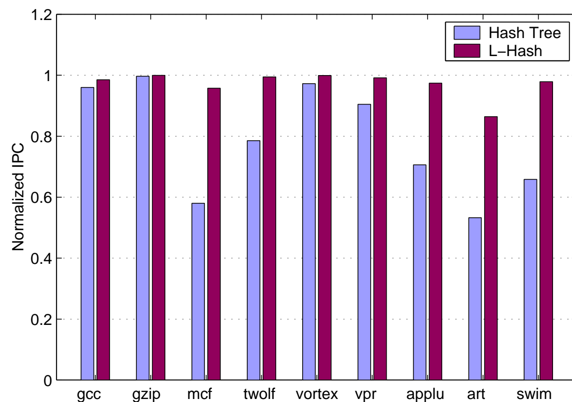


Figure 12: The performance overhead of tamper-evident processing for the baseline configuration.

Tamper-evident processing only requires memory integrity verification. In Section 3.2.3, we briefly mentioned the overheads associated with the schemes of [18]. We summarize performance results for memory integrity verification, and therefore, tamper-evident processing, for the baseline configuration in Figure 12.

There are two possible schemes for integrity verification: hash trees and log hashes. The hash tree scheme (`Hash Tree`) verifies every memory access, and the log hash scheme (`L-Hash`) verifies a sequence of memory accesses just before a signing operation.

Assuming that the signing operation is infrequent, the tamper-evident processing can be done with less than 5% performance overhead for most cases, and

15% overhead in the worst case. On the other hand, the hash tree scheme has less than 20% overhead for most cases, and 50% overhead in the worst case.

7.4 Private Tamper-Resistant Processing

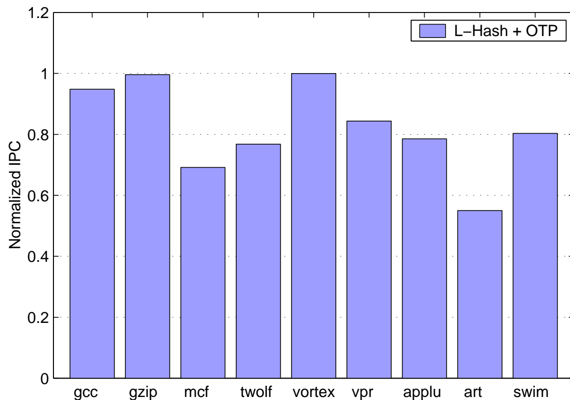


Figure 13: The performance overhead of private tamper-resistant processing for the baseline configuration.

Private tamper-resistant processing requires both memory encryption and memory integrity verification. As discussed in Section 3.3.3, the overhead of these two mechanisms are the only major concerns for our private tamper-resistant architecture. Other mechanisms either are very infrequent or do not have any performance overhead. Therefore, we estimated the performance overhead of the private tamper-resistant architecture by simulating both memory encryption and memory integrity verification as shown in Figure 13.

The figure demonstrate that the private tamper-resistant processing can be done with 55% overhead in the worst case (`art`), and less than 25% overhead in most cases. We note that these numbers correspond to the extreme case where all instructions and data are encrypted.

8 Related Research

8.1 Secure Processors

Secure co-processors have been proposed (e.g., [19], [17]) that encapsulate processing subsystems within a tamper-sensing and tamper-responding environment where one can run security-sensitive processes. A processing subsystem contains the private key of a public/private key pair [5] and use classical public key cryptography algorithms such as RSA [13] to enable a wide variety of applications. To maintain performance, the processing subsystems have invariably been used as co-processors rather than primary processors. Therefore, the primary processor also has to be protected. The processing subsystems of these processors typically assume that system software is trusted.

The eXecute Only Memory (XOM) architecture [9] is designed to run security requiring applications in secure compartments from which data can escape only on explicit request from the application. Even the operating system cannot violate the security model. However, XOM's integrity mechanism is vulnerable to replay attacks, which was also pointed out in [16]. In particular, XOM will not notice if only the first write to an address is ever actually performed. XOM can be fixed by using memory integrity verification to protect against replay attacks. In the AEGIS architecture that assumes an untrusted operating system, we have drawn from XOM, notably, the on-chip data tagging mechanism and the saving of contexts. Our implementation of our context manager is different because we use hash-trees to verify process state. This allows us to support a much larger number of processes running in tamper-evident and private tamper-resistant environments. We use the log-hash-based integrity verification mechanism of [18] for efficiency reasons. In this paper, we have proposed a fast encryption scheme that meshes well with the log-hash integrity verification scheme.

8.2 Cryptographic Processors

Conventional methods for encryption and decryption of memory blocks use DES [11], Triple DES [11], AES [12] to encrypt and decrypt memory blocks; this can appreciably increase memory access latency for reads. We have used one-time pads to hide virtually all the decryption altency.

8.3 Systems

The Trusted Computing Platform Alliance (TCPA) is an alliance led by Intel whose stated goal is 'a new computing platform for the next century that will provide for improved trust in the PC platform.' Palladium, Microsoft's proposed security model [4], is software that Microsoft says it plans to incorporate in future versions of Windows; it will build on the TCPA hardware, and will add some extra features. In Palladium, the *nexus* is a trusted security kernel. Palladium protects software from software, and it does not concern itself with physical attacks.

TCPA will implement DRM mechanisms but does not implement certified execution or secure virtual machines on its own. For this, it will probably rely on Palladium. Palladium works by providing a way

for applications to be executed in a secure context. However, hardware attacks remain possible.

TCPA and Palladium can be enhanced, i.e., made secure against a larger set of attacks, using the components in the AEGIS processor, namely, integrity verification and memory encryption. With integrity verification, applications could get guarantees that their data has not been modified, even by a physical attacker. Encrypting data that is in main memory, not just disk, will prevent physical attacks on the memory that attempt to read private data.

9 Conclusion

We have described the architecture of a processor that can be used to build secure computing systems where the processor is the only trusted component. This requires the integration of many architectural mechanisms into a conventional architecture, notably, memory integrity verification, memory encryption/decryption, and secure context management. Using simulation, we have shown that the performance overhead of integrating such mechanisms into a high-performance super-scalar processor is reasonable. We believe this overhead can likely be reduced with further architectural innovation.

References

[1] J. Agat. Transforming out timing leaks. In 27th *ACM Principles of Programming Languages*, January 2000.

[2] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 2001.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[4] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft “Palladium”: A Business Overview. In *Microsoft Content Security Business Unit*, August 2002.

[5] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[6] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon Physical Random Functions . In *Proceedings of the Computer and Communication Security Conference*, May 2002.

[7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[8] 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 (CHES 2001), LNCS 2162*, 2001.

[9] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9th *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 169–177, November 2000.

[10] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In 26th *ACM Principles of Programming Languages*, January 1999.

[11] NIST. FIPS PUB 46-3: Data Encryption Standard (DES), October 1999.

[12] N. I. of Science and Technology. FIPS PUB 197: Advanced Encryption Standard (AES), November 2001.

[13] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[14] L. F. G. Sarmenta. *Volunteer Computing*. PhD thesis, Massachusetts Institute of Technology, June 2001.

[15] 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.

[16] W. Shapiro and R. Vingralek. How to Manage Persistent State in DRM Systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[17] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. In *Computer Networks (Special Issue on Computer Network Security)*, volume 31, pages 831–860, April 1999.

[18] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Hardware mechanisms for memory integrity checking. In *Technical Report MIT-LCS-TR-872*, November 2002.

[19] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.