AEGIS: 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 for a single-chip AEGIS processor which can be used to build computing systems secure against both physical and software attacks. Our architecture assumes that all components external to the processor, such as memory, are untrusted. We show two different implementations. In the first case, the core functionality of the operating system is trusted and implemented in a security kernel. We also describe a variant implementation assuming an untrusted operating system.

AEGIS provides users with tamper-evident, authenticated environments in which any physical or software tampering by an 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, secure mobile agents, software licensing, and digital rights management.

Preliminary simulation results indicate that the 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 (TE) 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 (PTR) 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 untrusted external memory. We first show an implementation with an untrusted operating system. We also describe a variant implementation of the architecture, that may provide increased flexibility under a different secure computing model, wherein core functionality of the operating system, termed the security kernel, is trusted – the remaining part of the operating system is untrusted, as is 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

*This is an extended version of the ICS'03 paper. This paper fixes a vulnerability in the original on-chip cache tagging mechanism for virtually-addressed caches (See Section 3)

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

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 TE and PTR 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.

PTR 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 a 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] PTR 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. Finally, this PTR platform can enable secure mobile agents to perform electronic transactions on untrusted hosts [5].

The key architectural mechanisms required in an AEGIS processor that assumes an untrusted operating system are memory integrity verification, encryption/decryption of off-chip memory 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 describe a variant implementation, that assumes a trusted security kernel, wherein some of the hardware functionality can be eliminated. 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 processor.

We present our security model in Section 2. The AEGIS architecture is described in Section 3. Section 4 presents two essential mechanisms to protect off-chip memory: integrity verification and encryption. We describe how the architecture can be used for a certified execution application and a simple Digital Rights Management (DRM) application in Section 5. Simulation experiments to evaluate the performance overheads of the various mechanisms are presented in Section 6. Related work is described in Section 7, and we conclude the paper in Section 8.

[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

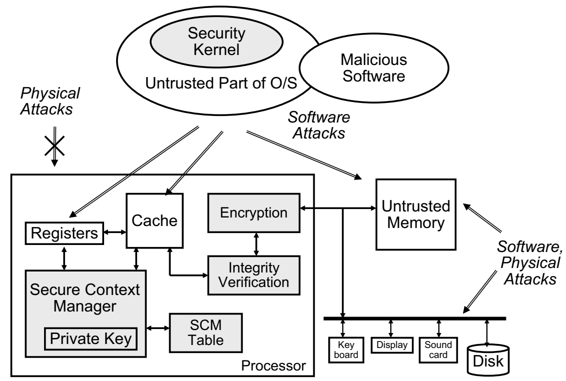

Figure 1: Our secure computing model.

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 attacks, 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 a Physical Random Function [7], or the secret part of a public key pair protected by a tamper-sensing environment [9].

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. In general, the operating system (OS) is also untrusted. Software attacks by the operating system or from other malicious software are therefore possible. In particular implementations, it may be assumed that there is a trusted part of the operating system, called the *security kernel*, that operates at a higher privilege level than the regular operating system. The processor is used in a multitasking environment, which uses virtual memory, and runs mutually mistrusting processes within TE or PTR environments.

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 [22].

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

We assume that programs are well-written and do not leak secrets via their memory access patterns. In particular, we do not handle security issues caused by bugs in an application program.

3 The AEGIS Architecture

This section describes a processor architecture with which tamper-evident (TE) and private tamper-resistant (PTR) execution environments can be built. We first focus on the high-level description of our architecture and how the environments are used by application programs. Then, the protection mechanisms to enable this architecture are discussed in more detail.

3.1 Trusted Computing Base: TCB

Our trusted computing base (TCB) consists of a processor chip and optionally a part of an operating system. We refer to the trusted core part of the operating system as the *security kernel (SKernel)*. Unless the entire operating system is trusted, the security kernel operates at a higher protection level than other parts of the operating system in order to prevent attacks from untrusted parts of the operating system such as device drivers.

In the following discussion of the high-level architecture, we do not distinguish between tasks that are accomplished by a security kernel and tasks that are accomplished by the processor. In most cases, the same functionality can be implemented in either one. We simply refer to the TCB when either one is concerned. In Section 3.4, we show how the required functionality can be partitioned between the security kernel and the processor in different ways.

3.2 Tamper-Evident Processing

The TE 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. TE mode does not provide any privacy for code or data; a PTR environment is required for privacy.

The AEGIS architecture provides the following new operations for an application program to enable TE processing:

- `enter_aegis`: Start execution in a TE environment.
- `exit_aegis`: Exit the TE environment and return to a standard processing mode.
- `sign_msg`: Generate a signature of a message and a program hash with the processor's secret key.
- `get_random`: Return a random number from physically secure random number generator. The number can be used to ensure the freshness of communication (See Section 5).

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

To enter TE mode, applications use the `enter_aegis` instruction. The instruction specifies a stub region that is used to generate a program hash ($H(Prog)$) that identifies the program. The program hash is stored in protected storage for later use. The stub region starts with the `enter_aegis` instruction and extends over a number of bytes that is specified as an argument to the instruction. The stub code gets executed immediately after the `enter_aegis` instruction, and is responsible for checking any other code and data that the application relies on. It does so by comparing their hashes with hashes that are stored in the stub region. The stub code must also check the sanity of the environment it is running in: processor mode, virtual address of the stub code (if it assumes an absolute entry point), position of the stack, etc. On some architectures such as x86, it is necessary for the TCB to check that the stack pointer is far from the stub code, as the stub would be helpless against an interrupt occurring and writing to the stack before it has a chance to change the stack pointer. This process guarantees that the *initial state* of a program is properly set up.

Once a program starts its execution in TE mode with the `enter_aegis` instruction, the TCB protects the program's state in both on-chip and off-chip memory. In fact, the integrity of the verified code and data should be protected as soon as they are used to compute the program hash $H(Prog)$. The register state of the program is protected and guaranteed to be preserved over an interrupt. The integrity of program instructions and data in the on-chip/off-chip memories is also protected. On-chip caches are secure from physical attacks, thus only need to be protected from malicious or buggy software. Off-chip memory, including pages swapped out to the disk, is vulnerable to both physical and software attacks. The TCB verifies the integrity of a block whenever it is read from off-chip memory.

The integrity verification mechanism (see Section 4.1) ensures that only one program or processor can legally modify a memory location. If the entire memory space is protected, the mechanism does not allow any sharing among different secure processes. Even any legitimate input from a I/O device would be prohibited. Therefore, a program should be able to access a part of memory space without integrity verification.

We handle this problem by using the most significant bit (MSB) of an address to determine whether the integrity of the address should be protected or not. Therefore, the upper half of the virtual memory space is protected and the lower half is not. The program lays out its code and data accordingly. This design allows a very simple implementation of the protection scheme even in hardware. This static division of memory space restricts processes to have only one half of the memory space for secure data. This is not a problem for 64-bit architectures. In the case of small address spaces, a finer granularity might be desirable to avoid wasting virtual address space.

Exporting Results The protections described above are enough to guarantee the correct execution of a program. However, in practice, there is additional functionality required for the TE mode to be useful; a user should be able to trust the result provided by a system when communication channels from a processor are untrusted.

For this purpose, a program can use the `sign_msg` operation. It returns the signature $\{H(Prog), M\}_{SKp}$ for a message M,[3] where $H(Prog)$ is the hash of the program that was computed when the `enter_aegis` instruction was executed, and SKp is the secret part of a processor's private/public key pair. The TCB signs the message only if the program is in TE 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 program (program authentication) running on a particular processor (system authentication). The signature of a message also prevent adversaries from forging messages (message authentication).

3.3 Private Tamper-Resistant Processing

The TE environment presented in the previous subsection can be extended to a PTR environment to support a private and authenticated execution. Additional protections are needed to ensure the privacy of registers, on-chip caches, and off-chip memory. One new instruction is needed to support this mode:

- `set_aegis_mode`: Enable or disable the PTR environment. Set the static key K_{static} that is used to decrypt *static content* corresponding to instructions and data that are encrypted in the program binary.

To enable or disable privacy from TE mode, programs use the `set_aegis_mode` instruction. The instruction enables the PTR environment and provides the static key encrypted with the processor's public key ($E_{PKp}\{H(Prog), K_{static}\}$), so the static key can be decrypted only by a trusted processor for a particular program. The processor decrypts the key and sets the K_{static} accordingly only if the program hash matches the hash of the executing program. The encryption scheme should be non-malleable so that an adversary cannot change the encrypted program hash and use the static key with a different program. If there is a security kernel, the operation uses $E_{PKp}\{H(SKernel), H(Prog), K_{static}\}$ to identify the trusted security kernel as well.

In the PTR environment, all the register values are considered private and protected. Whether instructions and data in the memory are private is determined using the second MSB of the address. Data stored to virtual addresses that have the second MSB set has its privacy protected.

Ensuring Privacy The privacy of registers and on-chip caches should be protected by the TCB from software attacks. When an interrupt occurs, the TCB saves the register values in private storage in the TCB and clears them before an untrusted interrupt handler starts. The TCB also protects on-chip caches so that no process can read other process' private data.

Whenever data that needs to remain private goes off-chip, the TCB encrypts it. Fast symmetric encryption and decryption can be used because the data only needs to be read by the processor that wrote it in the first place. Each process uses a pair of keys, K_{static} and $K_{dynamic}$. The static key K_{static} is used to decrypt instructions and data from the program binary, and obtained from the `set_aegis_mode` instruction. The dynamic key $K_{dynamic}$ is used to encrypt and decrypt data that is generated during the program's execution, and randomly chosen by the TCB when `enter_aegis` is called.

Simply encrypting memory is not sufficient to provide complete opacity of program operation. Information can be leaked via memory access patterns or other covert channels. Here we will assume that programs are well-written and do not leak their secrets via those channels. Techniques exist (e.g., [14, 1])

[3] If there is a security kernel within the TCB, `sign_msg` returns $\{H(SKernel), H(Prog), M\}_{SKp}$ so that a user can authenticate the security kernel as well as the processor.

which can check programs for information leaks and prevent them [9].

3.4 TCB Implementations

The high-level architecture described in the previous subsections can be implemented in many different ways depending on how to partition the required functionality between the security kernel and the processor. In general, relying more on the security kernel provides more flexibility and requires less architectural modification on the processor. On the other hand, putting mechanisms into the processor reduces the trusted code to be verified, and can sometimes result in better performance.

In this subsection, we present two reference implementations of the AEGIS architecture: the *Security Kernel Solution* and the *Untrusted OS Solution*. In the security kernel solution, some core functionality of the operating system is trusted, so that we can construct a secure system with minimal modifications to a conventional processor architecture. The untrusted operating system solution does not trust any part of the operating system (which means there is no security kernel within the TCB), and implements all mechanisms in the processor. Table 1 summarizes the two implementations.

3.4.1 Security Kernel Solution

Security Kernel Start-Up When a security kernel exists in the TCB, its identity should be verifiable by a user. In order to achieve this goal, the processor computes the hash of the security kernel $H(SKernel)$ when it boots up as in [2, 4]. After that, the integrity of the security kernel code is protected using the same mechanisms for other secure processes: trusted VM management and off-chip integrity verification. A user can identify a TCB with the security kernel hash $H(SKernel)$ and the processor's private/public key pair.

Initial Start-Up and Interrupts The security kernel manages the start-up of a program and interrupts, thus ensuring that *the initial state* is properly set up and *the states on an interrupt* are correctly restored when a program resumes execution.

Software Attacks on Memory The security kernel protects both on-chip caches and off-chip memory from software attacks. Indeed, traditional mechanisms such as virtual memory and privilege levels are adequate to protect applications from each other. Therefore, we include the virtual memory manager within the security kernel to properly protect the integrity and the privacy of memory from software attacks.

Physical Attacks on Memory Because we assume that an adversary cannot tamper with a processor chip, the on-chip caches are secure from physical attacks. To protect the off-chip memory from physical attacks, the hardware memory integrity verification mechanism in Section 4.1 is applied to the *physical memory space*. The mechanism uses hash trees to check if the value the processor loads from a particular address is the most recent value that it stored to that address. The mechanism guarantees that if the memory is written by any entity other than the processor, this tampering is detected.

When the OS swaps a page from memory to disk, the security kernel implements the hash tree scheme in software and protects the page. The hash tree allows the OS to verify the integrity of a page when the page is brought into the memory in the future. We note that it would also be possible to verify the integrity of off-chip RAM with a software checker in the security kernel as long as the integrity verification code always stays on-chip. However, this approach would significantly degrade performance compared to the hardware implementation.

Encryption The encryption and decryption of memory is done by a hardware engine placed between the integrity checker and the off-chip memory bus, which is detailed in Section 4.2. Although encryption in software is also possible, the hardware engine is chosen for performance.

For the PTR environment, the security kernel implements the `set_aegis_mode` operation. To set the static key for a program, the operation is used with $E_{PKp}\{H(SKernel), H(Prog), K_{static}\}$. The processor provides a special instruction `decrypt_key` for the security kernel. The instruction gets the encrypted key and returns $H(Prog), K_{static}$ only if the hash of the security kernel's matches the $H(SKernel)$ in the instruction. Once the security kernel obtains the decrypted key, it compares the program hash and sets the static key for a program if the hashes match. When context switching between processes, the security kernel is responsible for clearing the static key of the process that is being interrupted and, if appropriate, loading the key for the new process into the processor.

Signing Operation With a security kernel, the `sign_msg` operation is implemented as a system call. Because the program hashes are maintained by the security kernel, the operation cannot be

Problems	Security Kernel Solution	Untrusted OS Solution
SKernel start-up	Processor computes $H(SKernel)$	-
Process start-up	Managed by security kernel - trusted loader - trusted multitasking	Processor computes $H(Prog)$, checks the stack pointer
Registers on interrupts		Processor saves registers, clears the registers (PTR), and restores them on a resume.
On-chip caches	Trusted VM manager,	Secure process ID tags, virtual address for each block,
Off-chip RAM	Processor verifies physical memory	Processor verifies virtual memory
Pages on disk	Security kernel verifies paging	
Encryption (PTR)	Hardware encryption engine	Hardware encryption engine
`sign_msg`	Security kernel system call	Processor instruction

Table 1: Implementing the AEGIS architecture with/without a trusted security kernel in the operating system (OS). (PTR) indicates that the mechanism is only required for the private tamper-resistant environment.

done by the processor directly. Instead of a user level `sign_msg` instruction, the processor provides a privileged instruction `sign_kernel_msg` for the security kernel, which returns $\{H(SKernel), M'\}_{SKp}$ for message M'. Then, the security kernel uses this instruction with $M' = \{H(Prog), M\}$ to implement the `sign_msg` system call, which returns $\{H(SKernel), H(Prog), M\}_{SKp}$. Note that the $H(SKernel)$ is always included in the signature by the trusted processor so that a malicious security kernel cannot forge a message on behalf of another security kernel.

3.4.2 Untrusted OS Solution

The Secure Context Manager To have a secure execution environment without the security kernel, the processor needs to keep track of the processes that it is running in the AEGIS mode, so that it can securely keep track of their states. We introduce a *secure context manager (SCM)*, which is a specialized component in the processor that ensures proper protection for each secure process. For each secure process, the SCM assigns a non-zero secure process ID (SPID). Zero is used to represent regular processes.

The SCM maintains a table that holds various protection information for each secure process running in AEGIS mode. The table entry for a process consists of a SPID, the program hash ($H(Prog)$), the architectural registers ($Regs$), a hash used for memory integrity verification, a bit indicating whether the process is in the PTR mode, and a pair of keys for encryption (K_{static}, $K_{dynamic}$). We refer to the table as the SCM table. An entry is created by the `enter_aegis` instruction, and deleted by the `exit_aegis` 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 XOM [12], however, this severely restricts the number of secure processes. Instead, we store the table in a virtual memory space that is managed by the operating system and stored in off-chip memory. Memory integrity verification mechanisms prevent the operating system from tampering with the data in the SCM table. A specialized on-chip cache similar in structure to a TLB is used to store the SCM table entries for recent processes. To protect the encryption keys, the processor holds a master key K_M, which can be randomly generated when the system boots, and encrypts the encryption keys and register values in the SCM table when they are moved out to off-chip memory.

Initial Start-Up To ensure a valid initial start-up, the SCM implements the `enter_aegis` operation as a processor instruction. 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_aegis` instruction is executed. Once the instructions and data are used for the hash computation, they are protected by the on-chip and off-chip memory protection mechanisms, described in the subsequent paragraphs, so that they cannot be tampered with. The program hash is stored in the SCM table.

Registers on an Interrupt Given that interrupt handling and context switching are rather complicated tasks, we let the untrusted operating system manage all aspects of multitasking. The processor nevertheless has to verify that a TE 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. For PTR processes, once the register values are stored in the SCM table, the working copy of the registers is cleared so that the interrupt handler cannot see their previous values.

On-Chip Caches The on-chip caches are protected using tags. Whenever a process accesses a cache block, the block is tagged with the process' SPID. Regular processes are represented by the SPID value of zero. This SPID specifies the ownership of the cache block. Each cache block also contains the corresponding virtual address, which was used by the owner process on the last access to the block.

When a secure process accesses an address that requires integrity protection, the processor verifies a cache block before using it. If the active SPID matches the SPID of the cache block and the accessed virtual address matches the virtual address of the cache block, the access continues. Otherwise, the value of the cache block is verified by the off-chip integrity verification mechanisms, and the SPID and the virtual address of the block is updated.

Even with SPIDs and virtual address tags, malicious operating systems can still carry out a replay attack by changing the virtual-to-physical mapping if the cache is physically addressed. Let us consider the following scenario.

- Initially, a virtual address VA is mapped to a physical address $PA1$.
- Program A reads from VA. The value in $PA1$ is read from memory, checked by the integrity verification mechanism, and gets cached on-chip with SPID of A and the virtual address tag of VA.
- The operating system changes the mapping so that VA corresponds to $PA2$ that contains the same values with $PA1$.
- Program A writes a new value into VA. $PA2$ is read from memory, passes the integrity verification, gets cached, and updated. $PA1$ is still in the cache.
- The operating system changes the mapping back so that VA maps to $PA1$.
- Program A reads from VA, which returns a stale value in $PA1$.

In order to prevent the replay attacks, the processor should only allow one cache block for each virtual address of a process. A clean solution for all the problems related to the virtual-to-physical address mapping is to use virtually-addressed caches. For physically-addressed caches, the processor evicts all cache blocks for a page when a TLB entry for the page gets evicted.

In PTR mode, if a block's virtual address is in the private region, the block requires additional protection for privacy. Accesses to a private cache block are allowed only if the SPID of the cache block matches the active SPID and the active process is in the PTR mode. Otherwise, the block gets evicted from the cache and reloaded.

Off-Chip Memory For off-chip memory, we use the hardware memory integrity verification mechanism in Section 4.1. The memory verification algorithm is applied to each secure process' virtual memory space. Each TE process uses a separate hash tree to protect its own virtual memory space. Changes made by a different process are detected as tampering. Because we are protecting virtual memory space, pages are protected both when they are in RAM and when they are swapped to disk.

As described in the high-level architecture, the private cache blocks are encrypted when they are evicted from the L2 cache. Encryption and decryption is done by a hardware engine placed between the integrity checker and the off-chip memory bus (see Section 4.2).

Signing Operation The SCM implements the `sign_msg` operation as a processor instruction as described in the high-level architecture. The SCM returns $\{H(Prog), M\}_{SKp}$, which is the signature of the program hash and the message.

3.5 Performance Implication

Most mechanisms that are required for TE processing have marginal overhead on the processor performance. The `enter_aegis` instruction and the `sign_msg` instruction involve cryptographic hash computation and private/public key signing, respectively, which are rather expensive operations. However, these instructions are only used very infrequently; the `enter_aegis` 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 on every memory access, and may also cause additional latency for the `sign_msg` instruction. Therefore, the performance overhead of TE processing can be closely approximated by evaluating the performance overhead of the memory integrity verification.

The PTR processing requires only one additional run-time mechanism over TE processing: off-chip memory encryption. Therefore, the performance overhead of our PTR architecture can be estimated by only considering memory integrity verification and memory encryption. We study this performance impact quantitatively through simulations in Section 6.

4 Memory Protection Schemes

This section describes two mechanisms to protect off-chip memory: *integrity verification* and *encryption*. The memory integrity verification protects the integrity of off-chip data, and the encryption protects the privacy of the data.

Memory integrity verification mechanisms operate as a layer between the L2 cache and the encryption mechanisms, protecting the plaintext data. Therefore, an encrypted data block from memory is first decrypted and then verified by the integrity verification mechanism. Verifying plaintexts rather than ciphertext eliminates the need to protect the meta-data for encryption such as random vectors because such tampering will be detected by the integrity verification of the decrypted plaintext.

4.1 Integrity Verification

This section briefly summarizes an integrity verification mechanism based on cached hash trees [8], and discusses issues related to applying this scheme to our architecture. We use the hash tree scheme for simplicity, but note that there exists a more efficient scheme that can reduce the performance overhead of memory integrity verification [22].

4.1.1 Cached Hash Trees

Hash trees (or Merkle trees) are often used to verify the integrity of dynamic data in untrusted storage [13]. Figure 2 illustrates a hash tree. The memory space is divided into multiple chunks, denoted by V_1, V_2, etc. The chunks are the leaves of the hash tree. A parent is the hash of the concatenation of its children. In our case, each hash covers one L2 cache block. The root of the tree is stored in the SCM table where it cannot be tampered with.

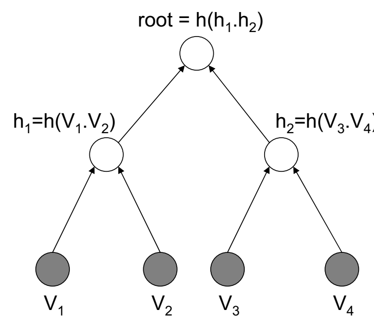

Figure 2: A binary hash tree. Each internal node is a hash of the concatenation of the data in the node's children.

To check the integrity of a node in the tree, the processor (i) reads the node and its siblings from the memory, (ii) concatenates their data together, (iii) computes the hash of the concatenated data, and (iv) checks that the resultant hash matches the hash in the parent. The steps are repeated all the way to the root of the tree.

To update a node, the processor checks its integrity as described in the previous paragraph while it (i) modifies the node, and (ii) recomputes and updates the parent to be the hash of the concatenation of the node and its siblings. These steps are repeated to update the whole path from the node to the root, including the root.

To reduce the performance overhead of the hash tree, we cache the internal hash nodes in the on-chip L2 cache with regular data. The processor trusts data stored in the cache. Therefore, instead of checking the entire path from the chunk to the root of the tree, the processor checks the path from the chunk to the first hash it finds in the cache. This hash is trusted and the processor can stop checking. When a chunk or hash is ejected from the cache, the processor brings its parent into the cache (if it is not already there), and updates the parent in the cache. Details and variants can be found in [8].

4.1.2 Initialization

To make initialization easier, we have simply attach a valid bit to each hash in the tree to indicate whether the cache line that it covers is actually present in the tree. Initially all the hashes in the tree are marked as invalid. Whenever a hash with a zero valid bit is read during memory checking, the processor automatically initializes it by computing a hash of its child cache line and setting the valid bit. This way, as soon as a virtual address has been accessed once in TE or PTR mode, the data that it contains is protected. Protecting data before that first access would be futile as the data predates the initialization of TE mode, and therefore could have been tampered with before any protection mechanism was activated. With this scheme, there is no need to allocate physical memory for hashes or data until they are used. Hashe in newly allocated pages must have zero valid flags or a memory integrity exception will be raised.

4.1.3 Tree Layout

In order to implement the hash tree scheme, a processor should be able to easily obtain a parent's address from a node's address. By laying out the nodes of the tree in breadth first or depth first manner, the address of a parent node can easily be computed from the address of a child.

When there is a security kernel case, we propose that the physical memory be split into three parts: an unverified region for programs that do not use the secure modes and for DMA accesses, a region for verified data, and a region for the nodes of the hash tree. The nodes of the hash tree should be laid out in depth first manner to make expanding the tree easy. The security determines the size of these regions base on the needs of running applications.

4.1.4 Checking Virtual Memory

When a virtual memory space is authenticated, a processor needs additional support to use the tree layout and determine the physical address of the parent hash for a cache block. In this case, the L2 cache contains virtual addresses, which are also used for on-chip cache protection. From this virtual address, a processor computes the virtual address of the corresponding parent node. We assume that the nodes of the hash tree are laid out in breadth first manner in their own virtual memory space, separate from the user space, so that the entire process virtual space can be utilized by the program. Finally, the processor needs to convert virtual addresses of parent nodes into physical addresses. For this we use a TLB; in practice, we should not use the processor core's standard TLB and should use a second TLB to avoid increasing the latency of the standard TLB. The second TLB is also tagged with process identifier bits which are combined with virtual addresses to translate to physical addresses.

4.1.5 Blocking Instructions

When data is loaded from memory, operations which do not generate a signature or reveal private information are immediately allowed to start using the fetched data. Memory checking is carried out concurrently in the background. This speculative execution on unchecked data is permissible because these operations do not break the security of our system when they are executed on tampered data, as long as an exception is eventually raised when the tampering is detected. Because these exceptions imply either a malicious OS or physical attacks, graceful recovery is not needed, and the exceptions need not be precise. Therefore, integrity checking latency is not directly added to the data access latency seen by the processor.

There are exceptions to this rule. In a TE environment, the processor must wait for integrity checking of all the previous memory accesses to complete before allowing the result of a (`sign_msg`) instruction to be exported outside of the processor. In a PTR environment, besides waiting when there is a signing instruction, the processor must also wait for the integrity checking to complete before executing an instruction that stores plaintext data (i.e., when storing to a non-private memory region).

4.1.6 Untrusted I/O

For untrusted disk, when virtual memory is being protected, pages will already be protected by the integrity verification even when they are stored on disk. For Direct Memory Access (DMA), an unprotected area for use in DMA transfers is set aside in the memory space. When the transfer is done, the process can copy it to protected memory and authenticate the data using some scheme of its choosing.

4.2 Encryption

For off-chip memory encryption, we use a symmetric key encryption algorithm rather than public/private key algorithms. In our case, it is safe to use symmetric keys because the same processor performs both encryption and decryption.

4.2.1 Advanced Encryption Standard

The National Institute of Standards and Technology specifies Rijndael as the Advanced Encryption Standard (AES), which is an approved symmetric encryption algorithm [15]. AES is one of the most advanced symmetric encryption algorithms in terms of both security and performance. While any symmetric key encryption algorithm can be used for our purposes, 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 encryption and decryption consist of 10 to 16 rounds of four transformations. 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 in 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 is in good agreement with one custom ASIC implementation of the Rijndael in 0.18μ technology [11, 19]. It is reported that the critical path of encryption is 6 ns per round 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 [19], 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 a 64-B L2 cache block, the module should be duplicated four times. Therefore, in this case, the AES implementation will result in the order of 300,000 gates.

4.2.2 Direct Block Encryption

We encrypt and decrypt off-chip memory on an L2 cache block granularity because memory accesses are carried out with that granularity. Encrypting multiple cache blocks together implies that all the blocks have to be decrypted to access any one of them.

To encrypt a dirty cache block when it gets evicted from the L2 cache, the cache block is used 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 3 illustrates this mechanism with Cipher Block Chaining (CBC) mode. The encrypted

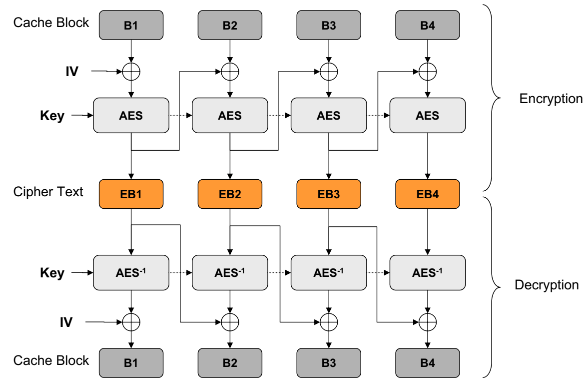

Figure 3: Encryption mechanism that directly encrypts cache blocks with the AES algorithm.

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 `IV`.

The initial vector `IV` consists of the address of the block and a random vector `RV`, and is padded with zeros to be 128 bits. To prevent adversaries from comparing whether two cache blocks are the same or not, `RV` is randomly generated as a non-zero value on each encryption. Zero indicates the block should be decrypted with the static key. After the encryption, the random vector `RV` is stored in the off-chip memory along with the encrypted cache block (`EB`). The random vectors are laid out linearly in memory as an array.

In our experiments, we used a 32-bit random vector for each cache block. Although the encryption is randomized, we note that an adversary may be able to find out that a cache block has the same value at different times if both happen to use the same random vector. To eliminate this information leak, we can replace the random vector by a counter and re-encrypt memory with a new dynamic key whenever the counter reaches its limit. When encryption is combined with the hash tree mechanism, some randomization should be included in the lowest level hashes or else we lose the benefit of randomized encryption. The most convenient way of achieving this is to include `IV` in the hash, though care must taken to ensure that using the same initial vector for the hashes and the encryption does not lead to any unexpected interaction between primitives.

For an L2 cache miss in a private memory space, an encrypted cache block (`EB`) and the corresponding random vector `RV` are read from memory. If the random vector is zero, the initial vector is set to zero and the static key is used for decrypting the block. Otherwise, the initial vector is computed from the address of the block and the random vector, and dynamic key is used. Once the data arrives, the decryption of four chunks (`B[1]`, `B[2]`, `B[3]` and `B[4]`) can be done in parallel, and stored in the L2 cache. Since decryption starts after reading data from off-chip memory, the decryption latency is directly added to the memory latency. 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.

5 Applications

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

5.1 Certified Execution

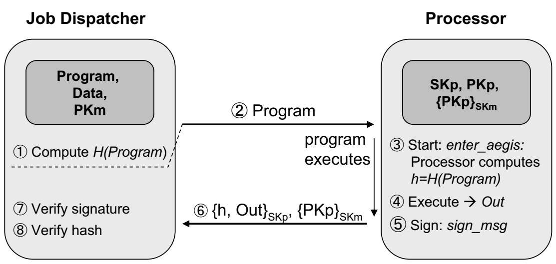

Figure 4: Certified execution for distributed computation.

A typical example of certified execution is grid computing. A number of organizations, such as `SETI@home` and `distributed.net`, are trying 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 can be performed, at the cost of reduced efficiency. Moreover, to detect malicious volunteers, it is assumed that these volunteers do not collude and are continuously malicious [18].

Using a TE 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 4 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 TE mode by using the `enter_aegis` 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 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 as correct.

5.2 Digital Rights Management

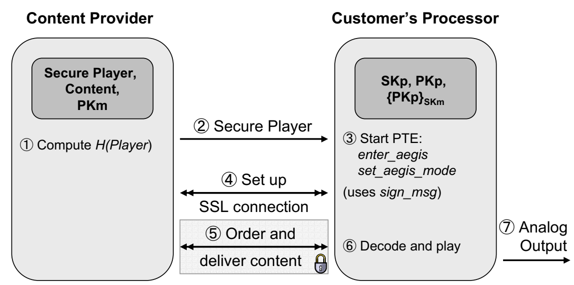

Figure 5: Digital rights management with PTR architecture.

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 [20]. A typical scenario is for an individual to buy a media file that can only be played on a single computer. This type of policy is enforced by encrypting the media file so that it can only be decoded by an authorized player on a particular processor. 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.

In Figure 5, we show how a bidirectional private and authentic channel can be created between a content provider, and a trusted program, running in PTR mode on a customer's computer. This channel can be used to send digital content to the customer. Once it is on the customer's machine, the content is managed by the trusted program which is designed to enforce the content provider's policy concerning access to the content. Since the trusted program is running in PTR mode, the content cannot be accessed except in ways that are approved by the trusted program, even if an attacker tries to use debugging tools, or tries to modify the hardware of his machine. Only physical attacks on the AEGIS processor could break the privacy of the system.

The protocol is very simple. First the content provider produces a trusted player program to run on the customer's machine. Embedded in the program is the content provider's public key. The content provider calculates a hash of the program that he will use to identify it (1), before sending it to the customer (2). When the player runs on the customer's machine, it uses the `enter_aegis` and `set_aegis_mode` instruction to enter PTR mode (3). The player program now has the public key of the server it wishes to access. It can use a standard protocol such as Secure Socket Layer (SSL) [16], with client authentication, to establish a bidirectional private and authenticated channel with the content provider (4), the `sign_msg` instruction being used to authenticate the client. In order to perform the SSL handshake, the player program requires a secure source of randomness. The AEGIS processor must therefore be equipped with a secure hardware random number generator that secure processes can use. Once the secure connection is established, it is used to transmit orders and content (5). Finally, the content is played (6) through a secure peripheral that gets encrypted content and outputs it in analog form (7).

6 Evaluation

This section evaluates the performance overhead of the AEGIS processor architecture through detailed simulations.

Our experimental results are indicative of the performance of both the security kernel and the untrusted OS solutions. The two solutions have about the same performance because they both use the same hardware mechanisms for integrity verification and encryption, and those mechanisms are responsible for the only two major performance penalties in our architecture (see Section 3.5 for more detailed discussion).

6.1 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. In the experiments, we use 4-B encryption random vector `RV` for each cache block while the memory bus is 8-B wide. To avoid wasting off-chip bandwidth, a processor always accesses two consecutive random vectors (8 Bytes) at a time, and uses small 32 entry buffer for them. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled for EV6 (21264) to maximize performance.

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
Initial vectors	32 bits
Initial vector buffer	32 8-B entry

Table 2: Architectural parameters used in simulations.

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 [10] are used as representative applications: `gcc`, `gzip`, `mcf`, `twolf`, `vortex`, `vpr`, `applu`, `art`, and `swim`.

6.2 Tamper-Evident Processing

As discussed in Section 3.5, the performance overhead of the TE processing can be estimated by the performance overhead of the off-chip memory integrity verification.

Figure 6 illustrates the impact of TE processing on application performance. For different L2 cache configurations, the IPCs are shown normalized by the corresponding IPC without TE processing. The figure first demonstrates that the performance overhead of the TE processing is relatively low. Even though the hash tree mechanism can cause over ten additional memory accesses per L2 cache miss, the performance degradation is less than 50% in the worst case.

Moreover, the performance degradation decreases rapidly as either the L2 cache size or the block size increases. Having a large L2 cache improves the per-

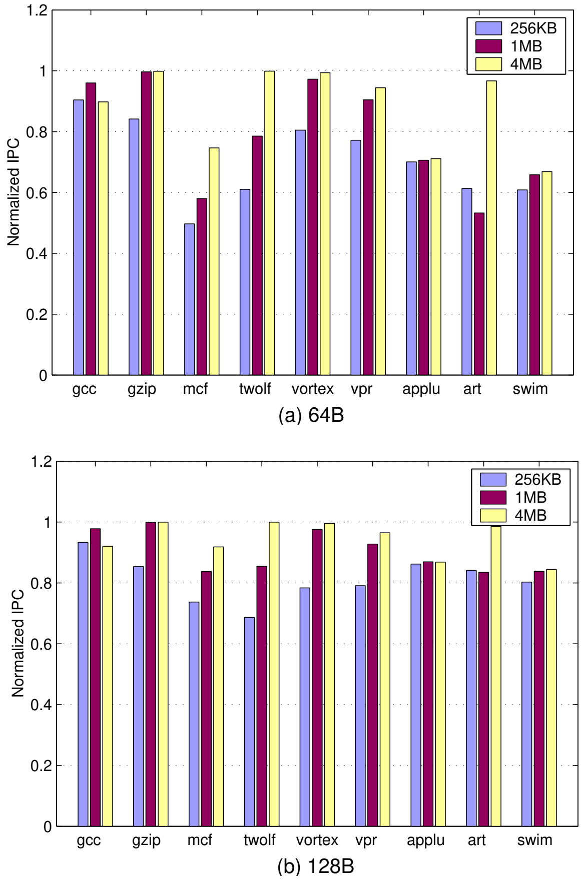

Figure 6: The performance overhead of TE processing. The results are shown for various L2 cache sizes with two block sizes (64B and 128B).

formance by reducing the number of off-chip memory accesses. Integrity verification show less performance degradation with larger L2 blocks because the larger blocks reduce the levels in the hash tree. However, we note that a larger L2 block size can degrade the performance of applications that do not use integrity verification.

Obviously, the performance overhead of the TE processing also depends on the application characteristics. Because the major overhead occurs for off-chip memory accesses, applications with less off-chip accesses show less performance degradation. For example, `gzip` shows less than 15% performance degradation for all cases, while the performance of `mcf` can be degraded by as much as 50%.

In summary, with the hash tree mechanism, the TE processing can be done with less than 20% performance overhead for most cases, and 50% overhead in the worst case. For a more detailed discussion of memory integrity verification, see [8] and [22].

6.3 Private Tamper-Resistant Processing

PTR processing requires both memory encryption and memory integrity verification. As discussed in Section 3.5, the overhead of these two mechanisms are the only major concerns for our PTR architecture. We first study the performance overhead of the encryption mechanism. Then, we estimate the performance overhead of the PTR architecture by simulating memory encryption and memory integrity verification together.

6.3.1 Encryption Performance

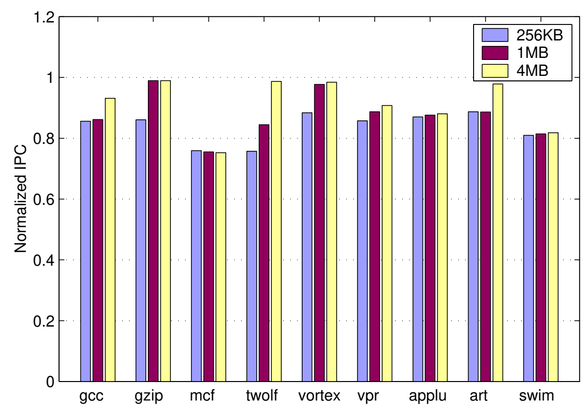

Figure 7: The performance overhead of the direct encryption scheme. Results for three different L2 caches with 64-B blocks are shown.

Figure 7 shows the performance overhead of the direct encryption mechanism compared to the standard case without encryption. In the experiments, we simulated the case where *all instructions and data are encrypted in memory.* The encryption degrades the processor performance by consuming additional memory bandwidth for random vectors, and by delaying the data delivery for decryption. Therefore, increasing the L2 cache size or L2 block size reduces the performance degradation for encryption as it does for integrity verification. In our experiments, the memory encryption results in up to 25% degradation.

Figure 8 shows the impact of changing the memory bandwidth on the encryption overhead. 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. With higher 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 en-

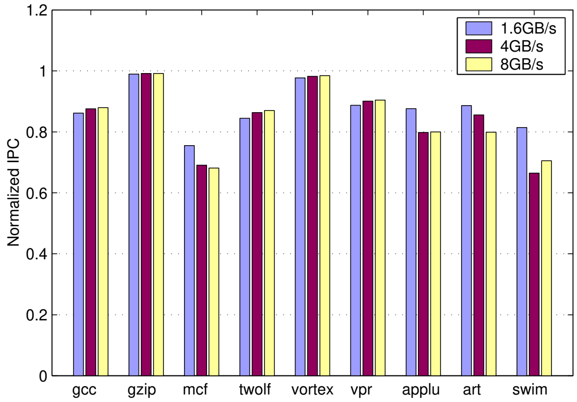

Figure 8: The impact of memory bandwidth on the memory encryption overhead.

cryption 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 random vectors. Because there are effects on both positive and negative sides, the bandwidth change does not significantly change the relative performance degradation in one way although it always improves the absolute performance.

6.3.2 Overall Performance

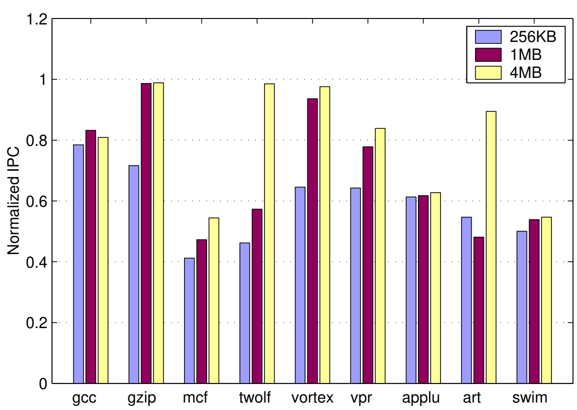

Figure 9: The performance overhead of PTR processing. Results for three different L2 caches with 64-B blocks are shown.

Finally, we study the performance of the PTR processing by simulating integrity verification and encryption together. Figure 9 demonstrates that PTR processing can be done with 60% overhead in the worst case (`mcf`), and less than 40% overhead in most cases. With the trend of larger on-chip caches and faster improvement of computation speed compared to the memory latency, the overhead should reduce with time. We also note that these numbers correspond to the case where all instructions and data are encrypted.

7 Related Research

7.1 Secure Processors

Secure co-processors have been proposed (e.g., [23], [21]) 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 [6] and uses classical public key cryptography algorithms such as RSA [17] to enable a wide variety of applications. To maintain performance, the processing subsystems have invariably been used as co-processors rather than primary processors. The processing subsystems of these processors typically assume that system software is trusted.

The eXecute Only Memory (XOM) architecture [12] is designed to run security requiring applications in secure compartments, where instructions are encrypted and 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 [20]. In particular, XOM will not notice if writes to memory are sometimes ignored. XOM can be fixed by using memory integrity verification to protect against replay attacks. In the AEGIS untrusted operating system solution, we have drawn insight from XOM, notably for the on-chip data tagging mechanism and the saving of contexts. Our implementation of the context manager is different because we use hash-trees to verify process state, which can be stored in off-chip memory. This allows us to support a much larger number of processes running in TE and PTR environments. Our architecture also provides flexibility for applications to use protection mechanisms only when they are necessary, avoiding unnecessary performance degradation.

7.2 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'. The proposed implementation in the first phase of TCPA is a *Fritz* chip - a smartcard chip or dongle soldered to the motherboard. When the PC boots up, the Fritz chip stores a hash of the boot ROM before executing it. The boot ROM stores a hash of the boot loader on the Fritz chip before executing it. This process is repeated throughout the boot process so that a trace of the system boot can be read from the Fritz chip. This is similar to the integrity-checking boot process described in [2]. Because the security mechanisms are implemented separately from the main processor, physical attacks on communication between off-chip components, such as memory and the Fritz chip, are possible.

Palladium [4], recently renamed to NGSCB, is software with minimal hardware support that Microsoft plans to incorporate in future versions of Windows. In Palladium, the *Nexus* is a trusted security kernel. Palladium protects software from software, but does not concern itself with physical attacks.

Because they are both vulnerable to hardware attacks, 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. Encryption of data in main memory would prevent physical attacks that attempt to read private data from memory.

Moreover, in the AEGIS architecture, it is possible to perform secure computation while only trusting a processor and an application program. In TCPA and Palladium, the user has to trust the entire software stack or at least part of the operating system.

8 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.

Our current architecture focuses on a single processor system where a secure process always executes on the same processor. In multiprocessor systems, a process can run on multiple processors. Therefore, efficient ways for multiple processors to share off-chip memory while preserving the protection should be developed as future work.

9 Acknowledgements

This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership.

We would also like to thank Ron Rivest and Krste Asanovic for many constructive comments, as well as all the members of our group who helped proof-read this paper.

References

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

[2] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[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] J. Claessens, B. Preneel, and J. Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3, Feb. 2003.

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

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

[8] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

[9] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

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

[11] 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.

[12] 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* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[13] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

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

[15] National Institute of Science and Technology. FIPS PUB 197: Advanced Encryption Standard (AES), November 2001.

[16] E. Rescola. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

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