AEGIS: A Single-Chip Secure Processor

by

Gookwon Edward Suh

Bachelor of Science in Electrical Engineering
Seoul National University, 1999

Master of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2001

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author .....
Department of Electrical Engineering and Computer Science
September 6, 2005

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

AEGIS: A Single-Chip Secure Processor

by

Gookwon Edward Suh

Submitted to the Department of Electrical Engineering and Computer Science
on September 6, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Trust in remote interaction is a fundamental challenge in distributed computing environments. To obtain a remote party's trust, computing systems must be able to guarantee the privacy of intellectual property and the integrity of program execution. Unfortunately, traditional platforms cannot provide such guarantees under physical threats that exist in distributed environments.

The AEGIS secure processor enables a physically secure computing platform to be built with a main processor as the only trusted hardware component. AEGIS empowers a remote party to authenticate the platform and guarantees secure execution even under physical threats. To realize the security features of AEGIS with only a single chip, this thesis presents a secure processor architecture along with its enabling security mechanisms. The architecture suggests a technique called suspended secure processing to allow a secure part of an application to be protected separately from the rest. Physical random functions provide a cheap and secure way of generating a unique secret key on each processor, which enables a remote party to authenticate the processor chip. Memory encryption and integrity verification mechanisms guarantee the privacy and the integrity of off-chip memory content, respectively.

A fully-functional RTL implementation and simulation studies demonstrate that the overheads associated with this single-chip approach is reasonable. The security components in AEGIS consumes about 230K logic gates. AEGIS, with its off-chip protection mechanisms, is slower than traditional processors by 26% on average for large applications and by a few percent for embedded applications. This thesis also shows that using AEGIS requires only minor modifications to traditional operating systems and compilers.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

It has been my pleasure to work with my advisor, Professor Srinivas Devadas. Srini provided me with an exceptional environment for research, and guided my work. His door was always open and he was never too busy to give advice on my research as well as my personal life. I thank Srini for being not only a good research advisor, but an excellent mentor.

I would like to thank my committee members, Professor Frans Kaashoek, Ronald Rivest, and Krste Asanovic, for their invaluable time and helpful feedback. Meetings with Frans made me think beyond a processor architecture and work on high-level issues such as programming models and detailed application scenarios. Ron provided me with valuable comments on cryptographic primitives used in AEGIS. Krste allowed me to use his computing resources as well as benchmarks in evaluating the AEGIS prototype. I thank Professor Mihir Bellare for discussion on hardware implementations of MAC and hashes. I also thank Professor Arvind and Larry Rudolph for helpful advice throughout my graduate life.

The work described in this thesis would not have been possible without the help of other members involved in the AEGIS project. I heavily collaborated with Dwaine Clarke, Blaise Gassend, and Marten van Dijk on the processor architecture and memory protection algorithms. Jae Wook Lee and Daihyun Lim implemented the PUF circuit in ASIC. Charles O'Donnell was in charge of software aspects of the AEGIS prototype. Ishan Sachdev helped me work out the details of the programming model by implementing the ideas in practice. As a UROP, Kevin Chen helped me implement PUF error correction. I thank my colleagues for their help in research as well as friendship.

I would love to thank my special friend Judy, Jongdeuk, or Joonhee Park for her encouragement and support through tough times. She made my life fun and enjoyable. I was fortunate to have more friends around me. Dan and Dave kept me healthy by playing many squash games together. Nirav, Mike P, Mike Z, Alfred, Albert, and Seongmoo all made CSAIL a great place to work. I also thank Sungwon, Yongguen, and Jongchul as well as all other friends for their support and many fun times we have had together.

My family always supported me during these years. Their prayer, encouragement and love for me is undoubtedly the greatest source of my inspiration and motivation. They have given me far more than words can express.

Most of all, I would like to thank God for His guidance and grace throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As computing devices become ubiquitous, the need for secure and trusted computation is escalating because we place more responsibilities on the devices that surround us. It has become common practice to use devices on the Internet to perform critical operations, such as financial transactions. Computing devices are also expanding into the physical world, controlling everything from home appliances to automobiles. Security breaches may incur not only the loss of private and sensitive data, but can also cause physical damage.

On the other hand, interconnectivity and the proliferation of embedded, portable devices are creating security risks that conventional solutions cannot handle. The Internet connects many parties who may have different interests and motivations to participate in common computation tasks. As a result, the owners of distributed computing devices cannot always be trusted. For example, in distributed computation on the Internet, such as SETI@home, participants are arbitrary computers on the Internet owned by unknown parties. To trust overall computation results, however, a server that distributes computing tasks must be able to trust each participating computer. To further complicate matters, computing elements are becoming disseminated, unsupervised, and physically exposed. For examples, sensor networks can be physically exposed to adversaries. In such systems, physical threats present a significant risk in systems.

To combat this, we propose a single-chip secure processor named AEGIS, which enables users to *trust* the computation on devices even when the owner or operating environment cannot be trusted. In an AEGIS system, the main processor is the only hardware component that needs to be trusted and protected. Although the system may also contain other

components such as off-chip memory, they are protected by the main processor. This approach, we believe, leads to a cheaper and more powerful solution when compared to existing platforms that either require multiple chips to be protected or try to fit all computational resources in a single chip.

To achieve trustworthiness, AEGIS provides a way for users to authenticate the processor hardware and software executing on it, and protects the integrity and the privacy of program execution from both software and physical attacks. To provide such security features using only a single-chip, this thesis presents three key primitives: physical random functions, integrity verification, and encryption. Physical random functions provide a cheap and secure way to authenticate hardware. For secure execution, integrity verification and encryption protect program state in off-chip memory.

The rest of this chapter is organized as follows. First, we describe the security features of the AEGIS processor and explain how the new security problems are addressed. Then, applications are discussed briefly to demonstrate how a trusted platform based on AEGIS can be used in practice. Finally, we compare our single-chip approach to existing trusted platforms and discuss the contributions of this thesis.

1.1 Security Challenges

This section first describes the challenges that AEGIS addresses, and discusses the security features that are required to handle those challenges. Then, this section also points out other aspects of secure computation that AEGIS does not solve.

1.1.1 Trusted Remote Interaction

Figure 1-1 illustrates the main security challenge that AEGIS solves in this thesis. The AEGIS system interacts with a remote party, called Alice, over untrusted networks. In this thesis, we refer to the AEGIS side as *local* and Alice's side as *remote*, looking from AEGIS' point of view. The local AEGIS system, including its storage, is under Bob's control who is an adversary. For example, Bob can install a malicious operating system on AEGIS, probe the memory bus, or even replace some hardware components. Bob may be an owner of the local system or the system may be physically exposed to Bob.

AEGIS addresses the question of how a remote party Alice can "trust" a local system

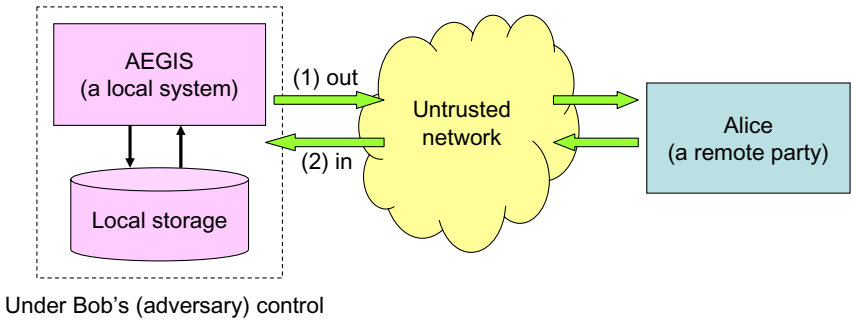


Figure 1-1: The main security challenges addressed by AEGIS.

under Bob's control. More specifically, this thesis focuses on two aspects of "trust". First, Alice must be able to trust the integrity of an outbound message from the local system (marked as (1) in the figure). This means that she knows that the message came from a specific system running a specific software stack without its operations or the message being altered by Bob (or any other adversary). Second, Alice should be able to trust the local system to maintain the confidentiality of secret messages that she sends (inbound messages marked as (2) in the figure). These secrets should be obtained only by the intended system with trusted software, and kept private even in the local storage controlled by Bob.

For this purpose of trusted interaction with remote parties, the AEGIS processor provides the following three additional security features beyond the features provided by conventional systems.

- *Outbound attestation*: To trust outbound messages from a system under adversary Bob's control, a remote party, Alice, must first be able to identify both the hardware and software of the system where the messages originate from. Knowing the identity of the hardware and software, Alice can decide whether the system is trustworthy or not.

- *Secure execution environments*: Knowing the identity of the processor and software is not sufficient for Alice to trust computations on the local system. To be trustworthy, the processor must be able to guarantee that the program cannot be tampered with during execution even by Bob who has physical access to the system. For this reason, AEGIS provides two secure execution environments. First, *tamper-evident (TE) environments* ensure that any physical or software tampering which alters the behavior,

state, or data of a running program will be detected. Second, *private and authenticated tamper-resistant (PTR) environments* protect the confidentiality of a program and its data, as well as detect tampering by any software or physical means.

- *Private storage*: The PTR environment ensures the confidentiality of private information while applications are executing. However, the system should also be able to keep secrets in its local non-volatile storage. For example, protected application code in software licensing and protected digital content in DRM (Digital Rights Management) are often stored locally. For this purpose, AEGIS provides ways to encrypt secrets so that only a specific software application running on specific hardware can decrypt them. This feature can be used by local applications to implement private local storage, or can be used by Alice (a remote party) to send secrets to the local system.

Note that all three security features must be supported by the processor in hardware, and cannot be done in a software-only fashion when the adversary has physical access to the system. For example, if the attestation is performed based on a secret key held as data inside software, the adversary can simply read out this key either from memory or from the hard-disk and impersonate a secure system. The other two functions can be compromised in similar ways if implemented only in software.

1.1.2 Other Security Issues

The above security features enable remote parties (Alice) to trust messages from a system under adversary's (Bob) control, and also allow private information to be sent to such a system with confidence that the secrecy of the information can be maintained. However, there exist a few more security challenges that are important in different application scenarios, yet are *not* handled by the proposed features.

First, an adversary Bob can still install any malicious software on the system that he controls. This is not an issue for a remote party Alice to trust the interaction with the local system. In terms of the integrity, Alice can detect the malicious software using the attestation mechanism, and choose not to trust the messages from the system. Also, malicious software cannot break the confidentiality of secrets that belong to other software because the private storage feature ensures that only specific software can decrypt the

secrets.

However, for systems that simply provide services locally, the fact that a remote party can tell whether a local system is trustworthy or not may not be enough. For example, automobile engine controllers simply generate control signals for an engine and are unlikely to be connected to the Internet. In such cases, it is critical for the system security to ensure that the adversary cannot install malicious software on the system and change the system behavior. For this purpose, AEGIS provides an extension called *secure booting* that restricts software that can execute on the processor.

Second, in some cases such as public terminals, users may locally interact with a computer system that is under a potential adversary's control. If so, the adversary can tamper with local I/O devices such as keyboards, mouse, and displays. For example, keystroke recording or keyboard sniffing can be used to find a user's password. Similarly, the display can be tampered with to show false information. Therefore, in such scenarios, providing secure I/O for local users becomes an important security challenge. However, the focus of this thesis is on securing remote interactions. The issues regarding secure local I/O are *not* discussed here.

Finally, assuming that an adversary has physical access to a compute system, he can carry out denial of service (DoS) attacks. For example, the adversary can simply unplug the power of the system or physically destroy the system. Unfortunately, it is practically impossible to prevent DoS attacks when an adversary has physical control of a system. AEGIS does *not* handle DoS attacks.

1.2 Applications

The AEGIS processor enables critical computations to be performed on a remote device independent of whether the device owner can be trusted or not. Therefore, mistrusting parties on the Internet can collaborate using AEGIS. Also, because even the owner cannot compromise the system, AEGIS can provide strong Intellectual Property (IP) protection guarantees. Finally, the physical security of our processor enables secure operation of embedded and mobile systems even in hostile environments. This section briefly describes example application scenarios to demonstrate the usefulness of the security features in AEGIS.

1.2.1 Digital Rights Management

The advent of large scale sharing of copyrighted media over the Internet has increased the importance of Digital Rights Management (DRM). We are starting to see applications that attempt to enforce simple DRM policies [126]. 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 secure processor. Unfortunately, in software-only DRM mechanisms, 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.

If a computer system uses the AEGIS processor, a stronger form of DRM can be supported. Here, the content provider can be seen as Alice and person who owns the system and purchases the media can be considered as Bob. Using the attestation feature, the content provider can send the media only to a trusted player on AEGIS that has proper DRM mechanisms. Then, the player can protect the media even from the owner using the PTR environment and the private storage feature. The PTR environment does not allow debugging tools to read private data.

Similarly, AEGIS can also support software copy protection where a piece of code gets encrypted so that only a specific processor can decrypt and execute it. Even if an adversary can obtain a copy of the encrypted binary, the binary cannot be executed on other processors.

1.2.2 Distributed Computation

Distributed computation or grid computing is a popular way of solving computationally-hard problems (e.g., `SETI@home`, `distributed.net`) in a distributed manner. A central server partitions an overall problem into small ones, and distributes the small computation tasks to a huge number of machines with different volunteer owners connected via the Internet. Unfortunately, maintaining reliability in the presence of malicious volunteers requires significant additional computation to check the results produced by volunteers.

AEGIS can be applied to this problem by considering the central server to be Alice and each volunteer to be Bob. If each participating computer contains the AEGIS processor, the computation results can be sent back to the central server with the guarantee that a

valid platform has processed data correctly [1] and privately using the TE and PTR environments. Therefore, AEGIS can enable commercial grid computing or utility computing on multitasking server farms, where computation power can be sold with guarantees.

1.2.3 Trusted Third Party Computing

A secure compute server with AEGIS can also be 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 in a way that ensures that no information regarding either the algorithm or the problem instance is leaked. In this scenario, both A and B can be seen as Alice and the owner of the compute server can be considered as Bob. For the computation, A sends her proprietary algorithm code to the secure server encrypted only for that AEGIS processor. Similarly, B also sends her proprietary input data encrypted so that only the trusted wrapper program running on the AEGIS processor can decrypt it. The PTR environment in AEGIS ensures that the confidentiality of both A's algorithm and B's data assuming that A's algorithm has an interface that can only output the computation results, not the input data.

1.2.4 Mobile Agents

A mobile agent is a software program that moves from host to host on the Internet, and performs certain operations on behalf of a user who dispatches it [17]. For example, a user can send a mobile agent that visits many travel websites to find the best airline ticket price. Unfortunately, the hosts may be under the control of an adversary who is financially motivated to break the system and alter the behavior of a mobile agent. In the above example, the owner of a travel website will be motivated to tamper with the mobile agent in a way that his ticket price appears as the cheapest one.

If the hosts on the Internet contain the AEGIS processor, mobile agents can perform sensitive electronic transactions in the PTR environment so that the dispatcher can be sure of the integrity and the privacy of the agents even if hosts are owned by untrusted parties. Here, the dispatcher can be seen as Alice and the host owners can be seen as Bob.

[1] 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.

1.2.5 Embedded and Mobile Devices

Embedded and mobile devices are often physically exposed to potential adversaries causing significant vulnerabilities. For example, nodes in sensor networks often contain a shared secret key and attach a message authentication code (MAC) to outgoing messages to ensure that fake messages from attackers cannot compromise the network. However, attacks can physically capture nodes and extract secret keys. Perrig et al. have noted that this attack is one of the most challenging issues facing sensor networks [101].

Such exposed systems can be secured against physical attacks using the AEGIS processor. In sensor networks, an adversary who captures a sensor node can be seen as Bob, and other nodes and base stations can be seen as Alice. Using the PTR environment and the private storage feature in AEGIS, secret keys can be protected in each node from both software and physical attacks. Even if a node gets captured, an attacker cannot extract the keys. Additionally, base stations can check the integrity of each sensor node using the attestation feature.

The secure booting mechanism in the AEGIS processor can also prevent attackers from changing the software of physically-exposed embedded systems. For example, only the manufacturer should be able to install software on the embedded computers in automobiles. If automobile owners can re-program an engine control system in an attempt to increase the performance, it could result in a safety hazard. Cell phone vendors may also want to control the software on the phone and charge fees for adding new software features to the phone. In game consoles, owners should not be able to change software and bypass copy-protection mechanisms that ensure only legal copies of games, authorized by the vendor, be played. Secure booting ensures that only software that has been authorized by a vendor can execute on the system.

1.3 Approaches for Trusted Platforms

There are many different approaches to achieve the attestation and protection that we require for a secure computing platform. This section briefly discusses existing approaches, points out their limitations, and describes how the single-chip secure processor addresses these limitations.

1.3.1 Tamper-Proof Packages

One conventional approach to building physically secure systems [131, 157] is to encase the entire system in a tamper-proof package. For example, the IBM 4758 cryptographic coprocessor contains an Intel 486 processor, a special chip for cryptographic operations, and memory modules (DRAM, flash, etc.) in a secure package. A secret key is stored in a battery-backed RAM. In this case, all of the components in the system can be trusted since they are isolated from physical access.

This approach can provide a high level of physical security, and also has the advantage of using commodity processors and memory components. However, providing high-grade tamper-resistance can be quite expensive [3] and active intrusion detection circuitry must be continuously battery powered even when the device is off. In addition, these devices are inflexible, for instance, their memory or I/O subsystems cannot be upgraded easily. As a result, this type of tamper-proof package is inappropriate for pervasive computing devices that need to be cheap and flexible.

1.3.2 Smartcards

Another popular form of a physically secure computing system is a smartcard. A smartcard is a single chip that contains an entire computer system including a secret key in EEPROMs, a simple processing core, and memory. This approach provides reasonably high physical security without expensive tamper-proof packaging. Because on-chip components are so small, it is fairly difficult for attackers to read the on-chip EEPROM, or directly change the on-chip processor operations. Smartcards also often incorporate protection circuitry that is active when the chip is powered on to prevent various side-channel attacks.

Only trusting a single chip is an attractive approach because it dramatically reduces the cost of the tamper-proof package. On the other hand, the fact that all system components, including memory, must fit into a single chip limits the computation power of the system. As a result, smartcards are popular for simple operations such as signing transactions on credit cards, but cannot be used for general-purpose computing that requires a fast processor with a large amount of memory.

1.3.3 Auxiliary Security Chips

Recent efforts to build secure computing platforms such as Trusted Platform Module (TPM) from Trusted Computing Group (TCG) [44], Intel LaGrande [52], and Microsoft NGSCB [86] implement security functions in an auxiliary chip which is separate from the main processor. For example, TCG mounts an additional chip (the TPM) next to the processor on the motherboard. Similar to smartcards, this chip is relatively simple and contains an embedded secret key in EEPROMs which can be used to authenticate the platform. The main processor communicates with the TPM through a simple on-board interconnect to perform security functions such as device/software attestation.

Essentially, this approach tries to combine the cost effectiveness of smartcards with the computation power of general-purpose computers. The TPMs are fairly cheap. Since the main processor does not need special structures such as EEPROMs to store secrets, this approach does not affect the cost of the main processor either. Unfortunately, this approach is insecure against physical attacks unless expensive tamper-proof packaging is added. For example, the communication between the main processor and the adjoining security chip (e.g., the TPM) is open to physical probing. Attackers can simply read a plaintext key when it is sent from TPM to the processor. Also, an adversary can tamper with program execution by changing off-chip memory.

1.3.4 AEGIS: Secure Main Processor

In AEGIS, all security features are placed into a single main processor chip. Compared to the other secure computing approaches, this approach enables a cheap, computationally powerful, and secure platform. The platform is cheap because only one processor chip needs to be trusted and protected. However, unlike smartcards, our platform can use off-chip memory to allow powerful general-purpose computing. Finally, for security, off-chip memory is protected by the processor.

1.4 Contributions

This thesis puts forward the AEGIS single-chip secure processor that enables a secure computing platform to be built with a main processor as the only trusted hardware component.

Security mechanism	Function	Cost		
		Average slowdown	Memory space	Logic gates
Physical random function (PUF)	Generate a unique secret (device authentication)	-	-	3K
Encryption (1) Read-only data (2) Read-write data	Protect the privacy off-chip contents (1) encrypt program code (2) protect general data	 (1) 0% (2) 8%	 (1) 0% (2) 6.25%	 90K
Integrity verification (1) Cached hash tree (2) Log-hash	Check the integrity of memory (1) one access at a time (2) a long sequence (> millions)	 (1) 22% (2) 4%	 (1) 33.3% (2) 6.25%	 100K N/A
Suspended secure processing (SSP) + permission checks in MMU	Isolate a secure part of a program from malicious software attacks	-	-	12K
Secure context manager (SCM) + tagging on on-chip cache blocks	Protect secure applications from malicious operating systems	-	-	N/A

Table 1.1: The function and the cost of key security mechanisms in AEGIS. The costs are obtained from simulation studies (performance), or from the RTL implementation (memory space, logic gates). Empty cells indicate that the cost is irrelevant to the mechanism.

1.4.1 Security Mechanisms

To realize the security features of AEGIS on a single chip, this thesis presents new security mechanisms. While the AEGIS architecture combines all these security mechanisms to achieve its security guarantee, the mechanisms are mostly independent of each other and can be separately applied as well.

Table 1.1 summarizes the key security mechanisms developed in this thesis. The table shows the security function of each mechanism along with its costs. The average slowdown shows the performance overhead that the mechanism causes based on simulation studies of SPEC 2000 benchmarks. The memory space overhead is the additional off-chip memory space that is required for meta-data, and given as the percentage of the regular memory space protected by the mechanism. Finally, the logic gate overhead indicates the additional hardware resources used to implement the mechanism. This area overhead is estimated from our own RTL prototype implementation. Empty cells indicate that the mechanism is irrelevant to the cost. The costs in the table only show typical values. The exact overheads can vary depending on each application.

Physical Random Functions

Physical Random Functions (or Physical Unclonable Functions, PUFs) provide a cheap and secure way of generating a unique secret key for each integrated circuit. In AEGIS, the PUF secret is used for the attestation mechanism to uniquely authenticate each processor chip.

The PUF can be used for any device that requires an on-chip secret key such as smartcards, RFIDs, key cards, etc. Because the PUF is infrequently used when generating a secret key, its impact on the processor performance is negligible. Also, the PUF circuit is very small, consisting of only a few thousand gates. In cases where physical security and cost are not important, non-volatile memory such as EEPROM or fuses can replace the PUF.

Off-Chip Memory Protection

Memory encryption and integrity verification mechanisms protect off-chip memory from physical attacks. AEGIS uses the encryption to protect private values in off-chip volatile memory (DRAM). However, the same scheme can be applied to any off-chip memory. For example, if software IP (Intellectual Property) is stored in off-chip flash memory or ROM, the IP can be encrypted so that an adversary cannot copy and execute it on different devices. The encryption module in hardware consumes about 90K gates if the Advanced Encryption Standard (AES) is used. Simpler block ciphers can reduce the size of the module.

The performance and memory space overheads of the encryption mechanisms varies depending on whether the encrypted memory is read-write or read-only. For read-write data, the mechanism requires additional meta-data (time stamps) to be stored in memory along with encrypted data. Typically, time stamps consume 6.25% more memory space in addition to the protected memory space (32-bit time stamps per 64-B block). The encryption degrades the performance because data needs to be decrypted first before being used by the processor core. The encryption of read-write memory incurs 8% slowdown on average, and 18% slowdown in the worst case. On the other hand, the encryption of read-only memory does not require any meta-data, and does not impact the performance; its decryption operations can often be completely overlapped with memory accesses.

This thesis presents two new integrity verification mechanisms, the cached hash tree (`CHTree`) and the log-hash (`LHash`). Both `CHTree` and `LHash` mechanisms check if a value read from off-chip memory is the most recent value written to the address by the processor. `CHTree` performs this check on one memory access at a time, and requires cryptographic hashes to be stored in memory along with protected data. Typically, the `CHTree` scheme consumes 33.3% more memory space for hashes in addition to the protected memory space, and incurs the performance overhead of 22% on average and 52% in the worst case. The `CHTree` module in hardware consumes about 100K gates.

The `LHash` scheme checks a sequence of memory accesses instead of verifying one access at a time. If the sequence is long, say more than millions of memory accesses, `LHash` can reduce the performance overhead down to 4% on average and 16% in the worst case. Unfortunately, `LHash` causes a very high performance overhead when the verified sequence is short. `LHash` requires time stamps instead of hashes to be stored in memory, which consumes 6.25% more memory space for a typical configuration. The size of the `LHash` hardware module is likely to be similar to that of the `CHTree`. However, the exact number is not available because the scheme was not implemented in the RTL prototype.

In AEGIS, the integrity verification mechanisms are applied to off-chip main memory (DRAM). However, these algorithms can be used to protect the integrity of any untrusted storage. For example, even software systems such as peer-to-peer network storage, or databases can be secured using the same `CHTree` or `LHash` algorithms.

Protection against Software Attacks

The thesis discusses two more mechanisms to further secure programs against software attacks. First, suspended secure processing (SSP) and access permission checks in the memory management unit (MMU) allows both operating systems and user applications to be partitioned into secure and insecure parts. The mechanisms isolate the secure part of a program from the insecure part of the same program. Therefore, vulnerabilities in the insecure part cannot affect the secure part.

Second, as a variant of the baseline architecture design that trusts the core part of an operating system, the thesis also presents an architecture that does not trust any part of the OS. The secure context manager (SCM) and on-chip cache tagging mechanisms ensure that a malicious OS cannot tamper with secure user applications. Using these mechanisms, an OS can be excluded from the trusted computing base, therefore bugs in the OS cannot affect a user application's security.

1.4.2 Summary of Contributions

In addition to the processor architecture and the security mechanisms that enable a secure main processor, the thesis also discusses various issues related to building a complete computing system using the AEGIS processor such as operating system support and programming models. Finally, the thesis describes an implementation on an FPGA and evaluates

the overheads associated with AEGIS. The following summarizes the main contributions of this thesis.

- *Integration of Physical Random Functions into a processor*: While PUFs were previously developed and introduced [40, 39], this thesis extends previous work by integrating a PUF into a secure processor. First, the PUF is enhanced with error correction. Second, protocols are developed to express private keys using a PUF. Finally, the thesis introduces an analytical model to understand PUF design issues and addresses how the PUF circuit can be tested after manufacturing.

- *Memory protection mechanisms*: This thesis introduces three new off-chip memory protection mechanisms. First, the cached hash tree is an integrity verification scheme that is secure, yet has an acceptable performance overhead. Second, a new encryption scheme based on counter-mode encryption [75] significantly improves the performance of previous schemes. Finally, the log-hash integrity verification scheme is developed to improve the performance of the cached hash tree for certain types of applications.

- *Secure processor architecture*: We develop two detailed secure processor architectures including both the instruction set and the hardware micro-architecture. The baseline architecture utilizes a security kernel, which is a trusted part of an operating system, to manage user applications. We also present a variant architecture that does not require any trusted software components. While previous work [73] also proposed a secure processor without trusting any part of an OS, the AEGIS architecture fixes security flaws in the previous work. Also, the processor architectures presented in this thesis incorporate new mechanisms to minimize the trusted code base by enabling only a part of an application to be trusted.

- *Programming model*: To be useful in practice, a new architecture must be supported by a high level programming language. This thesis proposes a programming model that exposes the new security features to a high level language. We also describe how the proposed model can be implemented in a compiler. This programming model design has been implemented in the GCC tool-chain as a separate master thesis project [116].

- *RTL implementation*: This thesis presents the first hardware implementation of a single-chip secure processor. Previous studies only performed simulations in software.

There are two key advantages of having a functional RTL implementation. First, the implementation forces the architecture design to be complete and enables us to discuss all practical issues of implementing the secure processor. Second, our evaluation not only addresses the performance overheads, but also shows the silicon area usage. Therefore, this thesis provides an accurate evaluation of overheads associated with the AEGIS processor.

1.5 Organization

The rest of the thesis is organized as follows. Chapter 2 provides the security model and the overview of our processor architecture. Then, the next three chapters (from Chapter 3 to Chapter 5) describe three key components of our secure processor: physical random functions, off-chip memory protection mechanisms, and processor architecture. Given the hardware architecture, Chapter 6 discusses the security kernel in the operating system, and Chapter 7 explains how the new architecture can be programmed in a high-level language. Chapter 8 discusses possible extensions and variants to the baseline design. Chapter 9 gives details on how the architecture can be used for security critical applications. The implementation of AEGIS is described in Chapter 10 and evaluated in Chapter 11. Finally, related work is discussed in Chapter 12 and the thesis concludes in Chapter 13.

Chapter 2

Secure Computing Overview

The AEGIS secure computing platform enables remote parties to *trust* the computation even when the platform is exposed to physical tampering or owned by an untrusted party. To achieve this goal, the platform must protect the integrity and the privacy of applications executing on them, and provide the attestation mechanism so that remote parties can authenticate the hardware and software of the platform. This chapter first describes the overview of the AEGIS architecture. Given the high-level view, this chapter also discusses potential attacks on a single-chip secure processor, the protection mechanisms against them, and the limitations of the current AEGIS design.

2.1 Security Features

This section demonstrates the basic security features of the AEGIS architecture using a simple example. Descriptions in this section can be seen as the overview of AEGIS from the application's perspective.

To illustrate the architecture, let us consider distributed computing on the Internet, where Alice wants Bob to compute something for her, and return the result over the Internet. The pseudo-code in Figure 2-1 represents a simple application sent to Bob's computer. Alice sends the input `x` to Bob, Bob's computer evaluates the function `Func()` for that input, and sends the result back to Alice.

In conventional computer systems, Alice does not know whether Bob's computer has in fact carried out the correct computation because Bob's computer is vulnerable to software and physical attacks. Furthermore, Bob could simply not execute the `Func()` function, or

```
DistComputation()
{
  x = Receive();                    // receive Alice's input

  result = Func(x);                 // compute

  Send(result);                     // send the result
}
```

Figure 2-1: The pseudo-code for distributed computing on traditional systems.

a third party could pretend to be Bob. If Bob is using the AEGIS processor, Alice can be guaranteed that the application has been executed correctly by Bob's computer. We call this a *certified execution*.

Outbound Attestation

To certify that the result is sent by Alice's application on Bob's computer, the processor provides an outbound attestation function that uses the combination of a program hash and an internal processor secret.

In the baseline design, the AEGIS processor has a unique private key that is known only to itself. Using public key infrastructure (PKI), Alice can certify that the corresponding public key belongs to a valid AEGIS processor. For example, the manufacturer of the processor can bootstrap the public key, and provide a certificate by signing the processor's public key with his private key. The processor carries its public key and the certificate, and sends them to Alice. Alice can obtain the manufacturer's public key using the PKI, and verify the certificate for the processor's public key.

When Bob powers up his computer, an operating system starts up and enters a core part called the security kernel using a special AEGIS instruction. At this time, the processor computes the cryptographic hash of the security kernel ($SKHash$), and stores the hash in a secure on-chip register. This hash not only includes the security kernel code and initial data, but also encodes the security kernel's entry point and the status of various protection mechanisms. This hash uniquely identifies the kernel because a good hash function makes it cryptographically infeasible to write another kernel with the same hash value.

In a similar manner, the security kernel computes the hash of a user application when the application enters a secure environment using a security system call. For example, when Bob starts the certified execution application shown in Figure 2-2, the security kernel

```
DistComputation()
{
  x = Receive();                        // receive Alice's input
  result = Func(x);                     // compute

  sig = sys_aegis_pksign(x, result);  // sign the input and the result

  Send(result, sig);                    // send the result
}
```

Figure 2-2: The pseudo-code for certified execution.

computes the hash of the application ($AHash$), which includes the code and initialized data for `DistComputation()`, `Func()`, `Receive()`, and `Send()`.

To have Alice remotely authenticate the security kernel and the application, the application uses the `sys_aegis_pksign` system call. In `DistComputation()`, the `sys_aegis_pksign` system call generates a digital signature (`sig`) of a string that contains Alice's input (`x`) and the computation result (`result`), signed by the processor's private key. Besides the input string, the security kernel includes the application's hash $AHash$, and the processor includes the hash of the security kernel $SKHash$ in the signature. Therefore, modifying either the application or the security kernel, or executing the application on a different processor will cause `sig` to change.

Since Alice knows the processor's public key, from the signature, she can verify that an application on Bob's computer has sent `x` and `result`. She can also check whether the result is from her own application running with a trusted security kernel by comparing two received hashes $SKHash$ and $AHash$ with the ones she has. Alice knows the hash of her own application. For the security kernel, she can compare $SKHash$ with the hash that vendor makes public.

2.1.1 Secure Execution Environments

Program hashes confirm the correct initial state of an application, however, in order to certify the final result, the processor must also guarantee that the application and the security kernel have not been tampered with during execution. More specifically the application's state consisting of registers and the virtual memory space must be protected from both software and physical attacks.

For this purpose, the AEGIS architecture provides two secure execution modes in addi-

tion to the standard (insecure) execution mode.

- **Tamper-Evident (TE) mode**: TE mode ensures the integrity of program state by *detecting* any tampering that alters the program state.
- **Private Tamper-Resistant (PTR) mode**: PTR mode additionally provides privacy as well as tamper detection. Both instructions and data can be kept as private.

Each application starts in the standard execution mode, and enters TE/PTR mode using the `sys_aegis_enter()` system call. For example, in `DistComputation`, the start-up code can enter TE/PTR mode before jumping to the main `DistComputation` function. Once in TE/PTR mode, the application's state is protected not only from other processes, but also from insecure parts of itself and physical tampering. The application can switch between TE and PTR modes depending on its security requirements.

To protect the secure execution modes, AEGIS provides four protected memory regions within each application's virtual memory space, which provide security guarantees even against physical attacks.

1. Read-only Verified memory region
2. Read-write Verified memory region
3. Read-only Private memory region
4. Read-write Private memory region

The Verified memory regions can be modified only by the application that owns the virtual space when it is within a secure execution mode (TE/PTR). The processor and the security kernel protect the integrity of these regions from both software and physical attacks. Similarly, the Private memory regions provide the privacy guarantee that the application can read them only in PTR mode. Because the processor encrypts values within the Private regions when storing them in off-chip memory, even physically probing the data bus does not reveal the values to an adversary.

The application specifies the protected memory regions when it enters a secure execution environment. Given the protected regions, the applications can decide a proper protection level for its code and data, and map them to appropriate regions. For example,

in `DistComputation`, the variables for I/O functions such as `Receive()` and `Send()` can be mapped to an unprotected region so that a DMA engine can read/write them. On the other hand, `result` can be placed in either the read-write Verified region or the read-write Private region depending on whether the result is confidential or not.

2.1.2 Suspended Secure Processing (SSP)

With the attestation mechanism and the protection on the application's execution state, Alice can trust Bob's result even when she cannot trust Bob or when his time-shared computer is in a hostile environment. However, in this approach, the entire application must be trusted and protected (that is, run in a TE or PTR execution environment). Unfortunately, verifying a large piece of code to be bug-free and "secure" is virtually impossible. Moreover, the vast majority of applications today are developed through the use of libraries that cannot be verified by the end-developer. It would therefore be beneficial to separate the unverified code from the security-critical code, and to run the unverified code in an insecure execution environment which cannot compromise the trusted regions of code.

This separation also reduces any overheads incurred from protecting an entire application. Looking at the certified execution example, the I/O functions `Receive()` and `Send()` do not need to be trusted or protected for certifiable execution. Any tampering of `x` within `Receive()` will be detected because `x` is included in the signature. Similarly, any tampering with `result` within `Send()` is detected. Thus, the processor does not have to pay any overheads in protecting such I/O functions. In fact, it is a common case that only a part of application code needs to be protected to garner the security requirements of the application as a whole.

The AEGIS processor supplies a special execution mode, called Suspended Secure Processing (SSP), where the application can temporarily suspend a secure (TE/PTR) mode and execute untrusted code. SSP mode is similar to the standard mode in a sense that it is an insecure mode that does not have permission to tamper with the protected memory regions. However, SSP mode allows an application to re-enter TE/PTR mode without its hash being re-computed.

For example, in the `DistComputation()` function that runs in TE or PTR mode, the application can suspend to SSP mode on a function call to `Receive()` or `Send()` as shown in Figure 2-3. After the function call, the application returns to the original TE/PTR mode.

```
DistComputation()
{
  x = TO_SSP(Receive());              // receive Alice's input
  result = Func(x);                   // compute

  sig = sys_aegis_pksign(x, result);  // sign the input and the result

  TO_SSP(Send(result, sig));          // send the result
}
```

Figure 2-3: The pseudo-code for certified execution with SSP mode.

Because `Receive()` and `Send()` run in SSP mode where they cannot tamper with other parts of the application, now the program hash only needs to include `DistComputation()` and `Func()`.

2.1.3 Other Security Features

The AEGIS architecture provides two additional security features that are not shown in the certified execution example. First, AEGIS provides a private-key decryption primitive, which applications can use to securely store private information in local non-volatile storage. The `sys_aegis_pkdecrypt()` system call gets an input string that contains the security kernel's hash $SKHash$, the application's hash $AHash$, and a message M encrypted with the processor public key. This system call decrypts the input string and returns the decrypted message to an application only if the current program hashes for the security kernel and the application match the ones in the input string. Therefore, anyone with the processor's public key can encrypt private data in a way that only a specific application with a specific security kernel can decrypt it.

Second, the processor is equipped with a hardware random number generator [57, 94, 102]. Both the security kernel and user applications can use the `l.aegis.random` instruction to obtain a 32-bit random number. This random number can be used to generate a secret key, or to prevent replay attacks on network communications.

2.2 AEGIS Secure Computing Model

This thesis considers a multi-tasking computer system that is built around a single processor chip with external memory and peripherals such as hard-disk and I/O devices. The AEGIS

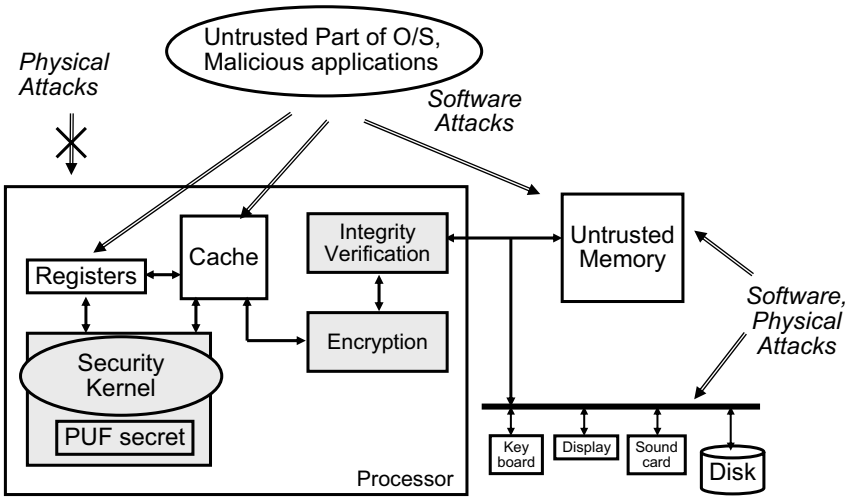


Figure 2-4: The AEGIS security model.

design in this thesis does not support shared-memory multi-processor systems.

Figure 2-4 illustrates AEGIS' approach to build a secure computing system. Briefly, *all* security features are placed into the main processor chip. This secure processor is trusted and protected from physical attacks *whenever it is powered on*, so that values of on-chip volatile memory such as registers and caches cannot be changed or observed by an adversary. The integrity and the confidentiality of on-chip volatile state is guaranteed against physical attacks. Additionally, it is assumed that an adversary cannot change the processor's internal hardware structure without destroying it.

The main processor chip is also protected against non-invasive physical attacks such as fault injection attacks or side-channel (covert channel) attacks. For example, in conventional systems, on-chip secrets can leak via side-channels such as memory access patterns [159] or power supply voltage [62]. Here we assume that either applications are written with techniques to prevent information leaking [1, 43], or the processor is equipped with mechanisms that are commonly used in today's smartcards to prevent side-channel attacks.

The AEGIS architecture in this thesis is built upon the assumption that the processor chip is already protected from the attacks described above. Therefore, AEGIS does not provide protection mechanisms against these attacks. In cases where an adversary is weak and incapable of carrying out such physical attacks, a processor chip may be secure enough as it is without additional protection measures. For applications that face strong adversaries, there exist many protection mechanisms that can be applied to secure a processor chip. The

related work chapter summarizes existing protection mechanisms.

There is one form of physical attack on the main processor that this thesis considers. An adversary can carry out an invasive attacks while the power is off, and try to read out private non-volatile information from on-chip. For example, the adversary can find out the layout of the processor to extract a non-volatile secret key embedded in on-chip ROM or extract a non-volatile EEPROM-based key. Thus, AEGIS must provide protection against such attacks.

Unlike the main processor chip, all off-chip components, including external memory, hard-disk, and I/O devices, are assumed to be insecure. They may be observed and tampered with at will by an adversary. Therefore, the processor architecture must provide protection mechanisms that ensure the integrity of program state in off-chip memory and the confidentiality of private values stored off-chip.

To provide secure multitasking functionality to user applications, a core part of the operating system, called the *security kernel*, is identified by the processor and trusted. The security kernel operates at a higher security privilege level than the regular operating system, and is given custody of mechanisms that protect itself and user applications from software attacks.

2.3 Protection Mechanisms

There are several security features that are required for computing systems to achieve the goal of enabling trusted interaction with remote parties under both software and physical attacks. This section summarizes these security features and provides an overview of the protection mechanisms to achieve the security features.

2.3.1 Secret Key

Before being deployed, a secure system must contain a unique private key and share the corresponding public key with a trusted party so that the system can be authenticated in the field. If an adversary can obtain this private key, he can impersonate the AEGIS processor. Physical Random Functions (or Physical Unclonable Functions, PUFs) provide a way to securely store a private key even when attackers can carry out invasive attacks while the processor is turned off.

2.3.2 Secure Execution

In the field, the first job of a secure system is to guarantee the secure execution of both the security kernel and user applications. The programs' behavior should not be altered by an adversary (integrity), and their secret information should not leak (privacy). At the same time, the system must provide multiple execution modes with varying security levels within a process so that only a small part of the code needs to be trusted. This secure execution consists of three aspects: secure start-up, program state protection during an execution, and secure mode transition.

At a start-up, the AEGIS system *identifies* the program's initial state by computing a cryptographic hash of it. Later, this program hash is combined with the private-key signature so that interacting entities can authenticate the program identity and ensure that the program has started at a desired location with appropriate protections.

During an execution, the program state must be protected in registers, on-chip caches, off-chip memory, and secondary storage such as hard-disk. AEGIS protects state in on-chip registers and secondary storage by having the security kernel manage multitasking and swapped out pages. On the other hand, caches and off-chip memory are protected in hardware by the processor. The memory management unit (MMU), which traditionally isolates each process using the virtual memory (VM) mechanism, performs additional permission checks to ensure that programs in a low security mode cannot tamper with high security memory regions. Off-chip memory is additionally protected against physical attacks by special hardware mechanisms called integrity verification and encryption.

The AEGIS architecture provides an ability for a program to suspend secure processing and execute untrusted code at a low security level. To ensure secure execution, the system must carefully manage the transition between the high security level and the low security level.

2.3.3 Secure Communication

Secure execution of a program is useful only if the program can communicate securely with another system. There are three aspects of secure communication that this thesis considers: authenticity, privacy, and freshness. For private and authentic communication, the processor provides digital signature and decryption operations with its private key. To

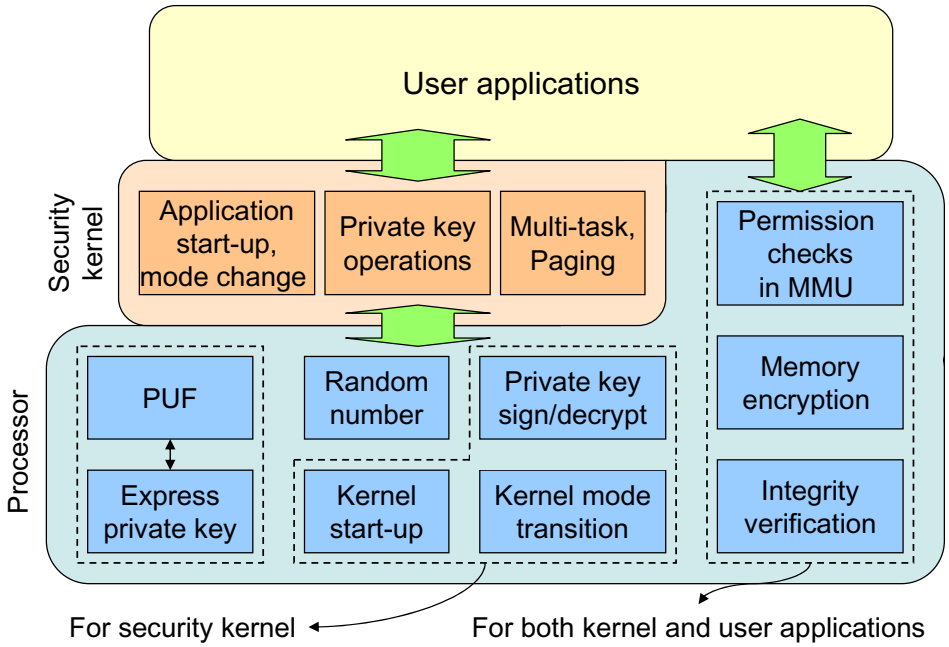


Figure 2-5: Security components of the AEGIS system and their partition between the processor and the security kernel.

prevent replay attacks, the processor provides a hardware random number generator.

2.3.4 Hardware Software Partition

Figure 2-5 summarizes the security components in the system, and illustrates how they are partitioned between the processor and the security kernel. The PUF hardware and the mechanism to store a private key using a PUF are inside the processor. Special PUF instructions are provided to the security kernel so that the kernel can obtain an encrypted private key to bootstrap the system, and reload the private key afterwards.

The security features for secure execution and communication must be provided for both the security kernel and user applications. In the AEGIS architecture, the processor protects the security kernel by managing the secure start-up process, mode transitions, and memory protection during execution. The processor also provides private key operations to the security kernel. For user applications, the security kernel manages most security functions such as identifying the applications with program hashes at the start-up, changing the security mode, protecting program states for muti-tasking, and providing private key operations. The only exception is memory protection. Since each memory operation must

be checked, the processor protects both on-chip caches and off-chip memory for both the security kernel and user applications.

2.4 Limitations

The processor architecture described in this thesis has some limitations. The security model assumes that the processor chip is protected against most physical attacks such as side-channel attacks, or invasive attacks that attempt to change the processor hardware or read out on-chip volatile memory while the power is on. The AEGIS architecture does not present protection mechanisms against these physical attacks.

The processor architecture also does not handle denial of service attacks, replay attacks on local non-volatile storage, and attacks that exploit software bugs.

- *Denial of Service (DoS) attacks*: The security model assumes that attackers have physical access to a computer system. Unfortunately, given this assumption, it is practically impossible to prevent attackers from carrying out denial of service attacks. (In the worst case, an attacker can simply destroy or unplug the computer.) As a result, the AEGIS architecture does not prevent DoS attacks.

- *Replay on non-volatile storage*: AEGIS can guarantee the integrity of both off-chip memory and local non-volatile storage while an application is running. However, unless there is help from a remote trusted party, the processor alone cannot guarantee the freshness of local non-volatile storage once the system is rebooted. The processor needs to be equipped with a physically-secure timer (or a one-way counter) to prevent replay attacks on the non-volatile storage.

- *Attacks exploiting software bugs*: Malicious software attacks often exploit software bugs such as buffer overflows or format string vulnerability. AEGIS ensures that bugs in an insecure part of an application cannot affect a secure part of the application. However, the processor does not prevent an application from being compromised if there is a bug in the secure part of the application.

The baseline AEGIS architecture also does not address security issues related to secure booting or local I/O channels such as keyboards, mouse, and display. However, as discussed in the introduction, these challenges are irrelevant to the main focus of this thesis, which

is enabling trusted remote interaction. For secure booting, the processor can be modified with an extension described in Section 8.1.

Chapter 3

Physical Random Functions

This chapter introduces a primitive called a Physical Random Function (or Physical Unclonable Function, PUF), which enables each processor to generate a unique secret key in a *cheap* and *secure* manner. In AEGIS, the PUF is used to express a unique private key that is used for attestation as well as decryption of private data. First, we briefly discuss existing memory technologies and their problems. Then, PUFs are introduced and evaluated. Finally, we describe how to strengthen the PUF and address practical issues when designing and implementing the PUF.

3.1 Goals

The goal in this chapter is to provide each secure processor chip with a unique secret key that is known only to that processor. For AEGIS, the goal is to provide a private key in a way that a trusted party such as a manufacturer can obtain the corresponding public key.

To be secure, an adversary should not be able to obtain the secret key even if he can perform invasive attacks while the processor is powered off. An adversary can open a package and observe the internal structures and electrical charges. As noted in the previous chapter, however, this thesis assumes that on-chip components are protected from physical attacks while the processor is powered up. Also, this chapter uses a PUF in a way that secret keys from the PUF never leave a chip.

In addition to the security, it is also crucial for the cost of the secure processor that embedding a secret does not adversely affect the manufacturing yield of the processor. Unlike auxiliary security chips, the main processors are fairly expensive to manufacture.

Therefore, lower yields will incur significant costs.

3.2 Non-Volatile Memory

Conventional solutions to embed secret keys in computing devices rely on non-volatile memories such as ROMs, fuses, and EEPROMs. Here, a processor can be equipped with on-chip non-volatile memory, which is programmed to contain a key that needs to stay secret. In this section, we consider each type of non-volatile memory as a candidate to embed secret keys in the processor, and discuss why a new solution is needed.

3.2.1 ROM

ROMs (Read-Only Memory) provide a cheap and reliable way of storing non-volatile information by hard-wiring each bit to either one or zero. In fact, ROMs would serve as a perfect solution if the goal is to store a common public key.

However, the ROM's disadvantages arise from its inflexibility. The information to be stored in the ROM must be determined before the fabrication, cannot be changed afterwards, and must be *common* to all processors that are fabricated with the same mask. As a result, ROMs can only be used for a key that is common to many processors and does not change over the life time of the processors. Moreover, ROMs are inappropriate for storing a secret key because the key can be easily read from the mask.

3.2.2 Fuses

Fuses store a desired value in non-volatile fashion by properly disconnecting selected wires. For example, IBM's eFuse technology exploits electro-migration to "break" a link. Fuses can be fairly cheap as advanced fuse techniques do not require new materials and use only one additional mask during fabrication. Also, unlike ROMs, fuses on each processor can contain a unique secret key.

While fuses provide a possible option to store a secret key on the processor, they have some limitations. First, fuses are one-time programmable. As a result, the secret key cannot be changed once it is programmed. If a key gets compromised by any means, the device cannot be programmed with a new key. Second, fuses are not as reliable as ROMs, and require testing after programming. In some cases, a blown link may even heal itself

over a long period. Finally, fuses can be vulnerable to attacks that alter the stored key by disconnecting additional unblown fuses.

3.2.3 EEPROM

EEPROM (Electrically Erasable Programmable Read-Only Memory) is non-volatile memory that can be erased and reprogrammed repeatedly through the application of higher than normal electrical voltage. Therefore, the processor can store its unique secret key in EEPROM, and even change it later.

The main problem of using EEPROM in the main processor, however, is its cost. On-chip EEPROM requires more complex fabrication processes (typically, 6-8 additionaly masks) compared to standard digital logic, which can adversely affect the manufacturing yield. As a result, in practice, EEPROM is used for small specialized chips such as smartcards or TPMs (Trusted Platform Modules), but not for main processors.

3.2.4 Common Security Issues

In addition to issues of flexibility, reliability, and cost, all non-volatile memory suffers from inherent security weaknesses. First, digital keys stored in non-volatile memory are vulnerable to physical attacks [3]. Motivated attackers can remove the package without destroying the secret, and extract the secret key from the chip. Unfortunately, protecting against such attacks is expensive and requires systems to be continuously powered because the secret key exists in digital form, which is relatively easy to read out, even when the computing device is turned off.

Also, both EEPROM and fuse storage need to be initially programmed and tested by a *trusted party* at a *secure location* before deployment. In practice, these restrictions increase provisioning costs.

3.3 Physical Random Functions

We can avoid the limitations of the existing non-volatile memory schemes by extracting secret keys from a complex physical system rather than storing them in non-volatile memory. This approach is called a Physical Random Function (or Physical Unclonable Function, PUF) [40].

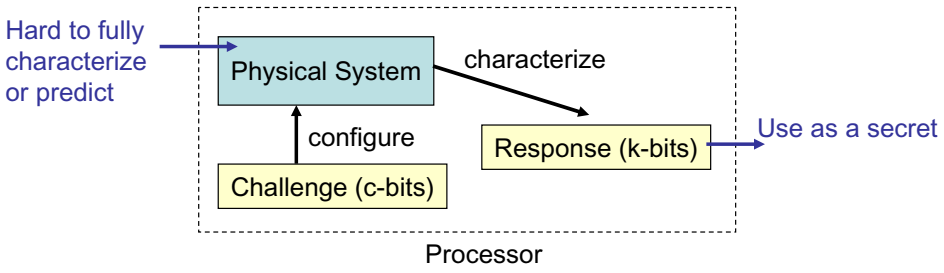


Figure 3-1: The overview of Physical Random Functions.

3.3.1 Concept

Figure 3-1 illustrates the PUF approach. The secure processor has a complex physical system (a PUF) whose properties are unique for each instance, and difficult to predict. To obtain a secret key, the processor configures the PUF using the input, called a challenge, and measures the property of the physical system. This result is called a response. Given that each instance of the physical system has unique properties, which are difficult to predict, the responses can be used as a secret key.

More formally, the PUF is a function that maps a challenge C to a response $R = PUF(C)$ based on an intractably complex physical system. (Hence, this static mapping is a "random" assignment.) The function can *only* be evaluated with the physical system, and is unique for each physical instance. In the following discussion, we assume that the challenge is 128 bits ($c = 128$), and the response is k bits.

While a PUF can be implemented with various physical systems, we use standard integrated circuits as the physical system, and the hidden timing and delay information as the property to generate responses. This type of PUF is called a silicon PUF (SPUF). Even with identical layout masks, the variations in the manufacturing process cause significant delay differences among different ICs. Because the delay variations are random and practically impossible to predict for a given IC, we can extract secrets unique to each IC by measuring or comparing delays at a fine resolution.

3.3.2 Arbiter-Based PUF

Figure 3-2 illustrates an example silicon PUF delay circuit. While this particular design is used to demonstrate the PUF concept, note that many other designs are possible. The

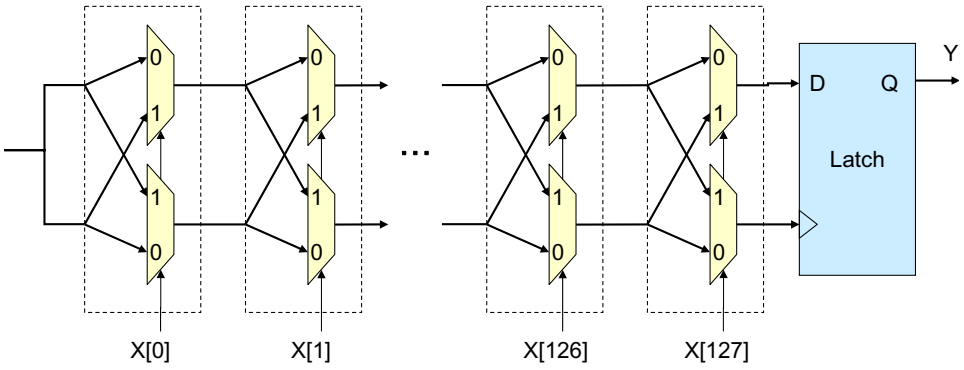


Figure 3-2: A silicon PUF delay circuit. The circuit creates two delay paths with the same layout length for each input X, and produces an output Y by measuring which path is faster.

circuit has a multiple-bit input X and computes a 1-bit output Y by measuring the relative delay difference between two paths with the same layout length. The input bits determine the delay paths by controlling the MUXes. Here, a pair of MUXes controlled by the same input bit $X[i]$ work as a switching box (dotted boxes in the figure). The MUXes pass through the two delay signals from the left side if the input control bit $X[i]$ is zero. Otherwise, the top and bottom signals are switched. In this way, the circuit can create a pair of delay paths for each input X. To evaluate the output for a particular input, a rising signal is given to both paths at the same time, the two signals race through the two delay paths, and the arbiter (latch) at the end measures which signal is faster. The output is one if the signal to the latch data input (D) is faster, and zero if the signal to the latch clock input is faster.

There are two ways to construct a k-bit response from the 1-bit output of this PUF delay circuit. First, one circuit can be used k times with different inputs. The challenge C is used as a seed for a pseudo-random number generator (such as a linear feedback shift register). Then, the PUF delay circuit is evaluated k times, using k different bit vectors from the pseudo-random number generator as input X to configure the delay paths.

It is also possible to duplicate the single-output PUF circuit itself multiple times to obtain k bits with a single evaluation. In this case, the challenge C can be directly used as the circuit input X. As the PUF circuit requires a small number of gates, the duplication incurs a modest increase in gate count.

3.3.3 Experimental Results

This arbiter-based PUF circuit with 64 stages has been fabricated and tested in TSMC's 0.18μm, single-poly, 6-level metal process [67]. The experimental results show that two identical PUF circuits on two different chips have different outputs for the same input with a probability of 23% (inter-chip variation). On the other hand, multiple measurements on the same chip are different only with 0.7% probability (measurement noise or measurement error rate).

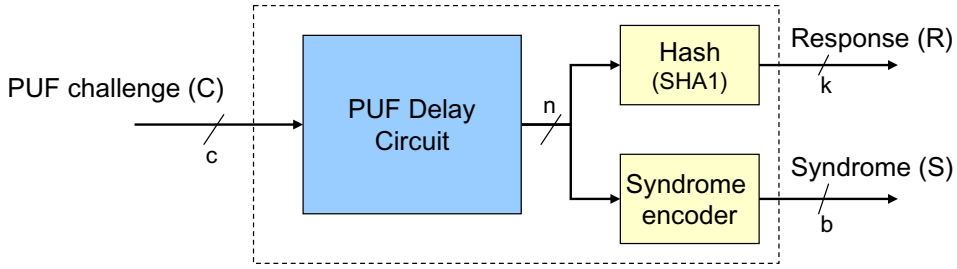
Because the circuit measures the relative delay difference, the PUF is robust against environmental variations. For realistic changes in temperature from 20 to 70 Celsius and regulated voltage changes of $\pm$2%, the output noise is 4.8% and 3.7%, respectively. Even when increasing the temperature by 100C and varying the voltage by 33%, the PUF output noise still remains below 9%. This variation is significantly less than the inter-chip variation of 23%, allowing for the identification of individual chips.

An ideally symmetric layout of the circuit in Figure 3-2 would increase inter-chip variation to 50%. The circuit fabricated on our test chips has systematic skews in the layout because the wires were auto-routed using CAD tools. Removing the skews with a careful layout would increase the inter-chip variation. Section 3.7 further discusses the issues in implementing a better PUF circuit.

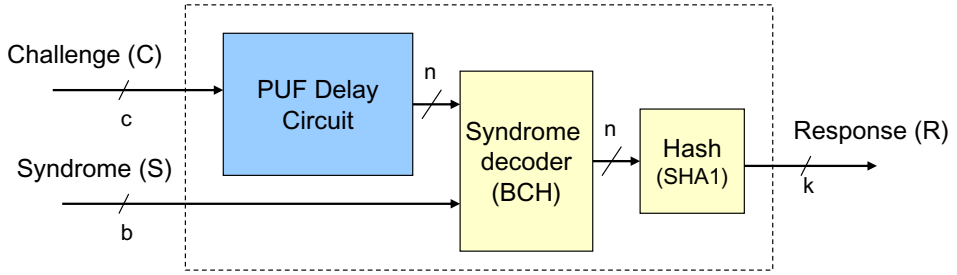
3.4 Reliable Secret Generation

The PUF circuit outputs as described are inappropriate to be used as cryptographic keys. Because of noise, the outputs are likely to be slightly different on each evaluation, even on the same processor and for the exact same challenge C. Cryptographic primitives such as encryption and message authentication codes require that every bit of a key stays constant. Therefore, we need to securely add error correction capabilities to the PUF so that the same secret can be generated on every execution.

Figure 3-3 shows how to apply error correction techniques to the PUF circuit output and enhance the reliability. For error correction, we provide two primitives, one for calibration and the other for re-generation. These primitives are used by the processor to construct higher-level protocols to express a private key or symmetric keys. The first primitive, `puf_calibrate`, gets the challenge C as an input and returns the k-bit response R *and* the



(a) `puf_calibrate`



(b) `puf_regenerate`

Figure 3-3: The reliable secret generation using PUF. The calibration primitive computes the BCH syndrome for future error correction. The re-generation primitive produces the same response again using the syndrome.

b-bit BCH syndrome S computed for the PUF delay circuit output. The PUF response is computed by hashing an n-bit output from the delay circuit. The BCH code is a popular error correcting code that is widely used for binary data and the syndrome is redundant information that allows a BCH decoder to correct errors on the PUF delay circuit output. Effectively, the response generated with `puf_calibrate` becomes a "reference response".

The syndrome here is computed before the output hash function. Because the one-way hash function amplifies errors, the error correction must be done on the PUF delay circuit output, not the hashed response.

The second primitive `puf_regenerate` gets two inputs: the challenge C and a syndrome S. With the syndrome, the processor corrects errors in the PUF delay circuit output, before hashing it to obtain the PUF response. This error correction enables the processor to generate the same PUF response as the previously run `puf_calibrate` primitive.

Unfortunately, the syndrome reveals information about the PUF delay circuit output,

which may be a security hazard. In general, given the b-bit syndrome, attackers can learn at most b bits about the PUF delay circuit output. Therefore, to obtain k secret bits after the error correction, we generate $n = k + b$ bits from the PUF delay circuit. Even with the syndrome, an adversary still needs to guess at least k bits to find the correct PUF response.

The BCH (n, k, d) code can correct up to $(d-1)/2$ errors out of n bits with an $(n-k)$-bit syndrome ($b = n-k$). For example, we can use the BCH (255,63,61) code to reliably generate 63-bit secrets. The processor obtains 255 bits from the PUF delay circuit ($n = 255$), and hashes them to generate the 63-bit response. Also, a 192-bit syndrome is computed from the 255-bit PUF delay circuit output. For some applications, 63-bit secrets may be enough. For higher security, the PUF primitives can be used twice to obtain 126-bit keys.

The BCH (255,63,61) code can correct up to 30 errors, that is, more than 10% of the 255 bits from the PUF can be erroneous and still be repaired. Given that the PUF has a bit error rate of 4.8% under realistic environmental conditions, this error correcting capability provides very high reliability. The probability for a PUF to have more than 30 errors out of 255 bits is 2.4×10^{-6}. Thus, the error correction fails only once in half a million tries. Even this failure only means that the BCH code cannot correct all the errors, not that it will generate an incorrect secret. The probability of a miscorrection is negligible. Therefore, the processor can always retry in case of an error correction failure.

In the above analysis, we assumed that the PUF circuit produces uniformly distributed outputs. If not, more bits should be generated to obtain the same level of security as the 63-bit random key. A paper on fuzzy extractors [31] provide a solution for this case.

3.5 Expressing a Private Key with PUFs

The PUF described so far can generate a unique symmetric key that is only known by a secure processor. However, the AEGIS architecture uses a private key for attestation and decryption of private data. Therefore, the PUF must be used to securely store a private key in a way that the corresponding public key is known to a trusted party.

Figure 3-4 illustrates how a private key can be expressed using a PUF response. The processor generates a private/public key pair using a hardware random number generator. Then, the private key is encrypted and MAC'ed (the message authentication code is computed) using a PUF response as a symmetric key. The encrypted private key can be either

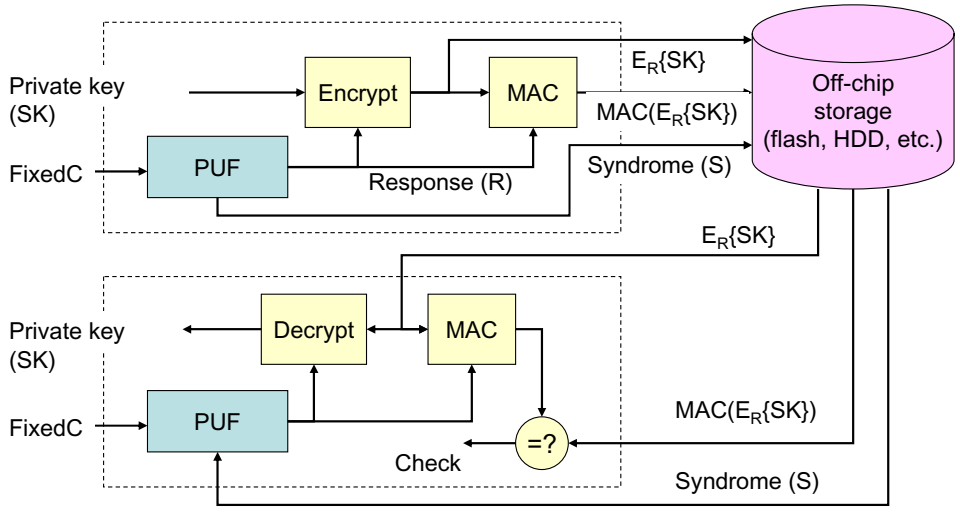


Figure 3-4: The method to express a private key with a PUF. The processor encrypts a private key with the PUF secret and stores the encrypted key off-chip. Only the same processor can decrypt the private key.

stored in off-chip non-volatile storage, which does not need to be protected, or can be distributed with the corresponding public key. Later, only the same processor can re-generate the PUF response, and decrypt the private key.

3.5.1 Key Generation

For the `l.puf.pksave` instruction, the processor first obtain the PUF response and the syndrome (R, S) =`puf_calibrate`$(FixedC)$ using a fixed challenge $FixedC$. At the same time, the processor produces a random private-public key pair (SK,PK) using an on-chip hardware random number generator. This private key is only known to the processor at this time. Then, the private key SK is encrypted using R as a symmetric key to produce $E_R\{SK\}$, and the message authentication code (MAC) of this encrypted key is computed $MAC_R\{E_R\{SK\}\}$. Finally, the processor returns the syndrome S, the encrypted private key $E_R\{SK\}$, the message authentication code $MAC_R\{E_R\{SK\}\}$, and the public key PK.

Using this instruction, a trusted party can obtain the public key for a private key that is only known to a specific processor. For example, a manufacturer can run a program that uses this instruction and outputs the public key and the encrypted private key along with the syndrome and the MAC. The encrypted private key must be stored in non-volatile memory and reloaded before the processor can perform any private-key operations.

3.5.2 Reloading

The second instruction, `l.puf.pkload`, reloads a previously-generated private key into a special on-chip register. The instruction has three inputs: a syndrome S, an encrypted private key $E_R\{SK\}$, and the message authentication code $MAC_R\{E_R\{SK\}\}$. The inputs must be saved from the `l.puf.pksave` instruction and provided to this instruction when the processor boots.

Given the inputs, the processor obtains the PUF response R using a fixed challenge $FixedC$, $R =$`puf_regenerate`$(FixedC, S)$, and checks the message authentication code. If the check passes, the private key is decrypted and put into the special register so that it can be used in the private key operations such as attestation and decryption.

3.6 Evaluation

PUFs are cheap, flexible, and more secure compared to the conventional non-volatile memory because they extract secret keys from standard digital circuits rather than storing them in a digital non-volatile memory. This section evaluates the PUF approach in terms of its functionality, cost, and security.

3.6.1 Functionality

With a PUF, each processor can generate a unique secret key as in the case of fuses or EEPROM. Moreover, by applying different challenges, a PUF can generate many different secret keys. We can have the effect of reprogramming EEPROM by choosing one of the exponential number of secret keys. Finally, by encrypting a secret key with a PUF response and storing in off-chip non-volatile memory, the processor with a PUF can store any chosen secret key. Therefore, a PUF can effectively provide the functionality of on-chip EEPROM without any on-chip non-volatile memory.

3.6.2 Cost

The PUF circuit consists of standard digital logic such as wires, MUXes, and latches. As a result, unlike EEPROM, PUFs do not add additional complexity to the manufacturing process, and do not incur any additional manufacturing cost. At the same time, the PUF

circuit only consumes a few thousand gates, and its logic area overhead is rather negligible for modern microprocessors with millions or billions of transistors.

The only noticeable cost comes from additional controls such as the output hash function and the BCH error correcting code that are added to enhance the basic PUF circuit. Fortunately, all such functions can be implemented in software running on the existing processor core, and require only about 11KB of code. Section 11.2 provides a more detailed evaluation of a PUF's area cost along with other components in the secure processor.

3.6.3 Security

PUFs provide significantly higher physical security than existing non-volatile memory alternatives. First, the processor does not contain any digital secret keys in a non-volatile form as the PUF response is dynamically generated and stored in on-chip registers on demand. To read out the keys in a digital form, an adversary must mount an invasive attack *while the processor is running and using the secret*, a significantly harder proposition.

In the rest of this subsection, we discuss plausible attacks on the PUF and show how the PUF design defeats each of them.

- **Direct delay measurement**: Attackers can open up the package of the secure processor and attempt to measure the PUF delays when the processor is powered off. However, probing the delays with sufficient precision (the resolution of the latch) is difficult and further the interaction between the probe and the circuit will affect the delay. Damage to the layers surrounding the PUF delay paths should alter their delay characteristics, changing the PUF outputs, thereby destroying the secret.

- **Duplication**: Attackers may fabricate the same PUF delay circuit without the processor around it so that they can directly access the PUF responses. However, the counterfeit PUF is extremely unlikely to have the same outputs as the original PUF. The PUF outputs are determined by manufacturing variations that cannot be controlled even by the manufacturers. Experiments show significant (23% or more) variations among PUFs that are fabricated with the same mask, even on the same wafer.

- **Model building**: Attackers may try to construct a precise timing model of the PUF delay circuit to predict the responses. However, model building is impossible because the delay circuit output never leaves the processor chip. Although the BCH

syndrome of the delay circuit output is revealed, it is unlikely that adversaries can infer the circuit output from the syndrome. Even if they can, the processor only uses one BCH syndrome for a fixed challenge, which only encodes one delay circuit output (or a few hundred if the same circuit is used multiple times), which is not enough to construct a timing model.

- **Information leaks in the BCH syndrome**: As discussed in Section 3.4, the BCH syndrome has $n - k$ bits. As a result, an adversary can at most obtain $n - k$ bits out of n bits from the delay circuit. k bits still remain secret.

3.7 Design Considerations

In the previous sections, we have described PUFs and discussed how they can be used to express a private key in the secure processor. While the PUF circuit only contains standard digital logic components, the circuit works in an analog fashion; it compares two path delays. As a result, unfortunately, the conventional design methods for digital circuits such as obeying setup and hold time constraints do not directly apply to the PUF design. This section discusses how to implement the PUF circuit so that it results in high variations and low error rates.

3.7.1 Manufacturing Variations

Before discussing the PUF design issues, let us first briefly study the manufacturing variations, which the PUF circuit exploits to generate responses. Here, our summary is mainly based on previous works [12, 13, 89] that study the impact of manufacturing variations on the digital circuit design.

Variations in integrated circuit performance are mainly due to two sources: environmental factors and physical factors.

- **Environmental variations**: Variations in voltage or temperature can have a significant effect on circuit performance. In a PUF, these variations are the sources of errors. Because of environmental changes, the same PUF may generate different responses over time.

- **Physical parameter variations**: Variations in the manufacturing process can cause physical parameters that characterize the circuit behavior to change. Aging effects such as electro-migration can also change the physical parameters. However, for a PUF, we assume that the aging effects are negligible as the circuit is used very infrequently when compared to the main processor core[1]. Thus, we will refer to this physical parameter variation as manufacturing variation.

The manufacturing variation can further be sub-categorized into die-to-die variations and within-die variations. The die-to-die variations result from lot-to-lot, wafer-to-wafer, and a portion of the within-wafer variations, which are often due to varying processing temperatures, equipment properties, wafer placement, etc. On the other hand, the within-die variations cause differences among components on the same die. Our PUF circuit exploits the within-die variations as we compare two paths on the same die.

The within-die variations consist of two components: systematic and random. The systematic components mainly depend on the design layout, and may eventually be predicted using advanced design tools. The random component is from the sources that cannot be predicted or controlled. For example, the placement of dopant atoms in the transistor channel varies randomly and independently from device to device. Ideally, the PUF circuit should exploit the random components rather than the systematic components.

One study reports the within-die variations (σ) for a 0.25um technology to be about 3 percent [13]. Further, it appears that the within-die variations are increasing as the manufacturing processes advance to smaller feature sizes [89].

3.7.2 Analytical PUF Model

Given this basic understanding of manufacturing variations, let us consider how the PUF circuit in Figure 3-2 works. Figure 3-5 uses a simple probabilistic model to understand the relationships among random variations, systematic skews, and measurement noise. Here, we use a random variable d to model the delay difference between two PUF delay paths (bottom path delay − top path delay) for a fixed input configuration X. Thus, this distribution represents the delay differences and the output Y over many instances of the circuit for a particular input.

[1]Moreover, these aging effects, even if they are appreciable, can be lumped in with environmental variations and corrected.

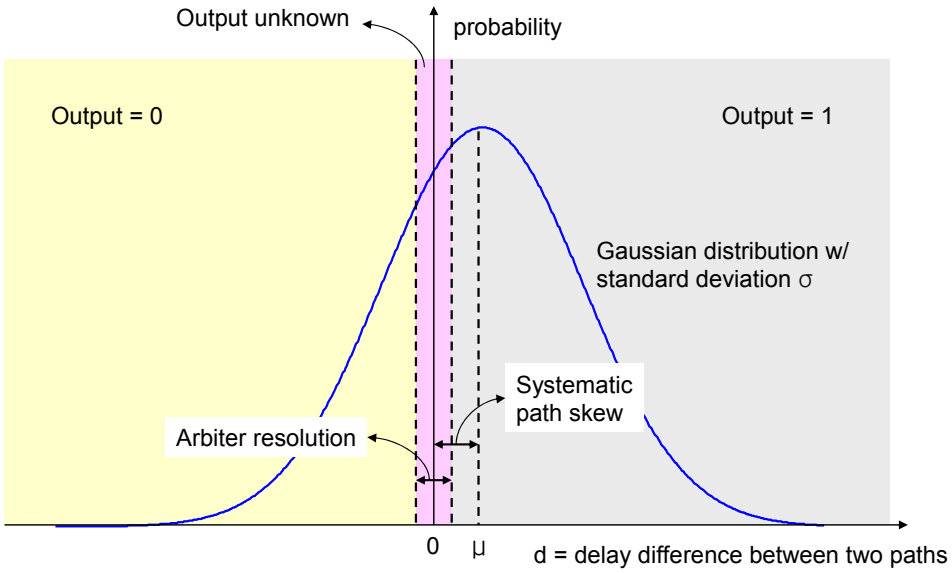


Figure 3-5: A simple probabilistic PUF model.

The delay difference is determined by two components. First, there is a systematic skew that is common to all instances (all chips). The systematic skew includes the systematic manufacturing variations, the skew in the arbiter (latch), and the skew in the layout. This skew is likely to be unique for each input X because it depends on the combination of paths in the 128 stages. Then, there is random variation that can be modeled with a normal (Gaussian) distribution. Therefore, the random variable d can be thought of as a normally distributed variable with the mean μ from the systematic skew and the standard deviation σ from the random variation.

The PUF circuit reliably produces a one if the delay difference d is greater than the resolution of an arbiter r ($d > r$), which implies that the top path is faster than the bottom one by at least the resolution. In the same way, if $d < -r$, the PUF consistently outputs a zero. If the delay difference is less than the resolution of the arbiter ($-r < d < r$), the output could be either one or zero, which will cause inconsistency among measurements up to a 50% error rate.

Given this model, we can estimate both the measurement noise (error rate) and the inter-chip variation for a particular input configuration. First, the measurement noise (e_m) can be estimated by computing the probability of the delay difference to be less than the

arbiter resolution ($-r < d < r$).

$$e_m = 0.5 \cdot P(-r < d < r) = 0.5 \cdot \int_{-r}^{r} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma)^2}. \tag{3.1}$$

Intuitively, from Figure 3-5, the error rate becomes smaller as either the skew (μ) or the random variation (σ) increases. A large systematic skew shifts the graph to one side and reduces the error rate. A large variation makes the graph wider and shorter to reduce the area within the arbiter resolution.

The inter-chip variation for a fixed configuration can be estimated by comparing the probability for an instance to output one ($P(Y = 1)$) and the probability for the output to be zero ($P(Y = 0)$). If we ignore the measurement noise,

$$P(Y = 1) = P(d > 0) = \int_{0}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma)^2}. \tag{3.2}$$

$$P(Y = 0) = P(d < 0) = \int_{-\infty}^{0} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma)^2}. \tag{3.3}$$

Let us define the inter-chip variation for a particular input as the probability that two instances of the PUF circuit have different outputs for that configuration. Then, the inter-chip variation can be estimated as follows:

$$\text{inter-chip variation} = 2 \cdot P(Y = 0) \cdot P(Y = 1). \tag{3.4}$$

Intuitively, the inter-chip variation is maximized at 50% when the output is equally likely to be either one or zero. Therefore, to obtain a high inter-chip variation, the systematic skew μ should be close to zero or the random variation σ should be large.

So far we discussed how the measurement noise and the inter-chip variation can be estimated when the input to the PUF circuit is fixed. If the systematic skew depends on the circuit input, the overall measurement noise and the inter-chip variation for a random input can be estimated by computing the average for many different inputs.

3.7.3 Circuit Design

Ideally, to generate good secrets, the PUF circuit output should be determined only by *random* manufacturing variations, not by other systematic skews. Otherwise, attackers

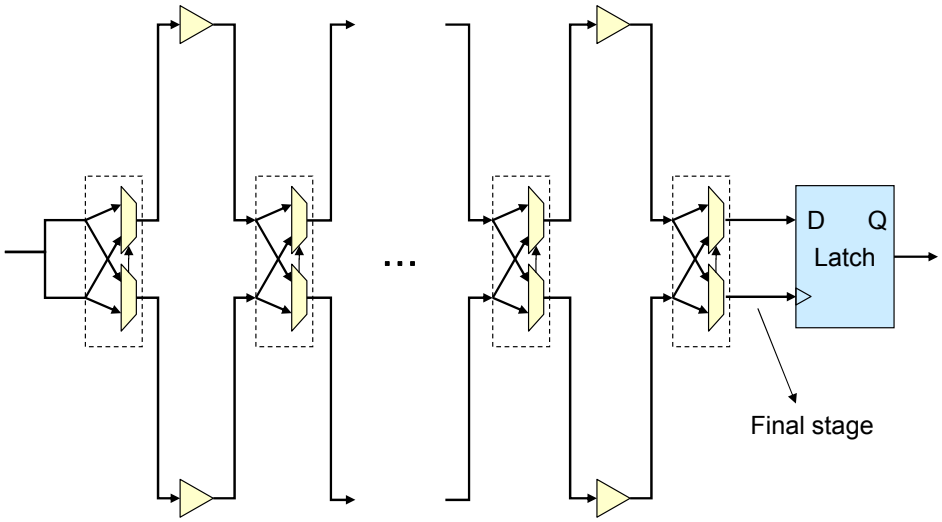


Figure 3-6: The symmetric PUF layout example.

may be able to guess the responses from the systematic components. In other words, the PUF circuit should have high inter-chip variation. Also, in order to reliably regenerate the same secret, the PUF circuit output should have low measurement noise. To achieve this goal, the designer should *minimize the systematic skew* μ and *maximize the random variation* σ in our PUF model.

To minimize the systematic skew, the layout of those two PUF delay paths should be *symmetric* so that both have the same delay in design. Figure 3-6 illustrates a symmetric PUF design. This symmetric layout ensures that two delay paths have the same delay. At the same time, the symmetry helps minimize the impact of the systematic manufacturing variations, which is mainly determined by the layout. By having the same layout pattern, both paths will have the same impact from systematic manufacturing variations. Finally, the wire delays in the final stage can be used to compensate for the skew in the arbiter.

To increase the standard deviation σ, which results in from random manufacturing variation, the length of the delay paths can be increased to have larger delay. Intuitively, long paths can be considered to be a collection of many short paths. Therefore, if the delay of each short path varies independently, the delay of the collection should vary by a larger amount. More formally, if a given manufacturing process results in random variation with standard deviation σ_{unit} per unit length, the σ for the entire path can be represented as $\sigma_{unit} \cdot \sqrt{L}$ where L is the total length of the path.

From Equation 3.1, the designer can compute the required length of the PUF paths L to obtain a certain measurement noise level if the amount of random variation per unit length σ_{unit}, and the resolution of an arbiter r are known. For example, if the measurement noise is desired to be less than 5% and the systematic skew is zero, σ should be about 8 times larger than r. Conservatively, the resolution r should be less than the setup/hold time of the latch, which is often in the order of tens of pico-seconds. Therefore, in this case, the PUF delay paths should be long enough so that the random variation is in the order of hundreds of pico-seconds.

Finally, in addition to having long enough delay paths, two paths should be routed far apart as illustrated in Figure 3-6. Separating the two paths ensures that the electronic coupling between them does not affect the propagation of signals. Also, far apart components are likely to have larger delay difference. In fact, we found from our FPGA experiments that routing the paths right next to each other can significantly reduce the variations in the circuit output.

3.7.4 Error Correction Parameters

Once the PUF circuit is designed, one remaining issue is to obtain the inter-chip variation and the error rates (on the same PUF) of the design so that we can determine the parameters for error correction. A couple of approaches can be taken to address this problem depending on the way that the PUF is implemented.

In a common case, the PUF circuit will be implemented as a hardwired block such that the circuit layout remains exactly the same no matter what the interfaces are. The characteristics of this hardwired PUF circuit can be determined precisely by fabricating test chips that allow direct accesses to the input and output. Using the test chips, we can obtain many circuit input-output pairs from many die, and determine the measurement noise and the inter-chip variation for a given process technology.

In some cases, fabricating test chips may not be a viable solution. For example, the PUF circuit may be designed as a soft-core block with placement and routing constraints for programmable logic such as FPGAs. Or, test chips may be simply too expensive. In such cases, the characteristics of the PUF circuit can be estimated using the model in Figure 3-5. First, the delay of each wire and transistor of the PUF delay path should be obtained from computer-aided analysis tools after synthesis. This delays are used to

estimate the systematic skew for each circuit configuration (X). Then, σ and r must be found for the fabrication technology and the chosen latch. Given these design parameters, we can simulate the PUF circuit and obtain the inter-chip variation and the measurement noise as shown in our model.

3.8 Manufacturing Tests

In practice, the IC fabrication process is not completely reliable and may result in defects. As a result, the PUF circuit must be tested before being deployed. This section briefly discusses how a manufacturer can test a fabricated PUF circuit for correct functionality.

A conventional approach to test the circuit is to allow scan chains to write and read all internal PUF registers so that any test vector can be put in and the results can be read out. While this method allows a flexible way of testing, it causes potential security vulnerabilities. First, the scan chains must be permanently disabled after testing so that adversaries cannot use them to simply read out PUF responses after deployment. This requires irreversible fuses on-chip. Also, even with fuses, the tester must be trusted not to obtain a specific response or model the PUF circuits, unless the PUF circuit can be shown to be hard to model.

Another possibility is to duplicate the PUF circuit many times so that at least one PUF circuit is fully-functional with a very high probability. The PUF circuit outputs are XOR'ed so that variations in one circuit will result in variations in the combined PUF output. In this case, we simply assume that at least one circuit has no defect and do not test the circuits. The disadvantage of this approach is that combining the outputs of multiple PUF circuits artificially increases measurement errors because an error in any one circuit will result in an output error.

In this section, we propose a selective use of scan chains that allows testers to check the PUF circuit against known defect patterns. This approach does not require disabling the scan chains after testing. Also, even the tester cannot model the circuit or obtain a specific challenge-response pair. Here, our approach applies to the case where one instance of the PUF circuit in Figure 3-2 is used multiple times to generate multiple PUF output bits. The approach as is does not apply if there are multiple copies of the PUF delay circuit that generate different bits in the same response.

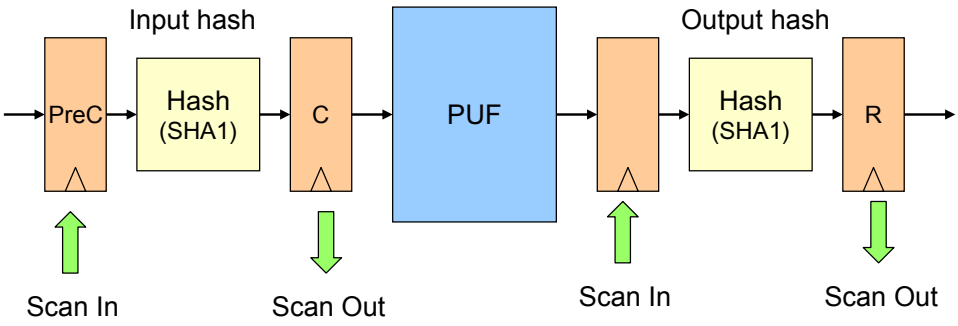


Figure 3-7: The scan chains for the PUF debugging.

Figure 3-7 illustrates our scan chains. First, the PUF circuit has an input hash function that produces the challenge to PUF as well as an output hash function. The input to this hash function is called pre-challenge $PreC$. The input registers of the hash functions are a part of the scan-in chain so that testers can set their values, but are not a part of the scan-out chain. Similarly, the output registers of the hash functions are a part of the scan-out chain, but not a part of the scan-in chain. The hash functions can be easily tested by setting an input and monitoring the output.

On the other hand, for the PUF circuit, testers can only set $PreC$ and monitor R. The input challenge C *cannot* be directly set, and the PUF circuit output *cannot* be directly read out. Because testers can only see the PUF response after the PUF circuit output is hashed, they cannot carry out a model building attack. Also, because they can only set $PreC$, testers cannot set the challenge C to be a specific value. Therefore, testers can only obtain responses for randomly chosen challenges, not for specific challenges.

For testing purposes, this configuration effectively allows the testers to set C to random test vectors and compare the PUF circuit output with known patterns. The challenge C is set to a random test vector by setting $PreC$ to an arbitrary value. Given a PUF circuit output, the test can easily compute the corresponding PUF response. However, there is no correct PUF output for a given input. Therefore, we test the PUF by comparing the PUF output with known *defect patterns* to ensure that it does not have certain defects.

For example, let us consider detecting single stuck-at faults and single gate/wire delay faults. If there is a single defect in one stage of the PUF delay circuit, the circuit output is no longer a random function, but becomes a deterministic function. Say that the defect causes either top or bottom path of one stage to be stuck-at zero or be slow enough to

dominate the total PUF path delay. Then, the PUF circuit output will be always one when that defective segment is a part of the path to the latch clock input, and zero otherwise. The PUF output can be simply computed from the PUF challenge C. Therefore, given a random test vector from a set of $PreC$, testers can compute all defect patterns for single stuck-at faults and single delay faults.

While this approach cannot guarantee that there is no defect, it can detect common defects, which are easy for adversaries to exploit. This, of course, is exactly what conventional testing for standard logic circuits attempts to do. Also, this approach can generalize to multiple faults in the circuit by computing and testing for more defect patterns. Note that adversaries need to discover the defects to exploit them and predict the PUF responses. PUF responses resulting from unknown defects will appear random because of the output hash function.

Chapter 4

Off-Chip Memory Protection

This chapter describes the details of the off-chip memory protection mechanisms. We first give an overview of the protection mechanisms and describe how they fit into the memory hierarchy. Then, the protection algorithms for encryption and integrity verification are explained in detail. Finally, we address practical issues in implementing the algorithms on a processor.

The discussion in this chapter considers four types of protection regions in the physical memory space. Protection regions can be either "IV protected" or "ME protected". The integrity verification mechanism detects any tampering that changes the content of the IV regions, while the encryption mechanism guarantees the privacy of the ME regions. The IV and ME regions can also be either "read-only" (RO) or "read-write" (RW). The ME regions are considered to be within the IV regions.

4.1 Overview

Since software attacks on on-chip caches can be prevented by the permission checks in the MMU, our encryption and integrity verification is only necessary for off-chip memory, which is exposed to physical attacks. The off-chip protection mechanisms are placed between the L2 cache and the off-chip memory as shown in Figure 4-1. As a result, the protection mechanisms operate on cache blocks that are written back to memory or read from memory.

For example, let us say that the cache writes back a data block that is in the IV and ME protected region of memory. First, this data block is processed by the memory encryption (ME) unit, which produces the encrypted data block and meta-data required for future

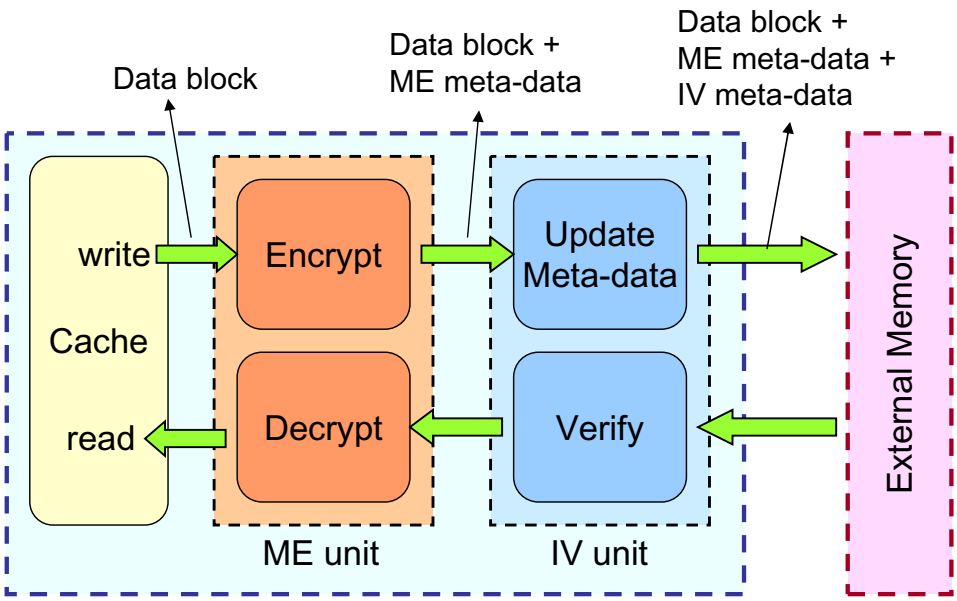


Figure 4-1: The integration of the off-chip memory protection schemes in the memory hierarchy.

decryption. Then, the integrity verification (IV) unit protects both the encrypted block and the ME meta-data, and stores its own meta-data in off-chip memory. If the data block is read by the cache later, the ME unit reads the encrypted data block and the corresponding ME meta-data from memory, which are verified by the IV unit. The ME unit decrypts the block and returns the plaintext data to the cache.

Here, the integrity verification is performed on the encrypted data block and the ME meta-data rather than encrypting data after the IV unit. There are a couple of reasons for this design. First, this ordering allows the integrity verification to work regardless of whether the encryption is enabled or not. In this way, the processor can first ensure the integrity of an application, and set up secret keys for the encryption module based on the identity of the application.

Second, only verifying decrypted blocks can result in a security vulnerability. For example, an adversary can replay an encrypted cache block and its ME meta-data stored in off-chip memory to find out whether the new value is the same as the old value. If blocks are verified after decryption, this replay will result in an integrity verification failure only if the two values are different, indirectly leaking information. On the other hand, if encrypted blocks are verified, this tampering is always detected and stopped (personal communication

with David Mazieres, 2005).

In the following description, we protect off-chip memory on an L2 cache block granularity. While the algorithms can be easily generalized to the cases where multiple blocks are protected together, that change will require accessing multiple cache blocks on a cache miss or a write-back.

4.2 Memory Encryption

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.

4.2.1 Security Goal

The goal of an encryption mechanism is to protect the confidentiality of values stored in off-chip memory. In the AEGIS security model, an adversary can observe and change encrypted data values and ME meta-data stored in memory. However, side channels such as an address bus are assumed to be protected.

4.2.2 Block Cipher

A block cipher is a symmetric-key encryption algorithm that maps a fixed-length block of plaintext (unencrypted text) data into a block of ciphertext (encrypted text) data of the same length. This mapping is based on a secret key provided as an input to the algorithm. Decryption is performed by the inverse mapping and requires the same secret key.

While any block cipher algorithm can be used in the memory encryption algorithm, we use the Advanced Encryption Standard (AES) [90] as a representative cipher. AES is a symmetric-key encryption algorithm approved by the National Institute of Standards and Technology (NIST) as a recommended standard. Also, we use AES with the block size of 128 bits, which means that each AES cipher encrypts and decrypts 128 bits at a time.

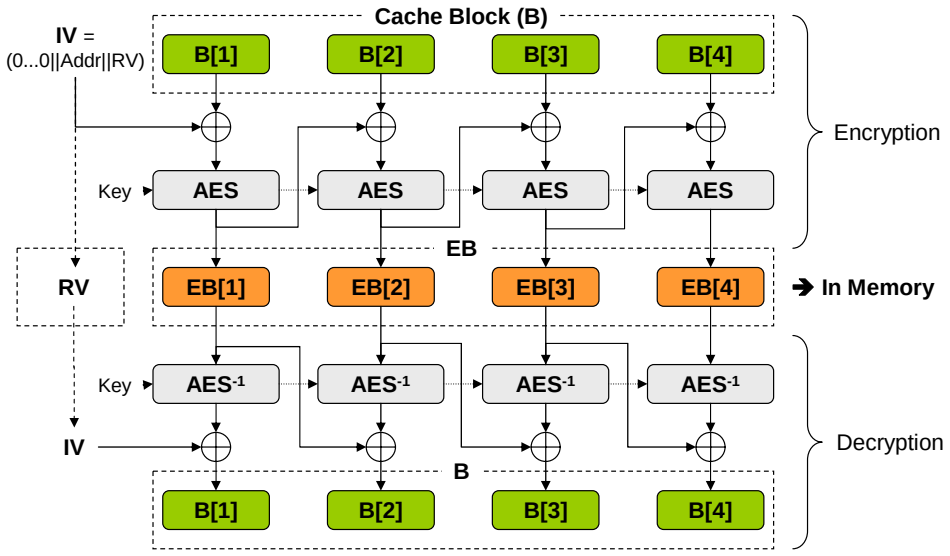


Figure 4-2: Encryption mechanism that directly encrypts cache blocks with AES (CBC mode).

4.2.3 CBC Encryption

The most straightforward approach for encryption is to use an L2 cache block as an input data block of the AES algorithm. For example, a 64 byte (64-B) cache block `B` is broken into 128-bit sub-blocks (`B[1]`, `B[2]`, `B[3]` and `B[4]`), and encrypted by the AES algorithm. Figure 4-2 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 `IV`. Here, `A` $\|$ `B` represents a concatenation of `A` and `B`. `IV` consists of the address of the chunk and a random vector `RV`, and is padded with zeroes to be 128 bits. This prevents adversaries from comparing whether two cache blocks are the same or not. After the encryption, the random vector `RV` is stored in the off-chip memory along with the encrypted data (`EB`).

While this direct encryption mechanism is secure, the main disadvantage of this mechanism is its decryption latency. Because the AES computation can start only after completing a read of the encrypted data block from memory, the memory latency gets increased by the AES latency.

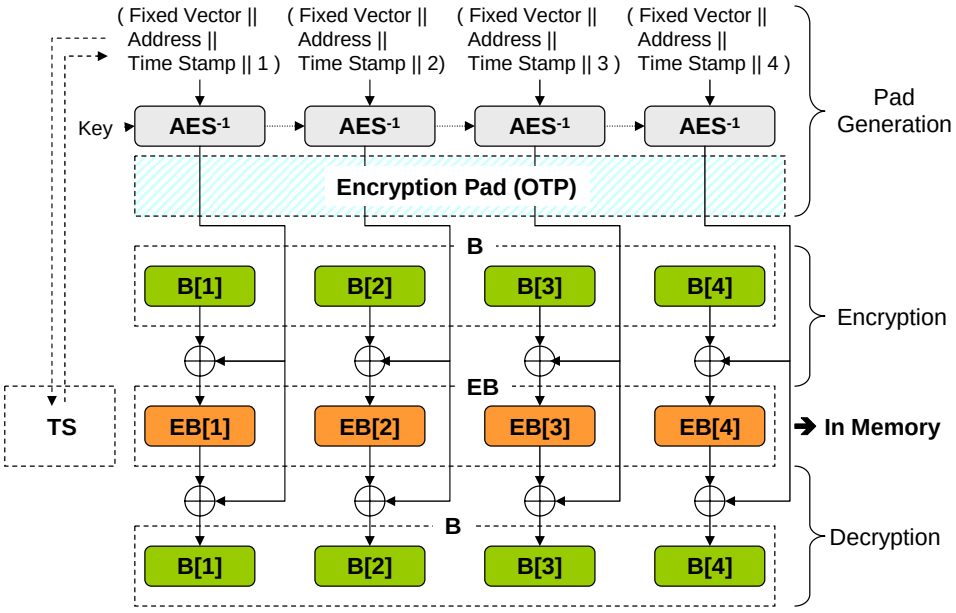


Figure 4-3: Encryption mechanism that uses one-time-pads from AES with time stamps (counter-mode).

4.2.4 Counter-Mode Encryption

To hide decryption latency, this thesis proposes a scheme that decouples the AES computation from the corresponding data access by using one-time pads [3] and time stamps. This new scheme can be seen as an instantiation of counter-mode encryption [75]. This subsection focuses on the algorithm to encrypt and decrypt the read-write ME region. The next subsection will optimize this algorithm for the read-only ME regions.

Figure 4-3 and 4-4 illustrate the scheme. In the figures, a cache block, `B`, consists of four 128-bit sub-blocks (`B[1]`, `B[2]`, `B[3]`, and `B[4]`), and a processor holds a counter TIMER in secure on-chip storage. Initially, the processor sets TIMER to be one, and chooses a random secret key K_{RW} for read-write ME regions. The algorithm allows a different key K_{RO} to be used for read-only ME regions, which can be set by an application.

For a write-back of a dirty cache block to memory[1], the block is encrypted by XOR'ing each 128-bit sub-block `B[i]` with an encryption pad `OTP[i]`. The pad is computed using the AES decryption on `(V || Address || TS || i)` with the secret key K_{RW} for the read-write ME region. `V` is a fixed bit vector that makes the input 128 bits, and can be randomly

[1]If the block that is being evicted is clean, it is simply evicted from the cache, and not written back to memory.

- For an initialization
  1. TIMER = 1.
  2. V = an arbitrary bit vector.
  3. K_{RW} = a random secret key.
- For an L2 cache write-back of a block in the read-write ME region write-back-block(Address, B, K_{RW}):
  1. TS = TIMER.
  2. For each $1 \leq i \leq 4$
     (a) OTP[i] = $\text{AES}^{-1}_{K_{RW}}$(V ‖ Address ‖ TS ‖ i).
     (b) EB[i] = B[i] $\oplus$ OTP[i].
  3. Increment TIMER.
  4. Write EB to Address in memory.
  5. Write TS to get-ts-addr(Address) in memory.
- For an L2 cache miss of a block in the read-write ME region read-block(Address, K_{RW}):
  1. Read the time stamp (TS) from get-ts-addr(Address) in memory (or a time stamp cache).
  2. If TS is equal to 0, K = K_{RO} (for read-only).
     Otherwise, K = K_{RW} (for read-write).
  3. For each $1 \leq i \leq 4$
     (a) Start OTP[i] = AES^{-1}_{K}(V ‖ Address ‖ TS ‖ i).
  4. Read EB from Address in memory.
  5. For each $1 \leq i \leq 4$
     (a) B[i] = EB[i] $\oplus$ OTP[i].
  6. Return B.
- If TIMER reaches a threshold
  1. Select a new key K'.
  2. For each Address in the read-write ME region,
     (a) B = read-block(Address, K_{RW}).
     (b) TIMER = 0.
     (c) write-back-block(Address, B, K').
  3. K_{RW} = K'.

Figure 4-4: OTP (counter-mode) encryption algorithm.

selected by the processor at the start of program execution. `TS` is a time stamp that is the current value of TIMER. Finally, the TIMER is incremented, and the encrypted block and the time stamp are stored in off-chip memory.

To read an encrypted block from memory, the processor first reads the corresponding time stamp from memory. To improve performance, it is also possible to cache time stamps on-chip. Once the time stamp is retrieved, the ME module immediately starts with the generation of the `OTP` using AES. At the same time, the read is issued to memory for the encrypted cache block `EB`. *The pad is generated while `EB` is fetched from memory.* Once the pad has been generated and `EB` has been retrieved from memory, `EB` is decrypted by XOR'ing with the pad.

For the decryption, the algorithm chooses a secret key based on the value of the time stamp. If the time stamp is zero, the key for the read-only ME region (K_{RO}) is used. Otherwise, the key for the read-write ME region (K_{RW}) is used. In this way, encrypted programs can contain private values in the read-write regions as well as the read-only regions.

When the TIMER reaches its threshold, 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^{-1}, thus guaranteeing the unicity of the one-time-pads.

4.2.5 Decryption of the Read-Only ME Regions

For the read-only ME region, the OTP encryption algorithm can be further optimized. Because read-only code and data are only encrypted once, they do not require time stamps. In this case, a constant zero can be used in a place of time stamps for both encryption and decryption. Therefore, the AES computation for decryption can start without loading anything from memory. Figure 4-5 summarizes this optimization for the read-only ME region.

- To encrypt a read-only block `B`
  encrypt-ro-block(`Address`, `B`, K_{RO}):
  1. For each $1 \leq i \leq 4$
     (a) `OTP`[i] = $\text{AES}^{-1}_{K_{RO}}$(`V` $\|$ `Address` $\|$ `0` $\|$ i).
     (b) `EB`[i] = `B`[i] $\oplus$ `OTP`[i].
  2. Write `EB`.
- For an L2 cache miss of a block in the read-only ME region
  read-ro-block(`Address`, K_{RO}):
  1. For each $1 \leq i \leq 4$
     (a) Start `OTP`[i] = $\text{AES}^{-1}_{K_{RO}}$(`V` $\|$ `Address` $\|$ `0` $\|$ i).
  2. Read `EB` from `Address` in memory.
  3. For each $1 \leq i \leq 4$
     (a) `B`[i] = `EB`[i] $\oplus$ `OTP`[i].
  4. Return `B`.

Figure 4-5: OTP (counter-mode) decryption for the read-only region.

4.2.6 Impacts on Memory Latency

The direct encryption scheme serves our purpose in terms of security, however, it has a major performance disadvantage. In Figure 4-6 (a), the AES decryption is performed immediately after each 128-bit sub-block is read. Therefore, if the AES decryption takes 40ns, we will get the decrypted result for the last sub-block, 40ns after the *last* sub-block is read. The decryption latency is directly added to the memory latency and delays the computation. We assume that the latency of any L2 miss is determined by the decryption of the last 128-bit sub-block (`EB[4]`). The total latency may be slightly reduced for accesses to the first sub-block if each 128-bit sub-block is returned separately. However, this optimization does not reduce the additional AES latency added to the memory latency.

In the new scheme, after the time stamp is read, we perform AES computation to generate encryption pads as shown in Figure 4-6 (b). This computation is overlapped with the following bus accesses for the encrypted cache block. After the last sub-block is read, most of the AES computation is done and a processor needs to perform only an XOR to obtain the entire decrypted block. For example, if it takes 80 ns to read the time stamp, and an additional 40 ns for the cache block, we can hide 40 ns of the AES latency.

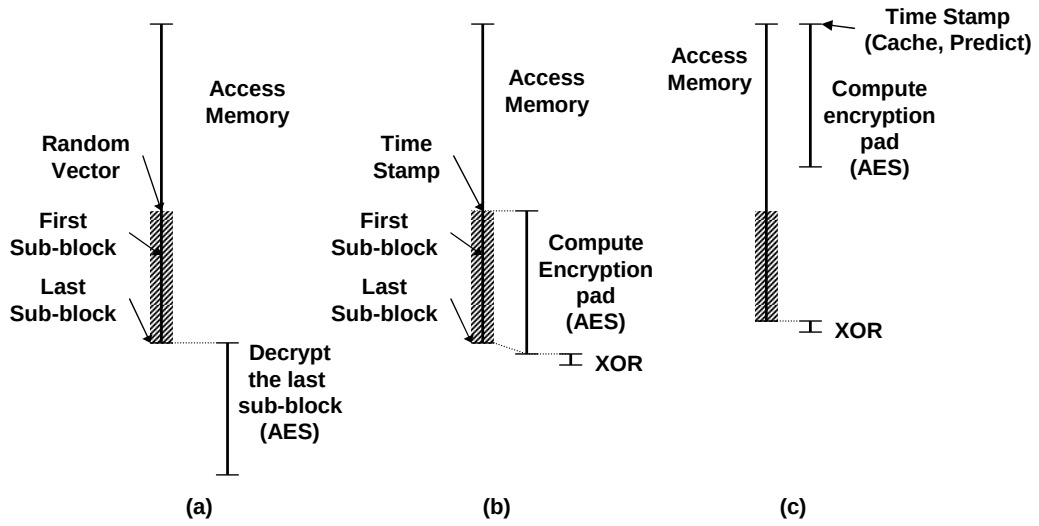


Figure 4-6: Impacts of encryption mechanisms on memory latency. The AES decryption can only be performed after data blocks arrive from memory in the direct (CBC) encryption method. As a result, the memory latency is effectively increased by the AES latency. On the other hand, the OTP (counter-mode) encryption allows the AES computation to be performed in parallel to reading data blocks from memory.

When overlapping the AES computation with data bus accesses is insufficient to hide the entire latency, the time stamp can be cached on-chip or speculated based on recent accesses. Also, the decryption of the read-only ME region does not require a time stamp. In these cases, the AES computation can start as soon as the memory access is requested as in Figure 4-6 (c), and completely overlapped with the long memory accesses.

The ability to hide the encryption latency improves processor performance. Further, it allows the use of a cheaper implementation of the AES algorithm with longer latency.

4.2.7 Security Discussion

The conventional one-time-pad scheme is proven to be secure [3]. Our scheme is an instantiation of a counter-mode encryption [75], which can easily be proven to be secure given a good encryption algorithm that is non-malleable.

4.3 Integrity Verification

This section presents memory integrity verification algorithms, which ensure that the values stored in off-chip memory have not been changed. The following subsections describe two

algorithms. The first algorithm is based on message authentication codes and is used to verify the read-only IV region. The second algorithm is called cached hash trees and is applied to the read-write IV region.

When describing the algorithms, this thesis uses a term *chunk* as the minimum memory block that is verified by the integrity checking. If a word within a chunk is accessed by a processor, the entire chunk is brought into the processor and its integrity is checked. In our instantiation, chunks are identical to the L2 cache blocks. We use a separate term because the IV unit not only verifies the L2 cache blocks, but also checks other meta-data such as the ME time stamps.

4.3.1 Security Goal

The integrity verification algorithms check if the value a processor loads from a particular address is the most recent value that the processor has stored to that address. An adversary can tamper with off-chip memory content including the protected values as well as the meta-data stored by the IV unit. On the other hand, it is assumed that on-chip volatile memory cannot be changed by an adversary.

4.3.2 Message Authentication Code (MAC)

A hash of a message is a fixed length cryptographic fingerprint of the message. It is hard to find two distinct messages with the same hash. This property is called collision-resistance. A message authentication code (MAC) is a hash computed over the message using a secret key and attached to the message; it is often used to authenticate a message. Later, a receiver recomputes the MAC of the received message and compares it with the attached MAC. If it is equal to the attached MAC, the receiver knows that the message it received is authentic, that is, sent by the sender with the valid secret key.

The idea can be simply extended to memory integrity checking for the read-only IV regions, which can contain instructions and read-only data. A processor contains a secret key K_{IV}, which is randomly generated along with the read-write ME key K_{RW}. As shown in Figure 4-7, the processor calls mac-init on an initialization. This procedure computes the MAC of each chunk in the read-only IV region and stores the MAC in memory. When the processor reads back a chunk from the memory on a cache miss, it uses mac-read-check, which recomputes the MAC of the loaded chunk and compares this with the MAC stored

- On an initialization
  mac-init(K_{IV}):
  1. For each chunk at `Address` in the read-only IV region
     (a) Read `Chunk` from `Address` in memory.
     (b) Compute `MAC` = $\mathrm{H}_{K_{IV}}$(`Address` ‖ `Chunk`).
     (c) Write `MAC` to get-mac-addr(`Address`) in memory.
- To read a chunk in the read-only IV region
  mac-read-check(`Address`, K_{IV}):
  1. Read `Chunk` from `Address` in memory.
  2. Return `Chunk` for a speculative execution.
  3. Read `MAC` from get-mac-addr(`Address`) in memory.
  4. Compute `MAC'` = $\mathrm{H}_{K_{IV}}$(`Address` ‖ `Chunk`).
  5. Check if `MAC` matches `MAC'`. If not, raise an exception.

Figure 4-7: The integrity verification algorithm using message authentication code (MAC).

in memory. To prevent an adversary from copying content at one chunk to another, the MAC is computed over the chunk in combination with its address. As noted before, `A` ‖ `B` represents a concatenation of `A` and `B`.

4.3.3 Hash Trees

Hash trees (or Merkle trees) are often used to verify the integrity of read-write data in untrusted storage [84]. Figure 4-8 illustrates this hash tree algorithm applied to the off-chip memory verification. The protected data chunks, from both the read-write IV region and the ME time stamps, are located at the leaves of a tree. Each internal hash is computed over a chunk below. The root hash of the tree is stored in secure on-chip memory where it cannot be tampered with while others are stored in off-chip memory.

In the figure, one chunk can contain four hashes, which results in a 4-ary hash tree (one parent hash covers four children hashes or one data chunk). For example, a 64-B chunk can have four 128-bit hashes. Because the size of the hash is fixed, a tree with higher arity requires larger chunks and larger L2 cache blocks.

To check the integrity of a chunk in the tree, the processor (i) reads the chunk, (ii) computes the hash of the chunk, (iii) checks that the resultant hash matches the parent

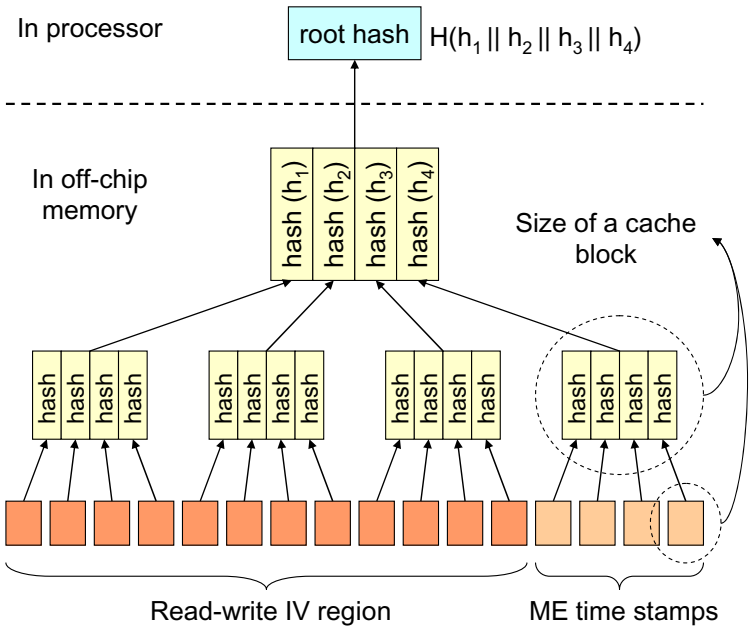


Figure 4-8: A 4-ary hash tree assuming that one chunk can contain four hashes. For example, 64-B chunks can contain four 128-bit (16-B) hashes.

hash, and (iv) checks the integrity of the chunk containing the parent hash. The steps are repeated all the way to the root of the tree. The root does not need to be checked as it is stored in secure memory.

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

With a balanced m-ary tree, the number of chunks to be checked on each memory access is $log_m(N)$, where N is the number of chunks in the verified memory space. Unfortunately, this logarithmic bandwidth overhead of the hash tree can be significant. For example, our experiments [42] showed that applying the hash tree to a processor protecting 4-GB memory can slow down the system by as much as factor of ten.

4.3.4 Cached Hash Trees: Making Hash Trees Fast

The bandwidth overhead of using a hash tree can be dramatically reduced by caching the internal hash chunks on-chip with regular data. The processor trusts data stored in the cache, and can access it directly without any checking. Therefore, instead of checking the

entire path from the chunk to the root of the tree, the processor checks the path from the chunk to the first hash it finds in the cache. We refer to this optimized hash tree scheme as `CHTree`.

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

In the following description of our algorithm, we use a term "hash cache" to refer to the cache that stores internal hash chunks. The term can represent either a shared L2 cache or a separate cache depending on the implementation.

The cached hash tree algorithm is shown in Figure 4-9. To write and read a protected chunk to/from memory, the algorithm provides two procedures chtree-write and chtree-read-check, respectively. There are three sources of memory reads and writes that require integrity verification. First, the L2 cache calls chtree-read-check on a cache miss and chtree-write on a write-back of a chunk that is in the read-write IV region but not in the ME region. Second, the ME unit calls the two procedures to access memory for both encrypted chunks and time stamps that require IV protection. Finally, the hash cache generates accesses to IV protected chunks when it reads and writes back internal hash chunks.

For chtree-write, the IV unit first computes the updated hash of the chunk to be written back. Then, the parent hash is updated with this new value. If the parent is the root hash, a register in the IV unit is updated. Otherwise, the parent is updated in the hash cache. Finally, the chunk gets written to memory. Note that the write to the parent hash in the hash cache may result in a cache miss and cause additional reads and writes to memory.

For chtree-read-check, the IV unit reads a chunk from memory and returns the chunk to a caller so that the value can be used speculatively while the verification is performed in background. To verify the integrity, the hash of the chunk is computed and compared with the parent hash read either from the root hash register or from the hash cache. As in chtree-write, this access to the hash cache can cause more memory accesses and recursive calls to chtree-read-check.

So far we have considered how the cached hash tree works when memory accesses are

- For an initialization
  chtree-init():
  1. Turn off the IV exception.
  2. For each `Address` to be protected
     (a) Read `Chunk` from `Address` in memory.
     (b) chtree-write(`Address`, `Chunk`).
  3. Flush the hash cache.
  4. Turn on the IV exception.
- To write-back an IV protected chunk to memory
  chtree-write(`Address`, `Chunk`):
  1. Compute `Hash` = H(`Chunk`).
  2. Update the parent with `Hash`.
     (a) If the parent is the root hash, update `RootHash` = `Hash`.
     (b) It not, write `Hash` to get-hash-addr(`Address`) in the hash cache.
  3. Write `Chunk` to `Address` in memory.
- To read an IV protected chunk from memory
  chtree-read-check(`Address`):
  1. Read `Chunk` from `Address` in memory.
  2. Return `Chunk` for a speculative execution.
  3. Compute `Hash'` = H(`Chunk`).
  4. Read the parent hash of the chunk.
     (a) If the parent is the root hash, `Hash` = `RootHash`.
     (b) If not, read `Hash` from get-hash-addr(`Address`) in the hash cache. (Or directly from memory, if buffers are full (See Section 10.4.3)).
  5. Check if `Hash` matches `Hash'`. If not, raise an exception.

Figure 4-9: The cached hash tree algorithm.

being verified. To initialize the hash tree on a start-up, the simple procedure chtree-init can be used. The procedure disables the integrity verification failure exception, and writes each chunk that needs to be protected. The writes result in the parent hashes being computed and stored in the hash cache. Then, the procedure flushes the hash cache so that all hashes have to be updated from leaves to the root. Finally, the IV exception is enabled to start protection.

4.3.5 Security Discussion

Let us consider the security of two proposed mechanisms. In the AEGIS security model, an adversary has complete control of off-chip memory. Therefore, during execution, attackers may *replay*, *relocate*, or *substitute* both data and meta-data such as MACs and hashes in memory.

The MAC scheme is applied only to read-only regions. As a result, there exists only one value stored by the processor for each address. Replay attacks are irrelevant. Both relocation and substitution attacks are prevented by the MAC because both an address and a value are included in the MAC computation. The security of the cached hash tree scheme is the same as the standard hash tree, which is proven to be secure against all three types of attacks [84].

Recently, researchers have found weaknesses in both MD5 and SHA-1, which are the two most widely used cryptographic hash functions. Wang found a way to find collisions in MD5 [143]. Wang et al. have also shown that collisions in the the full SHA-1 can be found in less than 2^{69} hash operations, much less than the brute-force attack of 2^{80} operations [144]. The cached hash tree algorithm is based on the collision-resistance of hash functions. Therefore, the underlying hash function must be secure in order for the hash tree algorithm to be secure. Fortunately, the hash tree algorithm can use any hash function. For example, more advanced hash algorithms such as SHA-256 [93] can be used instead of MD5 or SHA-1.

4.4 Real World Issues

4.4.1 Memory Layout

To implement the integrity verification and the encryption schemes, the layout of meta-data such as time-stamps, MACs, and hashes should be determined. The layout should be simple

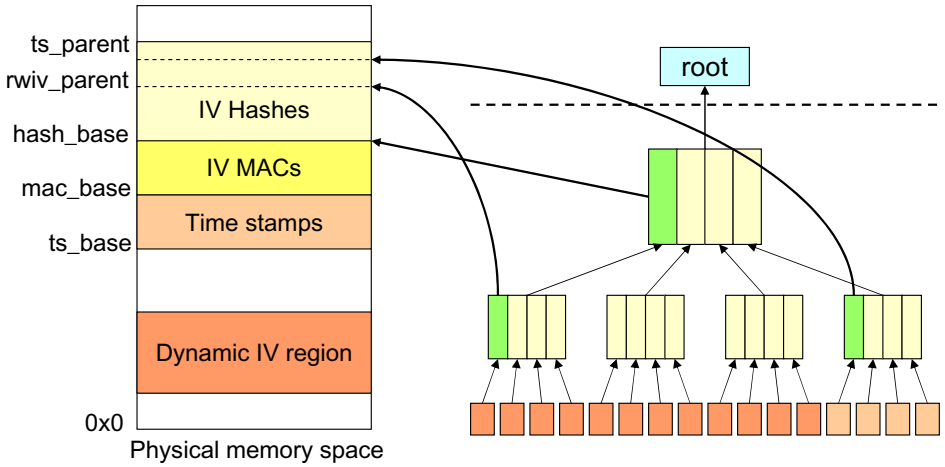


Figure 4-10: An example layout of off-chip protection meta-data.

enough for hardware to easily compute the address of the corresponding meta-data from the address of a data chunk. In the above description, we used three procedures get-ts-addr, get-mac-addr, and get-hash-addr to obtain the meta-data's address. This subsection describes how these addresses are computed.

To have the hardware units (IV, ME) compute the meta-data addresses, the security kernel must determine the layout of each meta-data in the physical memory space, and appropriately set up five special registers containing base addresses. Figure 4-10 illustrates an example layout. First, the security kernel computes the size of each meta-data regions from the protected IV and ME regions. For example, the encryption of read-write regions requires one time stamp for each L2 cache block (or chunk). Therefore, the amount of memory required to store time stamps (ts_tot) can be computed from the size of the read-write ME region ($rwme\_tot$).

$$ts\_tot = ts\_size \times \frac{rwme\_tot}{chunk\_size}. \tag{4.1}$$

ts_size is the size of an individual time stamp, and $chunk\_size$ is the size of a chunk. Thus, for 64-B chunks and 32-bit (4-B) time stamps, the amount of memory for time stamps can be obtained by dividing the size of the read-write ME region by 16.

The amount of memory required to store MACs (mac_tot) can be obtained in the same

way from the size of the read-only IV region ($roiv\_tot$).

$$mac\_tot = mac\_size \times \frac{roiv\_tot}{chunk\_size}. \tag{4.2}$$

Here, mac_size is the size of a MAC (often 128 bits). For 64-B chunks, this operation is a simple division by 4.

Obtaining the size of memory for the hash tree is a bit tricky. If we assume that the ME regions are also protected for integrity, the hash tree covers both the read-write IV region ($rwiv\_tot$) and the ME time stamps (ts_tot). Thus, the tree needs to have at least $((rwiv\_tot + ts\_tot)/chunk\_size)$ leaf nodes at the bottom. For m-ary tree with ($m = chunk\_size/hash\_size$), the tree of depth d has m^d hash nodes at the leaves. As a result the depth can be determined by

$$d = \lceil log_m(\frac{rwiv\_tot + ts\_tot}{chunk\_size})\rceil. \tag{4.3}$$

Finally, given the depth d, we can compute the amount of memory that is required for hashes ($hash\_tot$).

$$hash\_tot = hash\_size \times \frac{m^{d+1} - 1}{m - 1}. \tag{4.4}$$

Knowing the size of each meta-data region, the security kernel lays out the regions in the physical space as in Figure 4-10. The meta-data can be located anywhere in the memory as long as the region is not used by the main processor. Given the layout, the security kernel can set five address registers so that hardware can compute the meta-data addresses. ts_base, mac_base, and $hash\_base$ are simply the base addresses of the three regions. For computing the addresses of hashes, the IV unit needs two more addresses besides the base address. $rwiv\_parent$ represents the address of the parent hash of the first chunk in the read-write IV region. ts_parent points to the parent hash of the first ME time stamp chunk protected.

Given these five addresses and the base addresses of protected regions, the ME and IV units can easily compute the meta-data address from the address of a protected chunk. First, get-ts-addr computes the address of a time stamp by

$$\textsf{get-ts-addr}(addr) = ts\_base + ts\_size \times \frac{addr - rwme\_base}{chunk\_size}. \tag{4.5}$$

Here $rwme\_base$ is the base address of the read-write ME region. In a similar way, get-mac-addr computes the address of a MAC for a read-only verified chunk.

$$\text{get-mac-addr}(addr) = mac\_base + mac\_size \times \frac{addr - roiv\_base}{chunk\_size} \tag{4.6}$$

where $roiv\_base$ is the base address of the read-only IV region. Finally, get-hash-addr computes the address of the parent hash for a given chunk. This computation is a bit more complicated compared to the previous two because we have to distinguish data chunks, time stamp chunks, and hash chunks.

$$\text{get-hash-addr}(addr) = \begin{cases} rwiv\_parent + hash\_size \times \frac{addr - rwiv\_base}{chunk\_size} & \text{for data chunks} \\ ts\_parent + hash\_size \times \frac{addr - ts\_base}{chunk\_size} & \text{for time stamps} \\ hash\_base + hash\_size \times (\frac{addr - hash\_base}{(chunk\_size/hash\_size)} - 1) & \text{for hash chunks.} \end{cases} \tag{4.7}$$

In this subsection, we discussed how the security kernel can determine the amount of memory required for off-chip protection meta-data, decide the layout in the physical memory, and set the five address registers so that the protection hardware can compute the addresses. After this setup, the security kernel can begin secure processing. We note that all equations shown above can be performed with simple shift operations because all constants are powers of two.

4.4.2 Re-Sizing Protected Regions

In the description of the algorithms, we discussed how the integrity verification mechanisms are initialized. For the initialization, it does not matter what the memory contents are as long as the memory is protected *after* the initialization. The AEGIS architecture identifies the initial memory contents using the program hashes.

On the other hand, the protected regions may be re-organized during execution. For example, the security kernel can decide that it needs a larger read-write IV region. Therefore, the processor should provide a way to *re-size* and *re-locate* the protected regions as well as the meta-data regions during secure processing. For encryption, this is relatively simple because the processor only needs to copy existing time stamps to new locations. The challenge is to re-size the IV regions with a guarantee that the memory contents cannot be changed during the re-sizing.

Here, we propose a simple solution based on a collision-resistant hash function.

1. With the protection enabled, the processor reads and computes the hash of the parts of the IV regions (both read-only and read-write) that will be still protected after the re-sizing. This hash is stored in a secure on-chip register.

2. The protection mechanisms are re-initialized with mac-init (with a new key) and chtree-init.

3. The processor again reads and computes the hash of the parts of the new IV regions that were previously protected, and checks if this new hash matches the saved hash.

This check ensures that no tampering can happen during the re-initialization. Also, this procedure does not need to keep the old meta-data during the re-initialization.

4.4.3 Direct Memory Access

The integrity verification and encryption schemes allow only the primary processor to access off-chip memory. For untrusted I/O such as Direct Memory Access (DMA), a part of memory is set aside as an unprotected and unencrypted area. When the transfer is done into this area, the security kernel copies the data into protected space. Once protected, the I/O input can be checked for its integrity and decrypted by the application itself using a scheme of its choosing.

Chapter 5

Processor Architecture

This chapter describes the AEGIS processor architecture, which provides secure execution environments and secure communication capabilities under software and physical attacks. The main goals of this processor architecture are to protect the security kernel's integrity and privacy during execution, provide secure communication capabilities (attestation and decryption) to the security kernel, and provide mechanisms to the security kernel so that it can protect user applications.

This chapter is organized as follows. We first describe the architectural features to provide secure execution environments, which include secure execution modes, memory protection, and secure mode transition. Then, the processor's private-key operations for secure communication and other miscellaneous features such as random number generation and debugging support are discussed. Finally, the architecture is briefly summarized, and its security is discussed.

5.1 Secure Execution Modes

To allow for varying levels of security, AEGIS not only has user and supervisor modes, but also has four secure program execution modes: standard (STD) mode, tamper-evident (TE) mode, private tamper-resistant (PTR) mode, and suspended secure processing (SSP) mode (see Chapter 2).

To keep track of which security mode a supervisory process is currently in, the processor maintains the supervisor mode (SM) bits, which can be updated only through special mode transition instructions. A user process also operates in its own security mode independent of

the supervisory process. The user mode (UM) bits keep track of the user process' security mode, and are managed by the security kernel.

Mode	VM Control	Memory Verified/Private	Mode Transition	Security Insts
STD	Y	-/-	TE/PTR (`l.aegis.enter`)	N
TE	Y	RW/-	PTR (`l.aegis.csm`) SSP (`l.aegis.suspend`) STD (`l.aegis.exit`)	Y
PTR	Y	RW/RW	TE (`l.aegis.csm`) SSP (`l.aegis.suspend`) STD (`l.aegis.exit`)	Y
SSP	N	R/-	TE/PTR (`l.aegis.resume`)	N

Table 5.1: The permissions in each supervisor execution mode.

Table 5.1 summarizes the four supervisory security modes and their capabilities in terms of control of the virtual memory (VM) mechanism, accesses to the Verified and Private memory regions, transitions to other security modes, and the permission to use security-critical instructions.

STD mode has the privilege to control the VM mechanism so that conventional operating systems can run on the processor. However, the operating system in STD mode cannot use any security instructions except for entering a secure mode (TE/PTR). Trusted security kernels run in TE and PTR modes, which provide protection for secure execution as well as full capability to control VM, change the execution mode, and use security instructions. Finally, SSP mode does not have any permission that may affect the TE or PTR modes, and can only switch back to the secure mode that was suspended.

The only major difference between TE and PTR modes is that the private regions of memory can only be accessed under PTR mode. While PTR mode can access both private and public memory regions, the main reason to have a separate TE mode is to avoid unnecessary performance degradation. PTR mode must ensure the authenticity of stores which can potentially write private data into public regions of memory (see Section 5.2). On the other hand, TE mode does not have to wait on stores to public regions because private information cannot be accessed in TE mode.

The user mode permissions are similar to the corresponding supervisor modes. However, the memory access permissions apply to the user application's virtual memory space. User

applications cannot access the supervisor's memory space. Also, the mode transition and other security instructions are supported by the security kernel as system calls. As a result, the user mode applications do not have permission to use the processor's security instructions directly. User mode has no control of the VM mechanism.

5.2 Memory Protection

To ensure secure program execution, the processor needs to guarantee the integrity and privacy of instructions and data in memory under both software attacks and physical attacks. The AEGIS architecture defends against software attacks using the virtual memory (VM) mechanism with additional access permission checks within the memory management unit (MMU), and protects against physical attacks with off-chip protection mechanisms, namely integrity verification (IV) and memory encryption (ME).

5.2.1 Overview

At startup, no protection mechanisms are enabled (including virtual memory). When the security kernel enters a secure execution mode (TE/PTR), the protected regions of physical memory are specified as parameters to the `l.aegis.enter` instruction and the processor initiates off-chip memory protection mechanisms.

Once the security kernel is in a secure mode and protected from physical attacks, it can configure and enable the virtual memory and access permission checks in the MMU to defend against software attacks. The security kernel first sets up its own virtual memory (VM) space, and defines protected regions within that space to ensure that other parts of the operating system in SSP mode cannot tamper with the security kernel. Each user application specifies protected regions within its own virtual memory space when it enters a secure execution mode. Then, the security kernel can accordingly set the access permission for those regions.

5.2.2 Protection Against Physical Attacks

As shown in Figure 5-1, the processor separates physical memory space into regions designated "IV protected" or "ME protected", to provide protection against physical attacks on off-chip memory. In practice, the ME regions will be inside the IV regions. The integrity

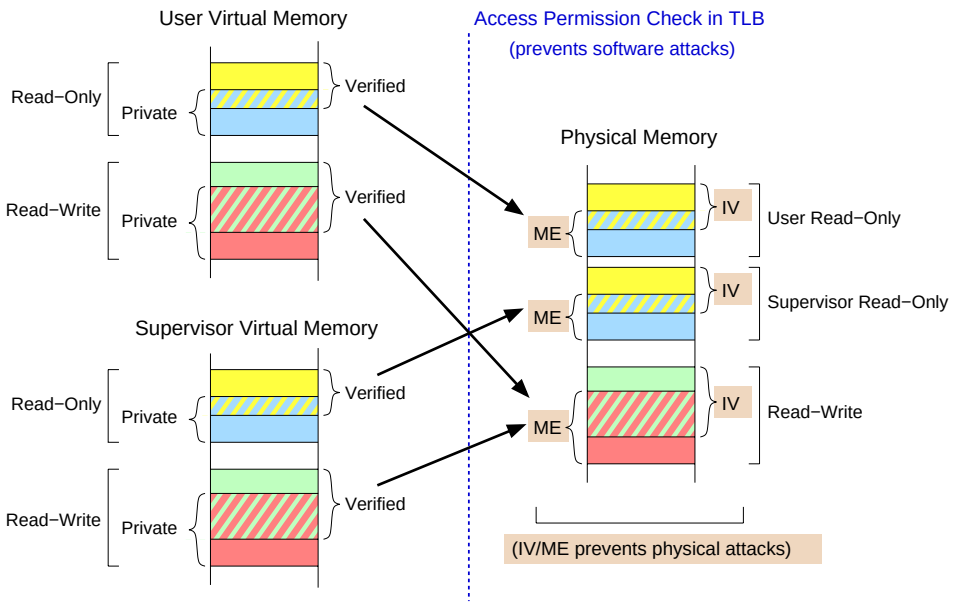


Figure 5-1: Protected regions in virtual and physical memory

verification mechanism detects any tampering that changes the content of the IV regions, while the encryption mechanism guarantees the privacy of the ME regions. The IV and ME regions can also be either "read-only" (RO) or "read-write" (RW).

Memory encryption is handled by encrypting and decrypting all off-chip data transfers in the ME regions using a counter-mode encryption scheme (see Section 4.2). The location of the time stamps is specified by the security kernel before starting secure processing. The secret key for the read-write ME region is randomly chosen on each execution by the processor when the `l.aegis.enter` instruction executes, and the secret key for the read-only ME region is set by the security kernel or user applications.

The processor protects the read-write IV region using the cached hash tree and protects the read-only IV region using the MAC scheme (see Section 4.3). The processor chooses a random secret key for this read-only IV region upon start-up with the `l.aegis.enter` instruction, or when the IV region is changed. Again, the security kernel should reserve the memory for hashes and MACs, and properly set the special registers of the integrity verification unit.

The processor only needs to have a single read-write IV/ME region (one each), because the read-write regions can be shared between user applications and the security kernel; this can be handled easily by a security kernel's VM manager. On the other hand, the

processor separately provides user-level read-only IV/ME regions and supervisor-level read-only IV/ME regions. The same IV/ME region cannot be shared between a user application and the security kernel because they are likely to require different secret keys.

As a result, the processor supports six IV/ME regions in the physical memory space. All six regions are delineated directly by twelve special purpose registers marking the beginning and end of each domain at a page granularity. Modifying these special purpose registers is done through a supervisor-level instruction, `l.aegis.cpmr`, which can be used only in the TE/PTR mode. Modification of the boundary of an existing IV region can be quite expensive, as it requires re-initialization of the integrity verification mechanism (see Section 4.4.2).

5.2.3 Protection Against Software Attacks

The conventional virtual memory (VM) mechanism isolates the memory space of each user process and prevents software attacks from malicious programs. To defend against software attacks from an unprotected portion of a process in SSP mode, the processor performs additional access permission checks in the MMU as explained below.

Both the security kernel and user applications define four protected regions in virtual memory space, which provide different levels of security. Because these regions are protected by the VM mechanism, the regions are specified at a page granularity.

1. Read-only Verified memory
2. Read-write Verified memory
3. Read-only Private memory
4. Read-write Private memory

The security kernel simply sets up its protection regions along with the VM mapping. The user application specifies these regions during a system call to enter a secure execution mode.

The processor grants access permission to each region based on the current secure execution mode. Specifically, Verified memory regions allow read-write access while in TE or PTR mode, but only allow read access in STD or SSP mode. The Private regions are accessible only in PTR mode.

To properly protect the user's Verified and Private regions from physical attacks, the virtual memory manager (VMM) in the security kernel needs to map the Verified and Private regions in the virtual space to the IV and ME regions in the physical memory space. Figure 5-1 illustrates how this is done, with Verified regions mapping to IV regions and Private regions mapping to ME regions.

5.2.4 Speculative Execution

One last noteworthy point about memory protection involves the use of integrity verification. Since the latency of verifying values from off-chip memory can be quite large, the processor "speculatively" executes instructions and data which have not yet been verified. The integrity verification is performed in the background. However, whenever a security instruction is to be executed (either `l.aegis.*` or `l.puf.*`) or an exception takes place, the processor verifies all previous instructions and data before executing the security instruction or taking an exception.

In PTR mode, the processor must also wait for all previous off-chip accesses to be verified before initiating stores which write data to non-private memory. This is to confirm that a store was indeed intended since otherwise private data could leak into non-private memory.

There is one item of note regarding the use of the OTP (counter-mode) encryption with speculative execution. A previous study pointed out that, with the OTP encryption, speculatively using instructions and data before the integrity verification is complete can cause security holes because of information leakage through memory access patterns [127]. However, as discussed in Section 2, we assume that the address bus is protected with appropriate schemes such as oblivious RAMs [43] or HIDE [159]. It is important to note that without these protections for side channels, there are other security breaks that can compromise the privacy of applications. The leakage through memory access patterns is orthogonal to the OTP encryption scheme.

5.3 Execution Mode Transition

The AEGIS architecture controls the transition of the supervisor's security mode while relying on the security kernel to control multitasking user applications. Here, we describe the processor support for the security kernel's mode transition. Chapter 6 discusses how

the security kernel can provide the same functions to user applications.

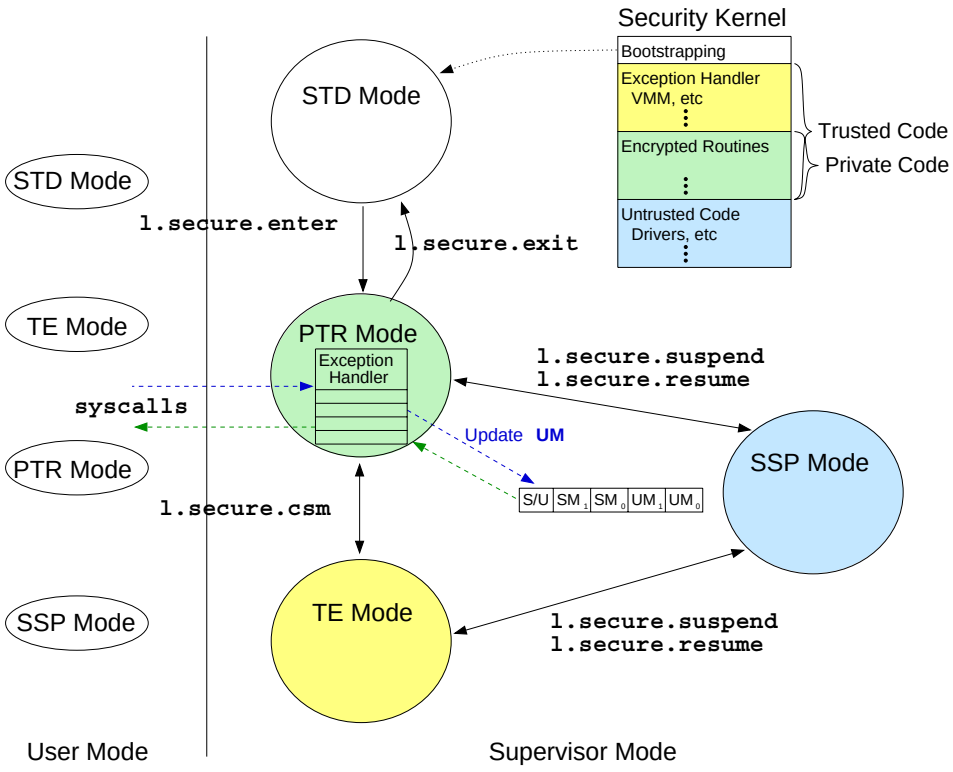


Figure 5-2: Security modes and transitions

Figure 5-2 shows one possible usage of a security kernel running on the AEGIS processor to illustrate the architectural support for secure mode transitions in supervisor mode.

5.3.1 Start-Up

The processor begins operation within the supervisor STD mode. Regular operating systems that do not use the AEGIS security features can keep operating in STD mode. It is likely, though, that a security kernel would transition to TE or PTR mode very early, as critical kernel subsystems such as the Virtual Memory Manager (VMM) must be executed in a protected mode to be secure. To enter TE or PTR mode and begin executing in a secure environment, the security kernel executes `l.aegis.enter`. This instruction, as with all of the security instructions introduced here (`l.aegis.*`), requires supervisor permissions to run.

To ensure a secure start-up, the `l.aegis.enter` instruction must be called with parameters defining which mode (TE or PTR) to enter, boundaries for integrity verification (IV),

memory encryption (ME), a protection configuration, and program hash regions of memory. The processor then performs the following actions before returning control:

1. Turn off the virtual memory (VM).
2. Initialize and enable the integrity verification mechanisms.
3. Choose a random encryption/decryption key for the read-write ME region, and enable the ME mechanism.
4. Compute and save the security kernel's program hash.
5. Set the execution mode to be either TE or PTR.

The VM is disabled to ensure that secure execution begins in the physical memory space where the security kernel defines the protected regions. Then, the IV and ME mechanisms are enabled to ensure that no physical attack can tamper with the execution. Finally, the program hash is computed to record the identity of the security kernel.

The hash of the security kernel ($SKHash$) is computed over the binary defined by PC-relative offset parameters as well as the IV/ME boundaries and the protection setting. In this way, $SKHash$ not only identifies the security kernel itself, but also depends on its view of protection mechanisms and protected memory. Formally, $SKHash$ is defined as

$$SKHash \quad = \quad H([PC - d1] \rightarrow [PC + d2], B, P)$$

where $d1$ and $d2$ are the reverse and forward offset distances specifying the region of memory containing the kernel code, respectively, B contains all IV/ME boundary addresses, and P represents the protection setting. In the protection setting, the AEGIS architecture lets the security kernel select between TE and PTR, and choose to enable debugging features, or disable the off-chip memory protection when physical attacks are not of concern. Using a PC-relative offset for $SKHash$ allows for position independent code, and should delineate both the instruction code as well as any initialization data.

5.3.2 Changes between TE and PTR

Once running in TE/PTR mode, the "Change Secure Mode" instruction (`l.aegis.csm`) can be executed to transition back and forth between TE and PTR modes. The `l.aegis.csm`

instruction is rather straightforward; the processor simply changes the security mode by setting the SM bits. The `l.aegis.csm` instruction can only be used in either TE or PTR mode.

5.3.3 Suspend and Resume

As seen in Figure 5-2, the `l.aegis.suspend` instruction can be run from either TE or PTR mode to change the security mode to SSP. Conversely, the `l.aegis.resume` instruction can be issued from SSP mode to return to the suspended secure mode (TE or PTR).

To ensure that program's protected regions cannot be tampered with by code executing in SSP mode, extra precautions must be taken. First, programs in SSP mode have very limited capability, which cannot compromise the protected memory regions. They cannot write into the Verified regions or access the Private regions. The programs also cannot change the VM or use the other security instructions in SSP mode (see Section 5.1). Second, programs can only return to the suspended secure execution mode at the exact location specified by the `l.aegis.suspend` instruction.

The `l.aegis.suspend` instruction requires the address, at which the secure execution will resume, as a parameter. The processor stores the current secure mode (SM bits) and the resume address in secure on-chip memory before entering SSP mode. Then, when the program wishes to return to TE or PTR mode, it calls the `l.aegis.resume` instruction in SSP mode. The processor will confirm that the PC value of the resume instruction is the same as the address given by `l.aegis.suspend`, and will change the security mode back to the mode which initiated the transfer to SSP mode.

5.3.4 Exit

The `l.aegis.exit` instruction can be issued from TE or PTR mode, and will exit entirely to the unprotected STD mode, removing all memory of prior security state such as general-purpose registers, encryption keys and private data in the cache.

5.3.5 Exceptions

As seen in Figure 5-2, there is one additional way for the processor to enter a supervisor's secure execution mode; exceptions including traps (system calls), faults, and interrupts need to be serviced by the protected part of the security kernel. Thus, if the security

kernel is in TE/PTR/SSP mode (that is, `l.aegis.enter` has already been called without a `l.aegis.exit`), exceptions trigger the processor to enter the supervisory PTR mode at the hardwired handler location. The security kernel should ensure that proper handler code is at the location.

The IV failure exception is handled slightly differently. If the integrity verification fails, the processor aborts the secure processing by effectively performing the `l.aegis.exit` instruction and executes the exception handler in supervisory STD mode. Because the IV failure only occurs when the system is being physically tampered with, we do not consider recovery to be necessary. Also, with an IV failure, processor can no longer guarantee the integrity of the handler code.

5.4 Private Key Instructions

The protections described above can guarantee the secure execution of a program. For this secure execution to be useful in practice, however, the processor must be able to communicate with remote parties in an authentic and private way even when communication channels are untrusted. First, for authenticity, the interacting parties should be able to authenticate the processor, the security kernel, and the user application. Second, for privacy, the remote parties should be able to send secrets that can be decrypted only by a specific program on a specific processor.

For this purpose, AEGIS provides two private-key instructions, `l.aegis.pksign` and `l.aegis.pkdecrypt`, to the security kernel. The security kernel provides similar system calls for user applications.

5.4.1 Signing

The `l.aegis.pksign` instruction gets a message M as an input, and returns the signature $\{SKHash \| M\}_{SK}$. $SKHash$ is the program hash of the security kernel that was computed when the `l.aegis.enter` instruction was executed, and SK is the processor's private key. Here, $A \| B$ represents a concatenation of A and B.

This instruction can be executed only in supervisory TE or PTR mode. Also, the signature always includes the security kernel's program hash. That way, when remote parties receive a message signed by the processor's private key SK, they know that the

message is from a particular security kernel running on a particular processor.

5.4.2 Decryption

The second instruction, `l.aegis.pkdecrypt`, provides a private-key decryption operation. The instruction gets an input $E_{PK}\{SKHash \| M\}$, which contains the security kernel's program hash $SKHash$ and a message M, encrypted with the processor's public key PK. The processor decrypts the input, and returns the decrypted message M only if $SKHash$ matches the hash of the executing security kernel. Otherwise, the instruction results in a security exception. The encryption scheme should be non-malleable so that an adversary cannot change the encrypted program hash in a way that a different security kernel can decrypt the message.

Using this instruction, a remote system can send a private message only to a particular security kernel on a particular processor. The security kernel can provide the same service to user applications. The main use of this instruction is to provide a key that can be used to decrypt encrypted code in the read-only ME region. For example, in software copy protection, an application can be encrypted with a vendor's chosen key K, which is then provided to a particular processor after being encrypted by the processor's private key. The security kernel can also use this instruction to store private data in non-volatile storage by encrypting it in a way that only the specific kernel itself can obtain the data again later.

5.5 Miscellaneous Instructions

The secure processor must provides a few more security features. This section describes an instruction to generate physically secure random numbers, and an instruction to access special security registers.

5.5.1 Random Number Generation

Since many cryptographic security applications require a source of pure randomness, a secure processor should implement a physically secure random number generation mechanism on-chip. For example, to ensure the freshness of off-chip communication, protocols often require sending a random nonce. Without a secure random number, an adversary can carry out replay attacks.

Register type	Permitted mode
IV, ME meta-data locations	Supervisory STD, TE, and PTR
ME read-only decryption keys	Supervisory PTR
MMU permission check registers	Supervisory TE, PTR
User security mode register (UM)	Supervisory TE, PTR

Table 5.2: The security registers that can be directly set by the security kernel using the `l.aegis.setreg` instruction.

For this purpose, the AEGIS processor provides the `l.aegis.random` instruction. The instruction returns a 32-bit random number generated by a physically secure hardware random number generator. Various hardware random number generators have been previously proposed [28, 57, 102]. It is also possible to use the existing PUF circuitry to generate a random number which is acceptable for cryptographic applications [94].

5.5.2 Special Register Accesses

In AEGIS, most security registers are only accessible by the processor, and updated as a by-product of the security instructions. For example, the program hash register containing $SKHash$ will be written by the processor when the `l.aegis.enter` instruction gets executed. However, there are some registers, which the processor allows the security kernel to directly manage using the `l.aegis.setreg` instruction.

Table 5.2 summarizes the registers that can be set by the security kernel. First, the supervisor process can set the registers specifying locations of the IV and ME meta-data such as MACs, hashes, and time stamps. The security kernel must set these register before it enters a secure mode. Tampering with these registers will result in an integrity verification failure. Second, once the security kernel enters PTR mode, it can set the keys for the read-only ME regions. In common cases, the security kernel will obtain these keys using the private-key decryption instruction `l.aegis.pkdecrypt`. Finally, as discussed before, the security kernel manages the access permission checks in MMU and the user mode transitions. Therefore, the registers for those mechanisms can be set by the kernel in TE or PTR mode.

5.6 Debugging Support

Modern microprocessors often have back doors such as scan chains for debugging. Application programs also commonly use software debugging environments such as `gdb`. While the debugging support is essential to test the processor and develop applications, it is clearly a potential security hole if the adversary can use debugging features to monitor or modify on-chip state.

In our design, the processor selectively provides debugging features to ensure the security of protected kernel and applications. Debugging features are enabled when the kernel is in STD mode so that the processor can be tested. In other modes, the debugging is disabled unless the security kernel specifies otherwise with `l.aegis.enter`. The processor includes whether debug is enabled or not when it computes $SKHash$. Thus, the security kernel will have different program hashes depending on whether the debugging is enabled or not. In this way, the security kernel can be debugged when it is developed, but the debugging will be disabled when it needs to be executing securely. This idea is similar to Microsoft NGSCB [86].

While most internal registers can be allowed to be accessed through a scan chain for debugging purposes if the security kernel is not in a secure mode, we note that some registers should *never* be accessible by the debugging interface. First, as discussed in Section 3.8, the input registers for the PUF circuits should not be a part of the scan-in chain, and the circuit output register should not be a part of the scan-out chain. Second, the debug interface should not be able to modify the SM bit registers. Otherwise, an adversary can bypass the `l.aegis.enter` instruction to enter a secure mode without properly setting the protection mechanisms.

5.7 Security Instruction Summary

Table 5.3 summarizes the new security instructions added in the AEGIS processor architecture. First, the processor provides PUF instructions so that a private key can be generated and securely expressed using a PUF. Then, the processor manages secure execution of the security kernel using a set of mode transition instructions and the `l.aegis.cpmr` instruction to manage the protection mechanisms for physical memory. For secure communication with remote parties, two private key operations and hardware random number generation are

Instruction	Description	Permitted modes
`l.puf.pksave`	Generate and encrypt a private key	TE, PTR
`l.puf.pkload`	Reload the private key	TE, PTR
`l.aegis.enter`	Enter a secure mode (TE/PTR)	STD
`l.aegis.cam`	Change between TE and PTR	TE, PTR
`l.aegis.suspend`	Suspend TE or PTR	TE, PTR
`l.aegis.resume`	Return from SSP to TE, PTR	SSP
`l.aegis.exit`	Exit secure processing	TE, PTR
`l.aegis.cpmr`	Change the IV/ME protection regions	TE, PTR
`l.aegis.pksign`	Sign a message with a private key	TE, PTR
`l.aegis.pkdecrypt`	Decrypt a message with a private key	TE, PTR
`l.aegis.random`	Generate a random number	Any
`l.aegis.setreg`	Set a security register value	Depend on registers

Table 5.3: The summary of the AEGIS security instructions. In permission, the security modes are for the security kernel except for `l.aegis.random`, which can be used in any user or supervisor mode.

provided. Finally, the processor allows the security kernel to control some security registers with the `l.aegis.setreg` instruction.

5.8 Security Discussion

This section discusses the security of the AEGIS processor design. The discussion in this section focuses on the protection of the integrity and privacy of the program execution. As discussed in the security model, AEGIS does not deal with denial of service (DoS) attacks or attacks exploiting software bugs.

5.8.1 Program Integrity

The term *integrity* refers to the correct execution of binary code as it is written. Binary code is executed correctly if the following three requirements are satisfied. First, the code should start at a correct entry point with correct initial state. Second, during the execution, the state can only be changed by the owner process itself unless the code explicitly asks for I/O or inter-process communication (IPC). Finally, the processor should carry out each instruction correctly as defined in the instruction set architecture. Because we assume that the processor is correctly implemented and protected, attackers cannot change processor behavior. Thus, the only possible attacks are on the program state.

In modern computer systems, program state consists of three parts: on-chip registers, instructions and data in virtual memory space, and state in non-volatile storage. Each instantiation of a program, called a *process*, has its own registers and virtual memory space. Registers include general-purpose registers, a program counter, and a stack pointer. Virtual memory space contains instructions and data. A program may also store a part of its state in non-volatile storage such as hard-disk drives, flash memory, etc.

We can categorize possible attacks into two types based on when they change the program state. First, before a program starts, attackers may initialize the state with malicious values to have a program start with unexpected state (*initialization attacks*). Second, during execution, attackers may *replay*, *relocate*, or *substitute* the state. In *replay attacks*, attackers replace a new value with an old value of the same location. The processor needs to be able to check the freshness of each state in order to detect a replay attack. In *relocation attacks*, attackers move a valid value from one location to another. The processor should be able to verify the location (address) where each state is saved. Finally, *substitution attacks* replace a state with an arbitrary value. The processor should verify that the state is generated by the owner process.

Attacks

With physical access to a system, attackers can have complete control over off-chip memory and local non-volatile storage such as hard-disks. Therefore, all types of attacks are possible for this off-chip state. On the other hand, on-chip state is safe from physical attacks since we assume that the on-chip registers and caches are protected while the processor is powered on. Only software attacks from different processes or from an insecure part of the same process are possible for on-chip caches.

- *Initial executables*: Instruction and data in the virtual memory space may be initialized arbitrarily since they are stored in off-chip memory. For example, an attack may replace an original executable with a malicious one in memory, and make the processor execute the malicious code.

- *On-chip/off-chip memory*: Physical attacks can arbitrarily change the contents of off-chip memory. Also, software attacks can change the contents of on-chip caches. Therefore, attackers can replay, relocate, and substitute instructions and data in mem-

ory. In general, the entire virtual memory space is vulnerable.

- *Local non-volatile storage*: Local non-volatile storage is also open to physical and software attacks. Therefore, replay, relocation, and substitution attacks are possible.

Protection

Now we briefly summarize how the AEGIS architecture protects program integrity against various attacks discussed in the previous subsection. The current design prevents all the attacks except replays on non-volatile storage.

- *Initialization attacks*: The main mechanism to ensure proper initialization is to compute the hash of a program when it enters a secure execution mode. For the security kernel, the processor computes the program hash on the `l.aegis.enter` instruction. For user applications, the security kernel computes the hash. Therefore, any changes in initial program state will be detected by the program hash.

- *Software attacks on memory*: Software attacks are prevented using permission checks in the MMU. Conventional virtual memory isolates each process' virtual memory space, and additional permission checks in the MMU prevent malicious software in the STD or SSP mode from tampering with protected memory regions.

- *Physical attacks on off-chip memory*: AEGIS uses the integrity verification mechanisms in Section 4.3 to prevent physical attacks on off-chip memory. These mechanisms detect all three types of tampering.

- *Attacks on non-volatile storage*: Applications can use conventional cryptographic tools such as a message authentication code (MAC) to verify data stored in non-volatile storage. MACs can easily prevent substitution and relocation attacks. During execution, applications can also prevent replay attacks using its memory, which is protected from replay attacks. Therefore, pages swapped to hard-disk can be protected by the security kernel. On the other hand, replays on untrusted non-volatile storage cannot be detected in the current design once the processor is powered off and rebooted.

5.8.2 Program Privacy

We want a program to be able to keep any of its state including registers, instructions and data in virtual memory space, and non-volatile state private. There are two ways for secret information to leak: direct channels and indirect channels.

- *Direct channels*: If a secret is stored off-chip, physical attackers can easily read the secret directly. Also, software attacks can read on-chip registers and caches. AEGIS uses encryption to prevent secrets in off-chip memory from being directly revealed. Whenever a private cache block is evicted, it is encrypted. In on-chip caches, the MMU isolates each virtual memory space and prevents STD/SSP mode from reading Private regions. The security kernel manages an interrupt and protects on-chip registers. Finally, data in non-volatile storage can be encrypted by the security kernel or a user application.

- *Indirect channels*: In the AEGIS security model, the processor is assumed to be protected against side-channel attacks.

5.8.3 Protection Summary

Problems	Protection
Security kernel start-up	Processor computes $SKHash$
Inter-process software attacks	Virtual Memory (VM)
Software attacks in SSP mode	Permission checks in the MMU
Physical attacks on off-chip memory	Integrity verification, memory encryption
Application start-up	The security kernel computes the $AHash$
Multi-tasking	Managed by the security kernel
Pages on disks	Verified and encrypted by the security kernel

Table 5.4: The protection mechanisms in the AEGIS architecture.

Table 5.4 summarizes the protection mechanisms used in the AEGIS processor architecture to ensure secure execution. Any attacks before the security kernel starts an execution, such as executing an untrusted security kernel, are detected by different program hashes ($SKHash$). During execution, there can be physical attacks on off-chip memory and software attacks on both on-chip and off-chip memory. The physical attacks are defeated by hardware IV and ME mechanisms, and the VM and the additional access checks in the MMU prevent illegal software accesses.

Once the processor guarantees the secure execution of the security kernel, the kernel protects multi-tasking user applications by computing the application's hash at a start-up, and managing context switches. Also, pages swapped out to secondary storage are protected in software by the security kernel.

Chapter 6

Security Kernel

The baseline AEGIS architecture relies on the core part of an operating system, called the security kernel, to manage and orchestrate the security mechanisms for multi-tasking user applications. This chapter outlines the functions that the security kernel must provide to complement the processor architecture. The goal of this chapter, however, is to provide the guidelines for designing the security kernel, not to describe a detailed implementation, which is out of the scope of this thesis. Therefore, we focus on the overview of each security function inside the security kernel and how each function is related to AEGIS' architectural features.

This chapter is organized as follows. First, we discuss the minimum set of functions that the security kernel must include and outline the overall structure. Then, the security kernel's start-up process is explained to illustrate how the proposed architecture can protect the kernel. Finally, given that we understand how the security kernel itself becomes secure, we discuss the kernel's functions to protect user applications. These security functions are grouped into two categories; the first group of functions secure the application's execution and the others provide system calls as an interface between the security kernel and user applications.

6.1 Operating System Partition

As described in Chapter 5, the processor identifies the security kernel by computing its hash when the kernel enters a secure mode (TE/PTR), and provides the private key operations so that the security kernel's hash can be authenticated. Knowing the security kernel's hash,

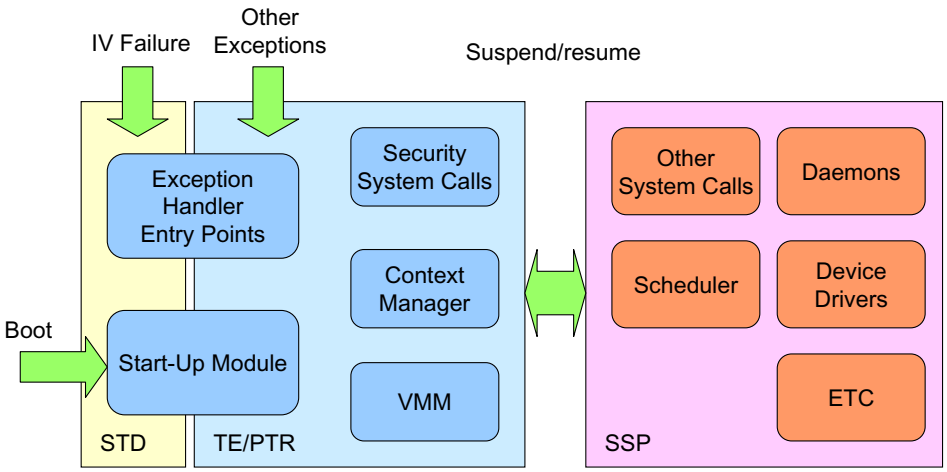


Figure 6-1: An example partition of operating system functions between the security kernel and the untrusted part of the OS.

remote parties must be able to trust the security kernel in order to trust the AEGIS system's behavior.

Unfortunately, verifying complex code to ensure that it does not contain bugs or malicious code is difficult. For example, if the security kernel is designed to be a conventional operating system with additional security functions, trusting such complicated code would be practically impossible. Therefore, the main goal of the security kernel design should be minimizing the components inside the kernel; the kernel itself needs to be verified.

As shown in Figure 6-1, the operating system functions are partitioned between the security kernel and the rest of the operating system. The security kernel starts in STD mode, but enters a secure mode (TE/PTR) immediately after setting up protection mechanisms for itself. To execute the rest of the operating system outside of the security kernel, the kernel suspends secure processing into SSP mode. Because only the modules inside the security kernel execute in a trusted mode, we only need to verify and trust those components in the security kernel.

The security kernel modules in the figure represent the minimum set of functions that need to be included. Essentially, the modules that manage protection related to security should be a part of the security kernel. On the other hand, other OS modules that provide services or manage resources do not need to be trusted because we do not consider denial-of-service (DoS) attacks. Physical DoS attacks cannot be prevented (see Section 2.4). Therefore, we need to include only the start-up module, the virtual memory manager (VMM),

the entry points for exceptions (interrupts, faults, and traps), the context manager, and the modules to service special security system calls in the security kernel.

Only the security-related parts of these components are necessary in the security kernel. For example, a traditional VM manager decides the virtual-to-physical mapping as well as paging to determine which pages should be in memory and which should be in disk. For the security kernel, the part that controls virtual-to-physical mapping should be included, but not necessarily the paging that manages memory resources. Similarly, the context manager that saves and restores application's state must be trusted, but not the scheduler that manages computation resources. In the following sections, we will discuss the functions that each security kernel module should perform in more detail.

6.2 Kernel Start-Up

Before discussing how the security kernel handles user applications, let us first understand how the security kernel can use the architectural mechanisms to protect itself from both software and physical attacks. This section explains the sequence of operations performed by the start-up module in the security kernel, which is in charge of setting up all protection mechanisms and entering secure processing.

The secure start-up consists of three phases. First, the start-up module must prepare to enter a secure mode by setting up proper protection regions and meta-data locations. Then, the security kernel enters a secure mode protecting itself from physical attacks. Finally, the VM mechanism protects against software attacks. The following summarizes these steps in more detail.

1. Determine IV/ME regions in physical memory. The protected regions should include all instructions and data that belong to the security kernel.

2. Determine the meta-data layout for the ME time stamps, the IV MACs, and the IV hashes. Also, compute the corresponding meta-data addresses, and set five meta-data address registers (Section 4.4.1). This step prepares the off-chip memory protection mechanisms to be enabled.

3. Set up the input parameters for the `l.aegis.enter` instruction (Section 5.3.1). The parameters consist of the offsets defining the program hash region, the addresses

defining the IV/ME regions, and the protection setting. Set the protection to enter PTR mode with debugging disabled and off-chip protection enabled.

4. Call `l.aegis.enter`. This instruction enables the off-chip protection mechanisms, computes the program hash, and enters PTR mode. Thus, after this step, the security kernel is protected from physical attacks.

5. Check the program counter to ensure that the entry point is correct, and check the stack pointer to ensure that the kernel has a large enough stack.

6. Set up the supervisor's virtual memory (VM) space, and Verified and Private regions within the virtual space (Section 5.2). This protects the security kernel from both user applications and the rest of the operating system in SSP mode.

Once in a secure mode (TE/PTR) after the initial start-up, the security kernel is protected from both physical and software attacks. During execution, however, the security kernel must suspend itself to execute either user applications or the rest of the operating system. Therefore, the kernel must be carefully designed to ensure the security of "re-entry". First, the entry points for exceptions including interrupts, faults, and traps must be located within the security kernel because the processor enters PTR mode on these incidents. Second, to ensure a secure resume from SSP mode, the kernel must properly set the resume address and save/restore its registers while in TE/PTR mode. Chapter 7 discusses the issues regarding the mode transition in detail when we describe the programming model.

6.3 Protection Management

So far we have discussed how the security kernel enters and re-enters a secure mode. This section describes functions of each module in the security kernel that protects both the kernel and user applications during execution.

6.3.1 Virtual Memory Manager

The first responsibility of the virtual memory manager (VMM) is to protect the security kernel's and user applications' memory space from software attacks. As in conventional operating systems, the VMM controls the virtual-to-physical mapping and isolates each process's memory space from others. In addition to process isolation, the VMM in the

secure processor must also manage the protection for Verified and Private regions within a process. The VMM sets registers defining the protected regions for each virtual memory space so that code in SSP mode cannot tamper with the protected regions. The regions are specified by each process as parameters of a system call to enter a secure mode.

Under physical attacks, secondary storage in our system cannot be trusted. As a result, if memory pages are swapped out to secondary storage such as disks, their integrity and privacy must be protected. In the security kernel, the VMM also takes this responsibility of protecting swapped out pages. For this purpose, the VMM can implement the encryption and integrity verification algorithms described in Chapter 4 for disks at a page granularity.

By having the VMM manage virtual-to-physical mapping, memory access permissions, and the protection of swapped pages, the security kernel can ensure that each process' virtual memory space is secure in both primary memory and secondary storage.

6.3.2 Context Manager

For multi-tasking, the operating system must maintain each process' state saving and restoring it on a context switch. Obviously, the integrity and the privacy of the process state must be ensured for security. Therefore, this function is handled by the trusted module called the context manager in the security kernel.

The basic operations of the context manager are the same as the ones in conventional operating systems; it maintains the data structure that stores each process' execution state, and saves/restores the state on a context switch. In our secure processor, however, there is more information to maintain for each process. In addition to the register values, the program counter, and the page table for virtual memory, the context manager should also manage the program hash, the secure mode bits (UM), Verified/Private regions within the virtual memory space, and the decryption/MAC keys for read-only ME/IV regions. Also, it is not enough for the context manager to save the register values on an interrupt. For privacy, the context manager should clear all registers before it transfers control to other parts of the operating system.

6.3.3 Exception Handlers

The processor enters the supervisory PTR mode and transfers control to exception handlers at fixed addresses on exceptions such as interrupts, faults, and traps. Therefore, to ensure

the security on such exceptions, the entry points of the exception handlers must be inside the security kernel. Once the context manager securely saves and clears the interrupted process' state, the main part of the exception handling such as reporting errors and servicing I/O can be done in the untrusted part of the OS.

The exception handlers operate in the same way as they do in conventional operating systems. The only difference is that the exception handlers in the security kernel must handle a few additional sources of exceptions that do not exist in conventional systems. The following briefly summarizes the new exceptions and how to handle them.

- IV failure: The processor raises an exception if the off-chip integrity verification fails. This exception indicates that the system is under physical attack, in which case the recovery from the exception and continuing execution are meaningless. As a result, unlike other exceptions, the AEGIS processor does not support a precise exception for an IV failure. Also, the processor aborts secure execution on this exception, effectively executing `l.aegis.exit`, because the integrity of the exception handler cannot be guaranteed. This exception handler is executed in STD mode, and simply aborts the security kernel with error reports.

- Instruction permission violation: Executing AEGIS security instructions (`l.aegis.*`, `l.puf.*`) in an inappropriate security mode results in an exception. This exception can be handled in the same way that exceptions for executing supervisor instructions in the user mode are treated.

- Memory permission violation: Accessing Verified or Private regions in STD/SSP mode can incur an exception. This exception can be treated the same as conventional permission violations for pages.

6.4 Security System Calls

To utilize the protection provided by the security kernel, user applications must be able to communicate with the security kernel. This section discusses necessary security services that the security kernel must provide as system calls. While the security kernel can provide many more services to user applications, in this section, we focus on the minimum set of system calls that are required to let the applications use the underlying processor features.

Note that only the security system calls need to be incorporated in the security kernel. Other system calls can be implemented outside of the kernel.

6.4.1 Execution Mode Transition

The user-level security mode is determined by the UM bits in the processor, which can be updated only by the security kernel in the supervisory TE/PTR mode. Similarly, the security kernel manages memory protection for user applications. Therefore, for user applications to control their own secure execution, the security kernel must provide system calls to user applications that duplicate the functions of the security instructions provided by the processor to the security kernel.

First, a user application must be able to control its secure execution mode. The following five system calls duplicate the mode transition instructions for user-level applications.

- `sys_aegis_enter()`: A user-mode application can set its initial protection and enter TE/PTR mode with this system call (corresponds to `l.aegis.enter`). This system call gets parameters defining the program hash region, the Verified/Private regions in the virtual memory space, and the initial protection setting such as which mode (TE or PTR) to enter. The system call performs similar operations to `l.aegis.enter`, but in a slightly different manner.

  1. Protect the Verified and Private regions by appropriately setting the access permissions and re-mapping the virtual-to-physical translation. Private regions are mapped to the ME regions, and Verified regions are mapped to the IV regions. For the read-only IV and ME regions, this also requires choosing a new key and initializing the user's read-only IV/ME regions.
  2. Compute the application's program hash ($AHash$).
  3. Enter TE or PTR mode by setting the UM bits.

- `sys_aegis_csm()`: This system call allows user applications to switch between TE mode and PTR mode by simply re-setting the UM bits (corresponds to `l.aegis.csm`).

- `sys_aegis_suspend()`: User applications enter SSP mode with this suspend system call (corresponds to `l.aegis.suspend`). As in the `l.aegis.suspend` instruction, the

system call gets the resume address. The security kernel stores this resume address along with the current UM bits, and changes the UM bits to SSP.

- `sys_aegis_resume()`: Applications resume from SSP mode to a secure mode using this system call (corresponds to `l.aegis.resume`). The system call checks whether the current program counter matches the address specified by the `sys_aegis_suspend()` system call, and restores the saved user mode (UM) bits.

- `sys_aegis_exit()`: This system call exits secure processing for a user application (corresponds to `l.aegis.exit`). The security kernel clears all registers, all private regions in memory and the application's secret keys, and sets the UM bits to STD.

The security kernel should also provide a system call that allows user applications to control memory protection for itself. `sys_aegis_cpmr()` serves this purpose. The system call gets parameters that change Verified and Private regions. If a certain region is changed, the system call re-maps the application's virtual memory so that each region gets proper physical protection with the IV and ME mechanisms. Also, the security kernel changes the permission checks accordingly to protect against software attack.

Obviously, the security system calls must be serviced only when the user application is in an appropriate security mode. Except for `sys_aegis_enter()` and `sys_aegis_resume()`, all other system calls in this section requires the user application to be in either TE or PTR mode. `sys_aegis_enter()` is only permitted in STD mode, and `sys_aegis_resume()` is only allowed in SSP mode.

6.4.2 Private Key Operations

For applications to communicate with remote parties, the security kernel must provide private key operations so that the applications can authenticate themselves and also receive secrets. The security kernel uses the private key instructions, `l.aegis.pksign` and `l.aegis.pkdecrypt`, to implement these services.

The `sys_aegis_pksign()` system call gets an message M as an input, and returns the signature $\{SKHash \| AHash \| M\}_{SK}$. $SKHash$ is the program hash of the security kernel, $AHash$ is the application's program hash, and SK is the processor's private key. Given this signature, a receiver with the corresponding public key can verify that the message is from a specific application running with a specific security kernel on a specific processor.

The security kernel can generate this signature using the `l.aegis.pksign` instruction with the input $AHash \| M$.

The second system call, `sys_aegis_pkdecrypt()`, provides a decryption operation. The system call gets an input $E_{PK}\{SKHash \| AHash \| M\}$, which contains the security kernel's program hash $SKHash$, the application's program hash $AHash$, and a message M encrypted with the processor's public key PK. The security kernel with the correct program hash can decrypt the input using `l.aegis.pkdecrypt`, and returns the decrypted message M only if $AHash$ matches the hash of the executing application.

These system calls can be executed only while the application is in TE or PTR mode. If the system calls are used in an inappropriate security mode, the security kernel raises an exception. Or if `sys_aegis_pkdecrypt()` is used to decrypt a message with a mis-matching program hash, the security kernel returns an error to the application.

6.4.3 Read-Only ME Key Management

Finally, the security kernel must provide one more system call that enables a user application to set its decryption key for the read-only ME region. The `sys_aegis_setkey()` system call gets a symmetric secret key as an input, and sets the user-mode read-only ME key using the `l.aegis.setreg` instruction. Because this system call handles a secret key, it can only be called when the application is in PTR mode.

Chapter 7

Programming Model

Having described the architectural features, this chapter discusses how secure applications can be written in high-level programming languages and compiled for the AEGIS secure processor. High-level abstractions and compilers are critical for programmers to fully exploit the security features provided by our architecture. Without adequate compiler support, programmers cannot be expected to manually control the different security levels for complex real-world applications.

In this chapter, we propose simple programming abstractions that expose the new security features such as different memory protection regions (Verified and Private) and security modes (STD, TE, PTR, and SSP) to application programmers. Given the programming model, we also describe how compilers can generate code to realize the abstractions on the AEGIS processor.

The goal of this chapter is to show one possible programming model and its realization to illustrate that the AEGIS architecture provides an adequate base upon which high-level abstractions can be built. We describe our programming model using the C language. For other languages, the same ideas and techniques shown in this chapter can be applied to develop similar extensions. The programming model design in this chapter has been implemented in the GCC tool-chain as a separate Master thesis project [116].

7.1 Programming Abstractions

The main goal of creating new programming abstractions is to allow programmers to use the new security features without worrying about details of their implementation. There

are three major security features added to our processor architecture as compared to conventional systems. First, the virtual memory space is partitioned into Verified, Private, and Unprotected to provide different protections for program code and data. Second, there are four different security modes, in which a program can execute. Finally, the processor and the security kernel provide private key operations.

Fortunately, the private key operations do not require additional programming abstractions as they can be directly exposed to the programmers as system calls. On the other hand, there is no easy way in the current C language to express different execution modes and levels of protection on memory. Also, it is not a viable option to always execute in PTR mode, and protect the entire memory as Private. This naive approach requires the entire program to be trusted and incurs unnecessary protection overheads. For example, if the security kernel is written this way, the entire operating system in kernel mode must be trusted.

In this section, we describe abstractions that enables programmers to easily control different memory protection levels and switch the security mode.

7.1.1 Memory Protection

Using high level languages, we propose a programming methodology that separates the application's procedures and data structures into three different protection types. This separation is more natural than specifying protections in memory space because programmers usually deal with functions and variables rather than virtual memory directly. From a programmer's perspective, functions and variables will either be Unprotected, Verified, or Private.

- **Unprotected**: Accessible by any part of the application; however, isolated from other processes. No physical protection.

- **Verified**: Readable by any part of the application, but only writable in TE/PTR mode. Integrity is maintained under physical attacks (IV protected).

- **Private**: Accessible only in PTR mode. Both integrity and privacy are maintained under physical attacks (both IV and ME protected).

Effectively, these three types correspond to the protection regions supported by our architecture (see Section 5.2), and determine where the procedures and variables are kept in the memory space. However, note that programmers do not need to choose between read-only and read-write as that distinction should be clear from the context. For example, the program code is read-only whereas the heap is read-write.

To specify the type of functions and global variables, our programming model provides three attributes "unprotected", "verified", and "private". Using these keywords, a programmer can easily express the level of protection. For example, the following pseudo code defines a few functions and global variables.

```
int a unprotected;
int b private;

int func1(int n) verified;
int func2(void) private;
```

The above code declares the global variable `a` as Unprotected and `b` as Private. As a result, `a` can be accessed anywhere in the application, whereas `b` can only be used while in PTR mode. The attributes for functions need a bit more explanation. Here, two functions `func1()` and `func2()` are defined, one as Verified and the other as Private, respectively. These function attributes apply to instructions and read-only data such as strings and constants inside the function. Therefore, adversaries cannot change `func1()`'s instructions and read-only data without being detected. Similarly, the instructions and read-only data that belong to `func2()` are encrypted and can only be executed in PTR mode. Note that Verified functions can execute in either TE or PTR mode.

In case the security attribute is not specified, each software module can have the default attribute. For example, a programmer can specify the default of a library module as Unprotected so that all functions and global variables in the library become Unprotected without specifying individual attributes in the source code.

In addition to global variables, high-level programming languages often have dynamically allocated data structures and local variables as well. For dynamically allocated data structures, the programming model provides three separate heaps (Unprotected, Verified, and Private) and four complementary heap allocation functions (e.g., `malloc`). The first three malloc functions `malloc_unprotected()`, `malloc_verified()`, and `malloc_private()` al-

locate memory in the corresponding heap. The programming model also provides `malloc()` without a particular heap specified. This function selects a heap based on the current execution mode, therefore allowing library functions to use a proper heap without using a special malloc function.

Finally, the programming model also provides three separate stacks (Unprotected, Verified, and Private) for local variables. Unlike previous cases, however, the protection type of a local variable is determined at run-time based on the current security mode that the corresponding function executes in. The local variables are considered as "Verified" in TE mode, "Private" in PTR mode, and "Unprotected" in SSP mode. As we will see in the next subsection, the programming model does not expose STD mode to programmers.

The main reason why we choose the protection for local variables at run-time rather than having programmers annotate them statically is to enable flexible sharing of functions. For example, imagine a function `sum()`, which computes the sum of all elements in an array. This function can be used in many different contexts. Unfortunately, if the local variables are statically determined to be one type (say Verified), this function can only be used for "Verified" arrays in TE or PTR mode. Unprotected arrays in SSP mode and Private arrays in PTR mode each would need their own `sum()` function. By choosing the protection at run-time, one function can be shared for all three types of data.

7.1.2 Execution Modes

Given the specified protection on its code and data structures, an application program must run in an appropriate execution mode (STD, TE, PTR, or SSP) in order to access the protected memory and ensure security. This subsection discusses how the execution mode should be determined in our programming model.

At first glance, it may appear that the protection level on application procedures (Unprotected, Verified, or Private) simply determines the security mode in which each function should execute. Indeed, Unprotected code must run in SSP mode to ensure security, and Private code must run in PTR mode to be decrypted. However, Verified code can be used in either TE, PTR, or SSP mode. For example, one Verified function can be shared by many procedures in TE or SSP mode. If this Verified function is executed only in TE mode, a programmer will need to create another Unprotected copy of the same function for SSP mode. Also, Verified functions will often be used to process Private data because Private

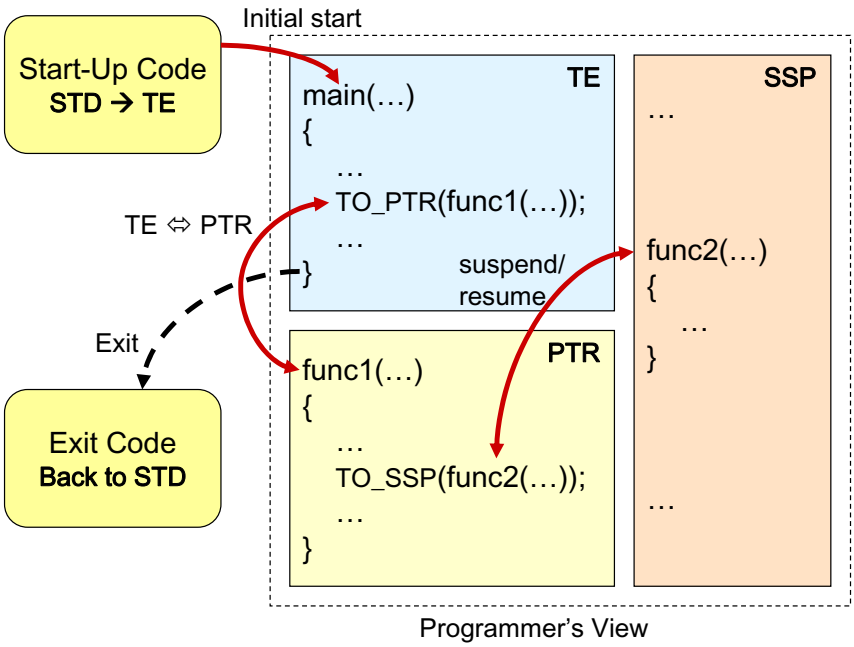


Figure 7-1: The programming abstractions for execution mode transitions. Applications start in TE mode, and the mode changes on an function call with special directives.

functions pay overheads of decrypting code. Therefore, the execution mode should be determined based on the context in which each function is called rather than the protection level of the function.

Figure 7-1 illustrates how the execution mode is managed in our programming model. In the model, STD mode is hidden from a programmer who only manages TE, PTR, and SSP modes. The compiler inserts start-up code that enters TE mode and calls the main function (`main()`). Thus, from the programmer's perspective, the program starts at the main function in TE mode. Because SSP mode provides the same protection and overheads as in the STD mode, there is no reason to expose the STD mode to application programmers.

Once started in TE mode, a programmer can change the execution mode at a function call boundary. This boundary is a natural abstraction for programmers as a function represents a basic unit of computation and is protected separately in memory. To execute a callee function in a specific mode, a programmer uses one of the following directives on a function call.

- `TO_TE(function call)`: Execute the callee function in TE mode with Verified stack.
- `TO_PTR(function call)`: Execute the callee function in PTR mode with Private

stack. Private functions must be executed in PTR mode.

- `TO_SSP(function call)`: Execute the callee function in SSP mode with Unprotected stack. Unprotected functions must be executed in SSP mode.

The directives only mean that the callee function executes in a new execution mode. They do not affect the execution mode, in which the caller function executes. Once the callee returns, the execute mode also returns to the caller's original mode. Also, note that the above directives can only be used in TE or PTR mode, not in SSP mode. There is no need for a special directive in SSP mode as the return from the function in SSP mode to the caller in TE or PTR mode implicitly works as a "resume".

Our function call directives determine the callee's execution mode based on the context where it is called. For example, a function `sum()` can be called in three different contexts using `TO_TE(sum())`, `TO_PTR(sum())`, `TO_SSP(sum())` and execute in three different modes.

Finally, the use of the mode transition directives are checked both statically at compile time and dynamically at run-time. For example, if a Private function is called without `TO_PTR()` directive in Verified function, the compiler will generate an error. Similarly, the compiler checks if each call from a Verified or Private function to an Unprotected function uses `TO_SSP()`. These checks ensure that the programmer knows and intends the mode transition. At run-time, if the directives are used while in SSP mode, the processor raises an exception.

7.1.3 Declassification

With the above abstractions, programmers can express the protection level on functions and variables, and control the execution mode. To complete our programming model, however, there is one remaining issue regarding the declassification of protected data. In our model, there are three types of data: Unprotected, Verified, and Private. To be practical, applications must be able to move data from a high security level (Private) to a lower security level (Verified, Unprotected) so that the result of a secure computation can be communicated to the outside world.

It generally depends on applications whether a certain declassification of protected data from a high security level to a lower level should be allowed or not. Therefore, programmers should make a decision when to declassify protected data. The role of the programming

model is to provide declassification methods to the programmers.

Ideally, the programming model along with static compiler checks can be carefully designed to ensure that programmers cannot declassify data in an unintended way. Previous efforts in this area such as JFlow [87, 115] address this problem. In our programming model, however, we provide a simple declassification method since the goal is to design an example model. More compiler support can always be added to enhance the basic model.

In our model, a programmer simply assigns a high security variable into a low security variable for implicit declassification. Obviously, the program should be in a security mode with permission to access the high security variable. For example, say that there are two variables `a` and `b`, which are Private and Verified, respectively. To declassify the Private value in `a`, a programmer simply writes `b = a`, so that the value gets copied and becomes Verified. Similarly, `a` can be used as a function argument as in `TO_TE(func1(a))`, and becomes declassified.

The assignments only declassify the values that are copied. If a Private variable contains a pointer to a Private array, passing the pointer as an argument to a function in TE mode will not declassify the array. In order to allow accesses to the Private array in TE mode, a programmer should explicitly copy the array into another Verified array.

7.2 Implementation

This section describes how compilers can implement the programming model presented in the previous section. We first briefly describe the overall tool flow, and discuss how procedures and variables with different protection are mapped to the protection regions in the application's memory space. Then, we explain how the execution mode transitions are handled at start-up and on a function call. Finally, we briefly discuss how multiple decryption keys can be used while the processor only supports one key at a time.

7.2.1 Compilation Flow

Before we dive into the details of our compiler extensions, let us briefly consider the overall compilation flow shown in Figure 7-2. First, a programmer writes an application program using the programming abstractions we described and puts the protection attributes and the mode transition directives in the program. Then, the compiler generates the bi-

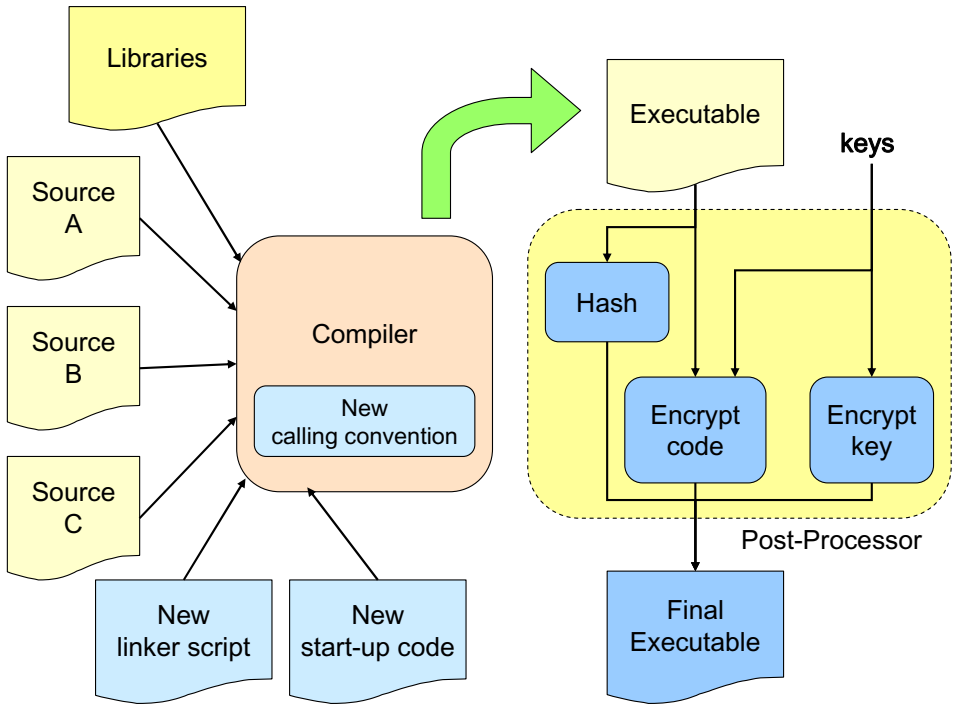


Figure 7-2: The compilation flow for our programming environment.

nary executable from the source code. Our compiler needs three new components for this compilation phase. First, the linker script has to change to understand different types of protection regions in the virtual memory space (Section 7.2.2). Second, the start-up code must contain new security features to start the application in TE mode (Section 7.2.3). Finally, the mode transition on a function call needs new extensions to the existing calling convention (Section 7.2.4).

Unlike conventional programs, the executable generated from the linker needs to be processed again in order to generate the executable for our secure processor. This post-processor performs two important operations. First, this tool computes the hash of the Private code and Private/Verified initialized data, and stores the hash in the Verified section of the executable. Then, the tool encrypts the Private code using a symmetric secret key supplied by the user. Finally, this symmetric key is encrypted with the public key of the processor given as an input to the tool. The encrypted key is stored in an Unprotected section of the executable. Both the hash and the encrypted key can be accessed by the start-up code as global variables. Section 7.2.3 discusses the details of where they are used.

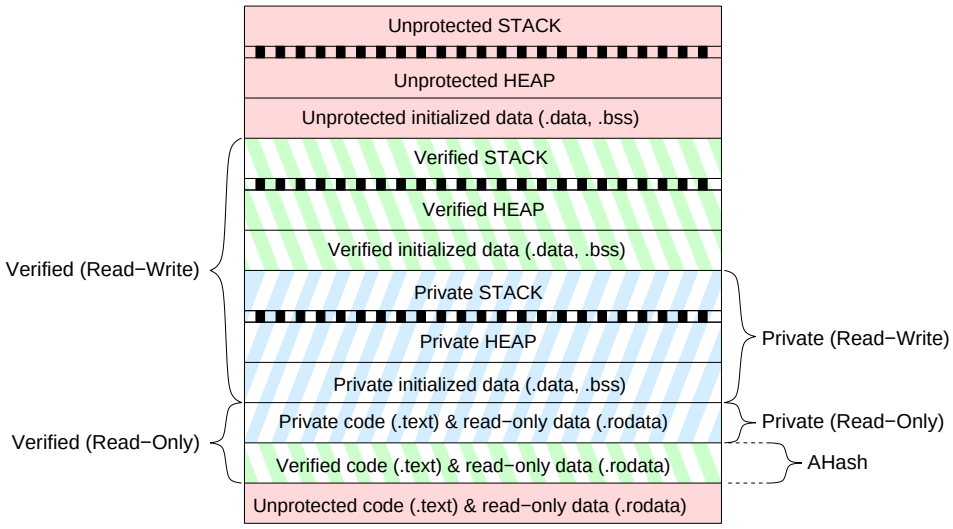


Figure 7-3: Typical program layout using four protection regions in the virtual memory space.

7.2.2 Memory Layout

Given the programming methodology, we depict in Figure 7-3 one way in which a compiler could arrange four protection regions in the virtual memory space and map three types (Unprotected, Verified, and Private) of code and data of an application to those regions. At the bottom of memory we have the read-only Unprotected region, which is not included in any of the protection regions. Here, code from the procedures marked as Unprotected is stored. The code includes instructions and also read-only data such as strings or constants. Above that unprotected region, we have the read-only Verified region, which includes code from Verified procedures. The top portion of this Verified region is also set as the read-only Private region and includes code from Private procedures. After the read-only regions, we have Private, Verified, and Unprotected read-write regions. Each read-write region contains global variables, the heap, and the stack of the corresponding type.

Here, Verified and Private regions provided by the AEGIS processor architecture are sufficient to naturally map all three types of code and variables in the programming model. Also, as noted in the programming model, this layout results in three different heaps and stacks. The base addresses of the three heaps are passed to the memory allocation functions so that each heap can be allocated separately. For three stacks, the compiler maintains three global variables called `sp_unprotected`, `sp_verified`, and `sp_private` that contains three

stack pointers. Each stack pointer variable is stored in the corresponding read-write region. As we will describe in the following subsections, the compiler moves an appropriate stack pointer into the stack pointer (SP) register to be used by the executing function.

7.2.3 Start-Up Code

Just like the execution start-up routines such as `crt0.o` in conventional systems, our programming model requires the secure start-up code to set up the secure TE environment before executing the main procedure. In addition to conventional jobs, however, our start-up code must also manage three stack pointers, specify the protected memory regions, and enter TE mode. This subsection summarizes the operations of the start-up code, and explains how the program hash can be computed for the application whose code is distributed in various protection regions.

The start-up code is placed in the read-only Verified region in Figure 7-3 so that its integrity is assured once the program enters TE mode. The code is public and should not be encrypted so that the processor can execute it without knowing the program hash. Also, we assume that the linker script is properly written so as to reflect the memory layout in the previous subsection. Therefore, the start-up code has accesses to symbols that represent the addresses of the protection region boundaries. In the sequel, we detail steps that are performed in the start-up code.

1. Prepare the parameters for the `sys_aegis_enter()` system call. The boundaries for Verified and Private regions are from the linker script based on our memory layout. The program hash region is set to cover only the read-only Verified code region (but *not* Private code), which includes the start-up code itself (see Figure 7-3). Finally, the protection is set to enter PTR mode with debugging disabled.

2. Call `sys_aegis_enter()` to enter PTR mode. At this point, the program hash $AHash$ is computed by the security kernel to identify the application.

3. Check if the current program counter represents the correct entry point. If not, abort the application.

4. Set three stack pointer variables `sp_unprotected`, `sp_verified`, and `sp_private` to point to the top of each stack. Set the SP register to be `sp_private` so that the

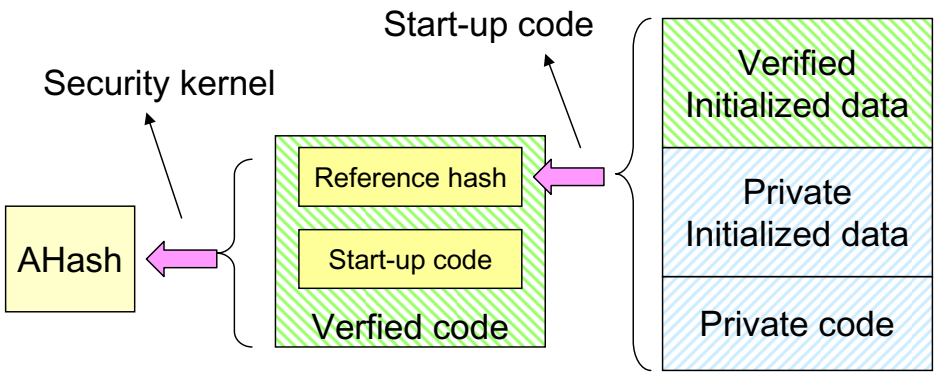


Figure 7-4: The summary of the application identification process using program hashes. The security kernel only identifies the Verified code, which contains the hash of the rest of the application.

start-up code uses the Private stack. This step needs to be in assembly, but the rest of the start-up code can be programmed in a high-level language.

5. Clear the `.bss` sections in the read-write memory regions.

6. If there is Private code, which is encrypted, set the key with the `sys_aegis_setkey()` system call. The key can be obtained using the `sys_aegis_pkdecrypt()` system call with the encrypted key stored in the Unprotected memory segment by our compiler tool (see Section 7.2.1).

7. Compute the hash of the Private code and the Verified and Private initialized data in Figure 7-3. Compare the hash with the one stored in the read-only Verified region (read-only data) that was computed at compile-time (see Section 7.2.1). If the hashes do not match, abort.

8. Enter TE mode using `sys_aegis_csm()`. Switch the stack from the Private one to the Verified one.

9. Jump and link to `main()`.

One point to notice is that the entire application can be uniquely identified with the *AHash* computed only over the public part of the read-only Verified region containing Verified code and read-only data. While the rest of the code and data is not directly included in the *AHash*, the public Verified region contains the hash of the other protected

regions, which is computed at compile-time (reference hash). As a result, the $AHash$ reflects the contents of all protected regions, and identifies the entire application. The start-up code ensures that the rest of the code and data has not been changed by computing the hash at run-time and comparing this hash with the reference hash value (Step 8). Figure 7-4 summarizes this approach.

In addition to only requiring the security kernel to identify one continuous memory region, this hashing approach has an additional benefit when some of the code is encrypted. In our architecture, the private key operations can be performed only after the application is identified using its program hash. If the security kernel includes the Private region in $AHash$, the hash value of the same application will be different for each processor because the encrypted code will be hashed without decryption. In our approach, however, each application always results in one program hash no matter where it executes.

7.2.4 Mode Transition

Our programming model provides three directives for function calls so that a programmer can change the secure execution mode of an application. For such function calls with a mode transition, the standard calling convention must be modified so that the execution mode and the stack can be securely switched before and after the function call. This subsection discusses how the compiler handles function calls with the mode transition directive.

In the following discussion, we use a typical calling convention to explain the modifications for each mode transition directive. This typical calling convention passes both arguments and a return value through the stack, and divides the work between the caller and callee by having caller-saved and callee-saved registers. Briefly, the caller saves the caller-saved registers, reserves a place for the return value in the stack, and puts the function arguments in the stack before the function call. The callee saves and restores the callee-saved registers, and puts the return value in the stack. Once the callee returns, the caller copies the return value, restores the stack pointer and caller-saved registers. We will refer to this as the "standard" calling convention.

For all directives, note that we can only change the caller part of the calling convention. To allow the same function to be called from many different places with different directives, the callee's convention should be common for all cases. Otherwise, there must be different copies of the same function depending on where it is called.

Now let us consider the function call with the `TO_PTR()` directive, which specifies that the callee should execute in PTR mode. Obviously, this transition is only relevant when the caller is in TE mode. For the mode transition from TE to PTR, the compiler changes the calling sequence as follows.

1. Confirm that the application is in TE mode. If in PTR mode, simply follow the standard calling convention. If in STD or SSP mode, the directive will result in an exception.
2. Save the set of registers that are caller-saved in the Verified stack.
3. Change to PTR mode using the `sys_aegis_csm()` system call.
4. Reserve the place for a return value in the Private stack by decrementing the Private stack pointer. The Private stack pointer needs to be loaded from `sp_private` in memory. Note that we still keep the Verified stack pointer (the current one) in the SP register.
5. Put the function arguments into the Private stack.
6. Switch the stack pointer from Verified to Private. Save the SP register, which contains the Verified stack pointer, into `sp_verified` in memory. Set the SP register to be the Private stack pointer used in the previous steps.
7. Jump and link to the callee function.
8. Once the callee returns, copy the return value from the Private stack to a variable accessible by the caller.
9. Restore the Private stack pointer to the value before the function call (pop all the function arguments and the return value).
10. Switch the stack pointer from Private back to Verified. The Private stack pointer is saved to `sp_private` in memory, and the Verified stack pointer is loaded from `sp_verified` in memory.
11. Change the mode back to TE.
12. Restore the caller-saved registers.

Besides changing the mode from TE to PTR, this function call sequence has only one modification to the standard calling convention. In our programming model, the protection type of local variables is determined based on the execution mode. The caller in TE mode should use the Verified stack whereas the callee in PTR mode should use the Private stack. Therefore, the stack pointer should be switched from Verified to Private, and function arguments must be passed in the Private stack for the mode transition to be transparent to the callee. Once the callee returns, the caller must switch back the stack.

In the above sequence, we note that the callee in PTR mode does not clear the caller-saved registers. As a result, private data may remain in registers when the callee returns. However, the caller overwrites those potentially private register values in the last step by restoring the previous values. Because the caller in TE mode is trusted and verified with the program hash, the private values that may remain in the registers for a short period in TE mode cannot be read by adversaries.

The `TO_TE()` directive can also be implemented in a similar manner. Besides switching the stack in the opposite direction from Private to Verified, the calling sequence for `TO_TE()` has two differences compared to the case of `TO_PTR()`. First, the execution mode must be changed from PTR to TE *after* the first stack switch, and from TE to PTR *before* the stack switches back. The application must be in PTR mode to access both stacks when it switches the stack pointer. Second, the caller must save and clear *all* registers rather than just caller-saved ones to ensure privacy. The following summarizes the call sequence for `TO_TE()`.

1. If already in TE mode, follow the standard calling convention. Otherwise, proceed to Step 2.

2. Save *all* registers in the stack (the Private stack), and clear the registers. Thus, the callee-saved registers will be saved again in the callee function. However, this step is necessary to ensure the privacy of registers in PTR mode. The callee will read and save the registers in the Verified stack, whose privacy is not protected.

3. Reserve the place for a return value in the Verified stack.

4. Put the function arguments into the Verified stack.

5. Switch the stack pointer from Private to Verified.

6. Change to TE mode using the `sys_aegis_csm()` system call.
7. Jump and link to the callee function.
8. Once the callee returns, change the mode back to PTR.
9. Copy the return value from the Verified stack.
10. Restore the Verified stack pointer to the value before the function call.
11. Switch the stack pointer from Verified to Private.
12. Restore the saved registers.

Finally, the `TO_SSP()` directive can be handled similar to `TO_TE()` since they both lower the security level. However, transitioning to SSP mode requires one additional protection measure to ensure that the secure execution can resume only at the specific point. The compiler uses `sys_aegis_suspend()` and `sys_aegis_resume()` rather than `sys_aegis_csm()` to change the execution mode. We summarize the call sequence for `TO_SSP()` below.

1. Check and memorize the current execution mode (TE or PTR). This information is used to determine what the current stack is.
2. Save *all* registers in the stack, and clear the registers.
3. Reserve the place for a return value in the Unprotected stack.
4. Put the function arguments into the Unprotected stack.
5. Switch the stack pointer to Unprotected (from Private in PTR mode, from Verified in TE mode).
6. Change to SSP mode using the `sys_aegis_suspend()` system call. As the resume address, use the address of Step 8.
7. Jump and link to the callee function.
8. Once the callee returns, change the mode back using `sys_aegis_resume()`. This system call checks the resume address.
9. Copy the return value from the Unprotected stack.

10. Restore the Unprotected stack pointer to the value before the function call.

11. Switch the stack pointer back to Verified/Private.

12. Restore the saved registers.

7.2.5 Multiple Decryption Keys

So far we assumed that all Private functions in the application are encrypted with a single key. As a result, the decryption key for a read-only ME region is set once in the start-up code and stays the same throughout the entire execution. It is relatively straightforward, however, to have different sections in Private code encrypted with different keys. For example, in a light-weight software copyright protection scheme, each library may be given to the developer after encryption with its own secret key. In this situation a compiler can add instructions which cease executing from the first library, swap the two decryption keys, and then continue executing code within the second library. The standard stack calling convention is all that would be necessary in this situation since the execution will never leave PTR mode.

Chapter 8

Extensions and Variants

This chapter discusses extensions and variants to the architecture described so far. First, we present two new extensions for secure booting and symmetric-key generation, which can add new functionality to the existing architecture. The secure booting ensures that only an authorized security kernel can run on the processor, and the new PUF instructions can generate symmetric keys so that the processor can be authenticated without expensive public-key cryptography. The second half of this chapter describes two alternative processor designs. We first discuss how the security features can be further moved into the processor to the point where the security kernel is completely removed. Next, we propose a new integrity verification mechanism that has a significant performance advantage over the cached hash tree for certain types of applications.

8.1 Secure Booting

Our baseline processor design targets an open computing system, and allows any software to execute on the system. Just like conventional computers, anyone who has physical access can reboot the system and install a new operating system and a new security kernel. This property is not a problem for most applications as the software can be authenticated by remote parties and secret information can be encrypted for particular software. For example, a content provider can send an encrypted media file to a trusted player in a way that malicious software cannot decrypt the file even on the same processor.

On the other hand, there exist proprietary systems where manufacturers or vendors should be able to restrict the software that can execute on the system. For example,

only the manufacturer should be able to install software on the embedded computers in automobiles. If automobile owners can re-program an engine control system to increase the performance, it could result in a safety hazard. Also, cell phone vendors may want to control the software on the phone and charge fees for adding new software features to the phone. Similarly, owners should not be able to change software on game consoles and bypass copy-protection mechanisms.

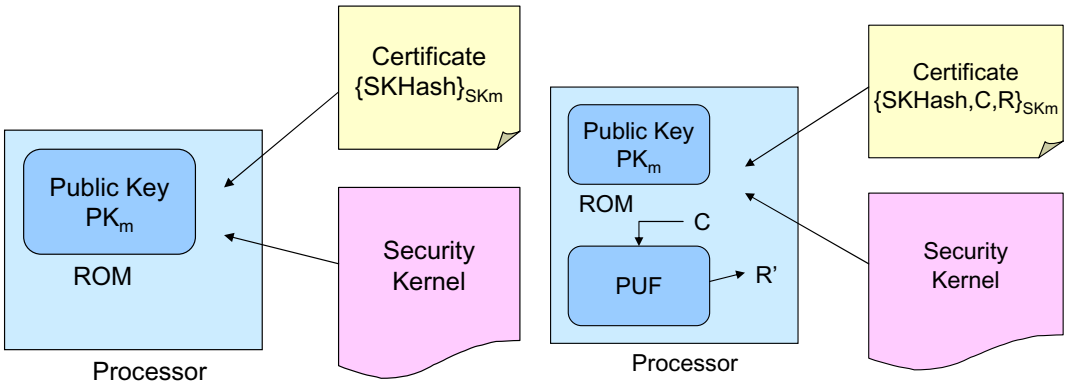
This section discusses the architectural changes required for *secure booting* of such proprietary systems. The goal of this secure booting mechanism is to only allow security kernels authorized by a manufacturer to control the system functions. Once accepted by the processor, the security kernel can implement similar schemes to restrict the user applications that can execute on the system.

Note that the goal is to prevent unauthorized programs not from simply running on the processor, but from controlling the system. For example, simply having a processor executing a malicious instruction stream on the game console would be useless to the adversary if that unauthorized program cannot control the console's display or sound system. Here, we assume that the processor is built so that I/O functions to control other components of the system can only be enabled in the supervisory TE/PTR mode. Therefore, our secure booting problem can be stated as follows; only allow an OS kernel to enter a secure mode if it possesses an appropriate credential from the manufacturer.

8.1.1 Simple Public-Key Approach

One possible approach to the secure booting problem is to embed a manufacturer's public key PK_m in the processor as shown in Figure 8-1 (a). Because the public key is not a secret, inexpensive ROM can be used to reliably store it on-chip without increasing the cost. Note that the public key in ROM is extremely difficult to change because the bits are hardwired.

Given the public key, the `l.aegis.enter` instruction can be slightly changed to check the credential of the OS kernel to control the entrance to the secure mode. In a simple case, the credential to run the program is a certificate that is the kernel's program hash signed by the manufacturer's private key. The `l.aegis.enter` gets the certificate as one of its inputs. In the start-up process, the processor checks if the hash computed for the current kernel matches the hash in the certificate, and also verifies the certificate with the embedded public key. The kernel is allowed to enter TE mode only if both checks pass.



(a) A baseline secure booting scheme (b) Extension for per-device secure booting

Figure 8-1: The secure booting mechanisms to restrict software controlling the secure computing system.

8.1.2 Per-Device Authorization

The above scheme ensures that only the kernel that is signed by the manufacturer can enter a secure mode and control the system functions. However, the manufacturer cannot control individual devices. Because the same public key is embedded on all processors from the same mask, the manufacturer can only authorize a particular kernel for either all of those devices or none of them.

To solve this problem, each processor must have an unique unchangeable ID. Here, we use the PUF responses as IDs. The PUF is a perfect candidate for this purpose, because it is extremely difficult to change the analog delay characteristics of the circuit. Also, the PUF does not require any programming.

Figure 8-1 (b) illustrates this secure booting mechanism with the PUF ID. Now, the certificate from the manufacturer contains three components: the kernel's program hash $SKHash$, a PUF challenge C, and the response R. In the `l.aegis.enter` instruction, the processor verifies the certificate, checks the program hash, generates a response R' from the PUF, and compares R and R' to ensure that they match. We note that the PUF does not need error correction because the processor can simply ensure the difference between R and R' is within a certain bound. In this way, the processor can check if the kernel is authorized for that particular device.

8.2 PUF Instructions for Symmetric Keys

The baseline architecture provides two PUF instructions, `l.puf.pksave` and `l.puf.pkload`, so that the processor can securely express a private key using a PUF. The use of a private key allows a single public key to be shared by many users. On the other hand, the private-public key operations are much more expensive compared to the corresponding symmetric key operations. Therefore, expressing symmetric keys with a PUF may be desirable for resource constrained embedded processors. For example, if the secure processors are used in sensor networks, their energy consumption is a major concern, which makes in turn symmetric key operations much more desirable.

This section presents two additional PUF instructions which allow the security kernel to bootstrap a unique challenge-response-pair and generate a unique symmetric key from the PUF. Using these instructions, each user can share a unique symmetric key with each security kernel (and user application) on the secure processor. These instructions are based on the protocol developed in previous work [39], but extended to incorporate error correction. We first describe the instructions provided for the security kernel, and then explain how the security kernel uses those instructions to provide the corresponding system calls to user applications.

8.2.1 Security Kernel

The processor provides the `l.puf.response` instruction so that a security kernel can obtain a secret PUF response. However, this instruction should not allow malicious users, who can even run their own kernels, to obtain a specific Challenge-Response Pair (CRP) used by another user.

To address this, `l.puf.response` does *not* let a kernel choose a specific challenge as shown in Figure 8-2. The input to the instruction is $PreC$, called "pre-challenge", rather than challenge C. The processor computes C by hashing the concatenation of $SKHash$ (the program hash of the security kernel) and $PreC$. Thus, `l.puf.response` returns the response R and the syndrome S as follows.

$$(R, S) = \texttt{puf\_calibrate}(C) = \texttt{puf\_calibrate}(H(SKHash \| PreC))$$

where $H()$ is an ideal cryptographic one-way hash function, and $\|$ represents the concate-

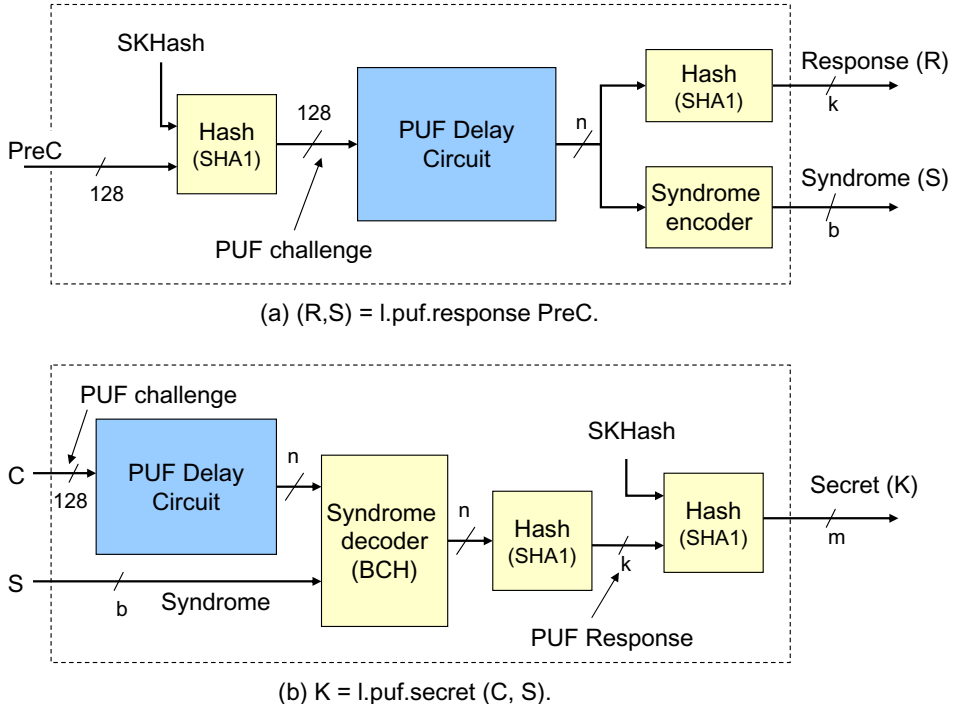


Figure 8-2: The PUF instructions to generate symmetric keys.

nation. As a result, a malicious kernel cannot obtain the response for a specific challenge C using `l.puf.response`. To do this, the malicious kernel would have to find the input $PreC'$ that produces the challenge $C = H(SKHash' \| PreC')$ for its program hash $SKHash'$. (This is equivalent to finding a collision in the one-way hash function $H()$.)

A second instruction, `l.puf.secret`, allows the security kernel to share a symmetric secret key with a user who already knows a CRP. This instruction takes a challenge C and a syndrome S as inputs and returns an m-bit secret K by hashing the PUF response R concatenated with the program hash $SKHash$,

$$K = H(SKHash \| R) = H(SKHash \| \texttt{puf\_regenerate}(C, S)).$$

The secret is the cryptographic hash of the program hash $SKHash$ concatenated with the PUF response $\texttt{puf\_regenerate}(C)$. Knowing the CRP, a user can easily compute the secret K for a trusted security kernel with $SKHash$. On the other hand, a malicious kernel cannot obtain the same secret from the processor because its program hash is different. Also, it is impossible to determine the response R from a secret K since it requires inverting the one-

way hash function. Therefore, this symmetric key is *unique* for a specific security kernel on a specific processor, and effectively authenticates both the security kernel and the processor.

8.2.2 User Applications

Just like the private-key operations, the new PUF instructions must be exposed to user applications so that they can generate their own symmetric keys to share with users. A user-level application is given access to the PUF via a system call to the security kernel `sys_puf_response`(*UserPreC*). The system call uses the `l.puf.response` instruction with input $PreC = H(AHash \parallel UserPreC)$ so that the challenge depends on both the security kernel and the user application ($AHash$ is the program hash of the application). The system call returns the response R and the syndrome S from the `l.puf.response` instruction.

The security kernel provides another system call `sys_puf_secret`(C, S) to user applications so that each application can generate a unique secret. The system call takes a challenge C and a syndrome S as inputs, and returns $H(AHash \parallel$ `l.puf.secret`$(C, S))$ so that the secret key depends on both the security kernel and the application. Therefore, this secret key effectively authenticates the security kernel and the user application as well as the processor.

8.2.3 Bootstrapping

Using `sys_puf_response()`, a user can securely bootstrap a unique CRP from the processor. In the absence of an eavesdropper, the user can use a randomly chosen $UserPreC$, and obtain the response in plaintext. This user can easily compute the challenge $C = H(SKHash \| H(AHash \| UserPreC))$. In this case, $UserPreC$ should be kept secret so that malicious users cannot use the same application with the same security kernel and generate the same CRP.

Bootstrapping can also be accomplished securely using private/public key cryptography even when eavesdropping is possible. A user runs an application which (1) obtains a response with an arbitrarily chosen $UserPreC$, (2) encrypts the response with his public key, and (3) outputs the encrypted response. Even though an eavesdropper can see the encrypted response, only the user can decrypt the response using his private key. Also, malicious users cannot use the same program with their own public key because changing the public key results in a different program hash $AHash$ for the application. $UserPreC$ can be public

in this case because knowing $UserPreC$ does not help in discovering the response if the private key of the user is kept secret.

8.2.4 Example

To illustrate how these symmetric-key operations can be used to authenticate the system, let us consider the distributed computation example discussed in Section 2.1. Using the symmetric key system calls, the distributed computation can be re-written as follows.

```
DistComputation()
{
  x = Receive();                // receive Alice's input
  result = Func(x);             // compute

  key = sys_puf_secret(C,S);    // get a PUF secret (known C)
  mac = HMAC(key, (result,x));  // sign the result

  Send(result,mac);             // send the result
}
```

Here, the parts that receive an input, perform the computation, and send the result back are exactly the same as before. On the other hand, the part to generate a private-key signature is replaced by two new operations that obtain a PUF secret key using `sys_puf_secret` and compute the message authentication code with the symmetric key.

Let us assume that Alice shares a unique PUF challenge-response-pair (CRP) with the processor using the `sys_puf_response()` system call. Within `DistComputation`, the `sys_puf_secret()` system call generates a secret key (`key`) depending on $SKHash$, $AHash$, and the processor's PUF response R. Modifying either the application or the security kernel, or executing the application on a different processor will cause the `key` to change. Since Alice knows both $SKHash$ and $AHash$ as well as R, she can verify, using the `mac` from Bob, that `DistComputation` was executed on Bob's computer.

8.2.5 Additional Attacks on the PUF

Unlike the private-key instructions, which do not reveal any PUF response to an adversary, the new PUF instructions in this section allow an adversary to obtain many challenge-

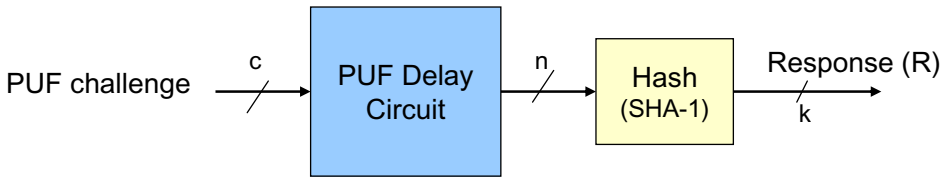


Figure 8-3: The output hash function to prevent model building. The attacker needs to inverts the one-way hash function to model the PUF circuit even if he can obtain many input/output pairs.

response pairs. The ability to obtain many CRPs poses a couple of additional threats to the PUF that are not possible for the private-key PUF instructions.

First, attackers can enquire the PUF many times and try to obtain all possible challenge-response pairs (CRPs). However, this is infeasible because there are an exponentially large number of challenges. For example, with 128-bit challenges, the attacker must obtain 2^{128} CRPs.

Second, because the PUF circuit is rather simple, attackers can try to construct a precise timing model and learn the parameters from many challenge-response pairs. However, the model building from CRPs is impossible because the PUF circuit output is never directly revealed to an adversary. As shown in Figure 8-3, the PUF response is the cryptographic hash of the delay circuit output. Therefore, to learn the actual circuit outputs, the attackers must invert a one-way hash function, which is computationally intractable.

On the other hand, the new instructions can also reveal many syndromes that is computed directly from the delay circuit output. Fortunately, each bit in the BCH syndrome depends on many input bits making it difficult for adversaries to infer the PUF delay circuit output from the BCH syndrome bits. However, further studies are required to evaluate this threat more thoroughly.

One simple way to eliminate the threat of the model building is to only generate one secret key from the PUF delay circuit even for the symmetric-key instructions. For example, if the PUF circuit generates one key K, the PUF can be implemented using a cryptographic hash function $PUF(C) = H(C\|K)$. In this way, only one syndrome is revealed to an adversary.

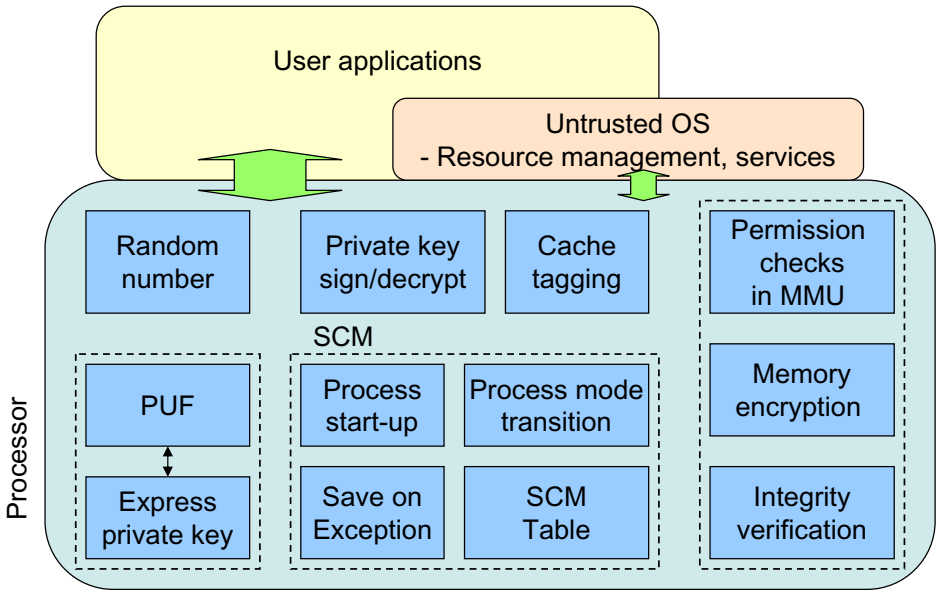


Figure 8-4: The overview of the system architecture without the security kernel.

8.3 Removing the Security Kernel

The baseline AEGIS architecture relies on the security kernel to manage memory protection of multi-tasking user applications in the face of software attacks. This design only requires minor modifications to the processor architecture, and provides flexibility because the software security kernel can be easily changed.

We can move the security functions of the security kernel to the processor to further reduce the Trusted Computing Base (TCB). Obviously, the removal of the security kernel feature comes at the cost of more complex processor architecture, which needs to be verified and trusted. As a result, there is a trade-off between the complexity of the trusted security kernel and the complexity of the processor. In fact, the security features required for secure computing systems can be partitioned between the security kernel and the processor in many different ways.

Here, we discuss how the security kernel can be completely removed, with its functions implemented in the processor. We call this the *Untrusted OS Solution*. In this approach, the processor handles all security-related functions whereas the operating system only manages resources and provides other services. Therefore, the security instructions (`l.aegis.*`, and `l.puf.*`) are directly provided to each process (both supervisor processes and user applica-

tions), not only to the operating system. As discussed in Chapter 6, the security kernel must provide the context manager, the exception handler, the virtual memory manager (VMM), and the security system calls to the user applications. In the following subsections, we discuss how each of these functions can be supported directly by the processor.

8.3.1 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 a secure mode (TE/PTR/SSP), so that it can securely manage their states. We introduce a *secure context manager (SCM)*, which is a specialized component in the processor that replaces the functions of the context manager in the security kernel and 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. Also, the SCM maintains the ID of the active secure process ID (the currently executing one).

SCM Table

The SCM maintains a table that holds protection information for each secure process running in a secure mode. The table entry for a process consists of a SPID, the program hash $AHash$, the architectural registers ($Regs$), a root hash used for memory integrity verification, the UM bits indicating the process' security mode, Verified/Private regions in the virtual memory space, and the keys for the IV/ME regions (K_{RW}, K_{RO}, and K_{IV}). The resume address and the previous execution mode on `l.aegis.suspend` are also stored in this table. We refer to the table as the SCM table. An entry is created by the `l.aegis.enter` instruction, and deleted by the `l.aegis.exit` 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 malicious operating system to impersonate the application that it killed.

The SCM table can be entirely stored on the processor as in XOM [73], 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.

Exception Handling

Exceptions such as interrupts, faults, traps require rather complicated services such as scheduling, error reporting, and system calls. Therefore, rather than having the processor handle these complex tasks in hardware, we let the untrusted operating system manage multitasking.

The processor nevertheless has to protect a secure process' state. For that reason, on an exception, the secure context manager performs a sequence of operations before it transfers control to the exception handler in the untrusted operating system. First, if there is an IV failure exception, the SCM kills the current secure process. Otherwise, the SCM stores the interrupted secure process' registers including the program counter in the SCM table. The root hash for the cached hash tree is also saved in the SCM table. Other security related state such as the program hash and the UM bits are already in the SCM table. Once the register values are stored, the SCM clears the working copy of the registers including secret keys for off-chip protection so that the untrusted OS cannot read private information. Finally, the SCM changes the active SPID to zero indicating a regular process, and jumps to the exception handler.

Now the exception handlers in the untrusted OS are free to perform their operations. To restart an interrupted secure process, the untrusted OS calls the `l.aegis.restart` instruction with the SPID. Then, the SCM restores the saved state for that secure process and jumps to the saved program counter address. Thus, the OS can choose which process to run but cannot tamper with the secure process' state.

8.3.2 Virtual Memory Protection

In the security kernel solution, the program's state in the virtual memory space is protected in three different ways. First, the conventional VM mechanism and the additional access checks prevent software attacks. Then, the off-chip memory protection mechanisms prevent physical attacks on the external RAMs. Finally, the VMM in the security kernel protects swapped pages stored in the secondary storage.

Unfortunately, it would be too complex and inflexible to implement the VMM functions such as the virtual-to-physical mapping and the protection of the swapped pages in hardware. Instead, we protect the secure process' virtual memory by adding tagging mechanisms in the on-chip caches and applying the off-chip protection mechanisms to virtual memory space covering both external RAMs and swapped pages in secondary storage. The Private/Verified regions within the virtual memory space are protected by the same access checks in MMU.

In this new approach, all but the additional access checks of the conventional virtual memory manager are part of the untrusted OS. As a result, a malicious OS can change the virtual-to-physical mapping or change the contents of swapped pages at will. However, our protection mechanism will detect the attack if such tampering occurs.

On-Chip Cache Protection

The on-chip caches are protected using tags. Whenever a process accesses a cache block within its Private/Verified regions, 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 a cache block in the cache that requires integrity protection (in the Verified regions), the processor checks the block's tag 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. If the verification is successful, 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.

1. Initially, a virtual address VA is mapped to a physical address $PA1$.

2. 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.

3. The operating system changes the mapping so that VA corresponds to $PA2$ that contains the same values with $PA1$.

4. 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.

5. The operating system changes the mapping back so that VA maps to $PA1$.

6. Program A reads from VA, which returns a stale value in $PA1$.

In order to prevent replay attacks, the processor should only allow one block for each virtual address of a process to be in the cache. 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.

Off-Chip Protection

For off-chip memory including both external RAM and secondary storage, we use the hardware memory integrity verification and encryption mechanisms in Chapter 4. Unlike the security kernel solution, however, the mechanisms are applied to each secure process' virtual memory space. For example, each secure process has a separate hash tree to protect its own virtual memory space. The root hash is stored in the SCM table, and switched by the SCM on an interrupt and a resume. Changes made by a different process or tampering of the virtual-to-physical mapping are detected by the hash tree. Because we are protecting virtual memory space, pages are protected both when they are in RAM and when they are swapped to disk.

Applying the protection algorithm to the virtual memory space is relatively straightforward. The algorithms simply use virtual addresses in computing MACs, hashes or AES. The only non-trivial problem is determining the physical address of meta-data.

In our solution, an L2 cache block contains its virtual address and the owner process' ID. Note that the cache does not have to be virtually-addressed. Either on a cache eviction or on a cache miss when the off-chip protection units need to obtain the address of the meta-data, the processor uses the mapping described in Section 4.4.1 and computes the virtual address of the corresponding meta-data. Finally, the processor converts this virtual address to the physical address. 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.

8.3.3 Security Instructions

Because there is no security kernel, the processor provides the security features directly to applications as instructions. Here, we briefly summarize the security instructions required to support the execution of a secure process.

Start-Up

To ensure a valid initial start-up, the SCM implements the `l.aegis.enter` instruction for all processes. The instruction gets the same inputs as the `sys_aegis_enter()` system call. The SCM initiates the IV and ME mechanisms for the virtual memory space, computes the program hash, and sets up Verified/Private regions. Finally, the SCM sets the security mode bits. On architectures such as x86, the SCM also checks the initial stack pointer to avoid a stack overflow if an interrupt occurs.

Other Mode Transitions

All mode transition instructions `l.aegis.csm`, `l.aegis.suspend`, `l.aegis.resume`, as well as `l.aegis.exit` are provided to each secure process. The instructions perform the same operations as in the security kernel solution (see Section 5.1), but change the mode of individual process.

Protection Change

The `l.aegis.cpmr` instruction allows each secure process to change its Verified and Private protection regions in memory. Because each process' virtual memory space is separately

protected by the integrity verification and encryption, the SCM must re-initialize those mechanisms on this instruction (see Section 4.4.2).

Private Key Operations

Finally, the processor provides the private key operations for each secure process. Now the `l.aegis.pksign` and `l.aegis.pkdecrypt` instructions perform the private key operations for each secure process. The operations are almost identical to the ones for the security kernel except that the instructions include each process' program hash $AHash$ in place of the security kernel's program hash $SKHash$.

8.3.4 Security Discussion

This subsection discusses the new set of security attacks that malicious operating systems introduce and shows how such attacks are prevented in the proposed architecture.

Attacks on Program Integrity

Operating systems can either directly access on-chip components or manipulate the virtual-to-physical mapping to change the way a program accesses memory. The following discussion summarizes the new attacks possible with malicious operating systems.

Because operating systems manage initial start-up of a process, there are many potential attacks that the OS can carry out in addition to changing initial code and data in off-chip memory.

- *Program counter (PC)*: The operating system can start the program from an arbitrary point by setting an initial program counter. Because programs are written assuming a specific entry-point, tampering with the initial PC may result in unexpected changes in program behavior.

- *Stack pointer (SP)*: Programs often assume that a stack is located at the top of the virtual memory space and does not overlap with other memory regions such as a heap. The operating system is responsible for ensuring that a stack does not grow too large. Therefore, a malicious operating system can set a stack pointer to point to a memory region in use, and make a program overwrite its own data and instructions.

- *Program location*: Many executables are built assuming that they will be loaded into a specific address in the virtual memory space. The operating system may load such programs into a different location and change program behavior through position-dependent jumps.

Once the program starts, registers should be modified only by the owner process itself. Unfortunately, the operating system manages interrupts, and is responsible for properly saving and restoring the register values. Therefore, a malicious operating system can arbitrarily change *the register values on an interrupt*.

The operating system is also responsible for virtual memory management, which allows the operating system to tamper with *state in virtual memory space* in two different ways.

- *Direct accesses*: The operating system can access any physical memory location in *both on-chip caches and off-chip memory* by mapping it into its own memory space. Therefore, the operating system can replay, relocate, and substitute any value in a process's virtual memory space by actually modifying the values in the physical memory space.

- *Changing the mapping*: The operating system can also relocate and replay state in the virtual memory space without actually accessing the state. Simply changing the virtual-to-physical mapping will cause a process to access different physical locations for accesses to the same virtual address. Therefore, the process may overwrite a wrong location (relocation) and not update the intended location (replay). For on-chip caches, this attack is a concern only if the cache is physically-addressed.

It is important to distinguish the two types of attacks because attacks that change the virtual-to-physical mapping without directly accessing the physical location require more protection mechanisms to detect.

Finally, there are a few additional services that operating systems traditionally provide, which have new program integrity implications if the operating system cannot be trusted.

- *Fork*: If an operating system is allowed to fork a process at will and create a duplicate process with the same privileges, it essentially gives the operating system an ability to replay. For example, if a secure movie player can be duplicated during an execution, the operating system can fork a process right before it starts playing a movie and use

the second process to play the movie again. Therefore, a fork should be allowed only on an explicit request from a process itself.

- *Stack pointer*: As mentioned above, the operating system should ensure a stack does not grow and overlap with a heap. If an operating system is malicious, this function should be performed by the processor.

- *I/O and IPC*: If the operating system is malicious, no input from an I/O or an inter-process communication (IPC) can be trusted.

Integrity Protection

Now we briefly summarize how the AEGIS architecture protects program integrity against various attacks discussed in the previous subsection.

- *Initialization*: The main mechanism to ensure proper initialization is to compute the hash of a program when it enters a secure execution mode. The program hash is computed depending on the location of the `l.aegis.enter` instruction. Therefore, changing an executable or an entry-point will result in a different program hash.

  The stack pointer is checked when a program enters a secure execution mode to ensure that it is sufficiently far apart from a heap. We leave checking a program location to the program. Right after the instruction to enter a secure mode, the program should check if it is loaded at the intended location (see Sections 6.2 and 7.2.3).

- *Registers on an interrupt*: The processor saves and restores the registers on an interrupt. Therefore, the operating system cannot tamper with the register values.

- *On-chip caches*: On-chip cache blocks are tagged to prevent attacks from a malicious operating system. First, an ownership tag indicates which process owns the cache block, and detects attacks that directly modify the cache block. Second, if physically-addressed, the cache blocks are also tagged with virtual addresses, which prevent relocation attacks by changing a virtual-to-physical mapping. Finally, the processor flushes the relevant cache blocks when an TLB entry changes to ensure that there cannot be a replay attack using virtual-to-physical mapping.

- *Off-chip memory*: The integrity verification mechanisms prevent attacks on off-chip memory. Unlike the baseline architecture, however, the variant architecture applies

the scheme to the virtual memory space rather than the physical memory space. As a result, the scheme detects any attack on off-chip memory either by directly modifying it or by changing the virtual-to-physical mapping.

- *Non-volatile storage*: Because they are applied to the virtual memory space, the integrity verification mechanisms protect swapped pages in local non-volatile storage.

- *Other issues*: Currently, our design does not support forks if the operating system cannot be trusted. Also, applications should be written assuming that all I/O and IPCs are potentially malicious.

Program Privacy

In conventional computer systems, an operating system can easily read an user application's secrets in registers, on-chip caches, or off-chip memory. In the AEGIS architecture, the ownership tag prevents the secrets in on-chip caches from being read by other processes including an operating system. Registers are cleared by the processor after they are saved on an interrupt. Finally, memory encryption prevents secrets being read from off-chip memory.

8.4 L-Hash Integrity Verification

Section 4.3 described the MAC-based approach and the cached hash tree to protect the IV regions. These approaches check the integrity of memory after every memory access. However, checking the integrity after every access implies unnecessary overhead when we are only interested in the integrity of a *sequence* of memory operations. For example, in the distributed computation application, knowing exactly which operation has failed is not useful.

Here, we introduce an alternative approach of verifying memory integrity with low runtime overhead. While the MACs and the cached hash tree offer better general-purpose solutions, this new scheme significantly reduces the IV overheads for a set of applications where only a long sequence of operations needs to be checked. For a special-purpose secure processor, this scheme can replace the MACs and the cached hash tree.

In the following description of the algorithm, we consider a simple case where all pro-

tected chunks are stored in a single cache for ease of understanding. However, as discussed in Chapter 4, the integrity verification mechanism must check both data chunks and the ME time stamps. Therefore, in practice, the algorithm will be applied to both. Also, we again use a term *chunk* as the minimum memory block that is verified, which is identical to the L2 cache blocks.

The new approach is based on the work presented by Blum et. al [11] on memory correctness checking; we have extended it to be implemented with collision-resistant multiset[1] hash functions (which we will describe shortly), and to incorporate caches.

Intuitively, the processor maintains a read log and a write log of all of its operations to off-chip memory. At run-time, the processor updates logs with minimal overhead so that it can verify the integrity of a *sequence* of operations at a later time. To maintain the logs in a small fixed amount of trusted on-chip storage, the processor uses multiset hash functions. When the processor needs to check its operations, it performs a separate *integrity-check* operation using the trusted state.

A multiset hash maps multisets into a small fixed-sized bit string. It is incremental in that it is efficient to update it when a new element is added to the multiset. We use `MSet-XOR MAC` based on the hash function SHA-1. `MSet-XOR MAC` requires one SHA-1 operation using a secret key in the processor, and one XOR operation to update the multiset hash incrementally. `MSet-XOR MAC` is set-collision resistant in that it is hard to find a set and a multiset which produces the same hash; the formal proof of its set-collision resistance can be found in [20], and is not included here. For our purpose, since the multiset hashes are used to maintain logs, we refer to them as log-hashes, and refer to our scheme as the log-hash scheme.

Figure 8-5 shows the steps of the Log Hash (`LHash`) integrity checking scheme. To verify a sequence of memory operations, the processor (IV unit) keeps two log hashes (ReadHash and WriteHash) and a counter (Timer) in trusted on-chip storage. ReadHash maintains information of data read from memory, and WriteHash maintains information of data written to memory. Because our log hashes maintain set information, Timer is used to mark the order of memory operations. We denote the (ReadHash, WriteHash, Timer) tuple as the object $\mathcal{T}$.

[1] A multiset is an unordered group of elements where an element can occur as a member more than once [113].

Initialization Operation
add-chunks($\mathcal{T}$, set of `Address-Chunk` pairs):

1. Increment $\mathcal{T}$.TIMER. `TimeStamp` = $\mathcal{T}$.TIMER.
2. For each pair:
   (a) Store (`Chunk`, `TimeStamp`) at address, `Address`, in memory.
   (b) Update $\mathcal{T}$.WRITEHASH with the hash of (`Address`·`Chunk`·`TimeStamp`).

Run-Time Operations

- For a cache eviction
  write-chunk($\mathcal{T}$, `Address`, `Chunk`):
  1. Increment $\mathcal{T}$.TIMER. `TimeStamp` = $\mathcal{T}$.TIMER.
  2. Update $\mathcal{T}$.WRITEHASH with the hash of (`Address` · `Chunk` · `TimeStamp`).
  3. If a block is dirty, write `(Chunk, TimeStamp)` back to memory. If the block is clean, only write `TimeStamp` back to memory (we do not need to write `Chunk` back to memory).
- For a cache miss, do read-chunk($\mathcal{T}$, `Address`):
  1. Read the `(Chunk, TimeStamp)` pair from `Address` in memory.
  2. If `TimeStamp` > $\mathcal{T}$.TIMER, raise an integrity exception.
  3. Update $\mathcal{T}$.READHASH with the hash of (`Address` · `Chunk` · `TimeStamp`).

  and store `Chunk` in cache.

Integrity Check Operation
integrity-check($\mathcal{T}$):

1. `New`$\mathcal{T}$ = (0, 0, 0).
2. For each chunk address covered by $\mathcal{T}$, check if the chunk is in the cache. If it is not in the cache,
   (a) read-chunk($\mathcal{T}$, address).
   (b) add-chunks(`New`$\mathcal{T}$, address, chunk), where chunk is the chunk read from memory in Step 2a.
3. *//checks that, for each address, each read matches the most recent write*
   Compare READHASH and WRITEHASH. If different, raise an integrity exception.
4. If the check passes, $\mathcal{T}$ = `New`$\mathcal{T}$.

Figure 8-5: `LHash` Integrity Checking Algorithm.

Whenever there is a new set of chunks of memory that need to have their integrity verified, the processor performs an add-chunks operation to add it to WriteHash. This operation effectively remembers the initial value of the chunks in WriteHash. In our processor architecture, to protect the IV regions, the processor adds all chunks in those regions on the `l.aegis.enter` instruction using add-chunks.

At run-time, the processor calls read-chunk and write-chunk to properly update the logs. When a chunk gets evicted from the cache, the processor logs the evicted chunk's value by calling write-chunk. The chunk is associated with a new time stamp by incrementing Timer and using the new value. WriteHash is updated with the hash of the corresponding address-chunk-time stamp triple. If the chunk is dirty, the chunk and the time stamp are written back to memory; if the chunk is clean, only the time stamp is written back to memory. The scheme updates WriteHash even if the processor invalidates a cache block.

The processor calls read-chunk to bring a chunk from the memory. The time stamp associated with the chunk is checked to be less than or equal to the current value of Timer. Because the processor only maintains hashes, the time stamps are used to ensure that the chunk the processor reads from memory is the most recent chunk it stored to memory and that its memory accesses are not being reordered by an adversary (we refer to [20] for a detailed argument). ReadHash is updated with the hash of the address-chunk-timer triple.[2]

The WriteHash maintains information on the chunks that, according to the processor, should be in memory at any given point in time. The ReadHash maintains information on the chunks the processor reads from memory. During runtime, compared to ReadHash, WriteHash is updated once more per address. Therefore, to check the integrity of operations, all addresses covered by $\mathcal{T}$ are read and ReadHash gets updated accordingly. If ReadHash is equal to WriteHash, and assuming that all of the time stamp checks passed, then the memory was behaving correctly during the processor's sequence of operations. This checking is done in the integrity-check operation.

The processor performs an integrity-check operation when a program needs to check a sequence of operations, or when Timer is near its maximum value. Unless the check is at the end of a program execution, the processor will need to continue memory verification

[2]An application store instruction, which results in a cache eviction, incurs both a read and write to memory. Thus, ReadHash and WriteHash are properly updated even when there are only stores to a location. The same occurs when there are only loads to a location.

after an integrity-check operation. To do this, the processor initializes a new WriteHash while it reads memory during an integrity-check. If the integrity check passes, WriteHash is set to the new WriteHash, and ReadHash and Timer are reset. The program can then continue execution as before.

To avoid reading the entire virtual or physical memory space on a integrity-check operation, the processor can incrementally add chunks on demand and use a table to maintain the list of chunks ever touched. For example, the processor can use the program's page table to keep track of which pages it used during the program's execution. When there is a new page allocated, the processor calls add-chunks for all chunks in the page. When the processor performs an integrity-check operation, it walks through the page table in an incremental way and reads all chunks in a valid page.

In this scheme, the page table does not need to be trusted. If an adversary changes the page table so that the processor initializes the same chunk multiple times or skips some chunks during the check operation, the integrity check will fail in that ReadHash would not be equal to WriteHash.

In our description and implementation, we have used two log hashes, WriteHash and ReadHash. It is possible to implement the scheme using one log hash, RWHash. When a chunk is evicted from the cache, the hash of its corresponding triple is "added" to RWHash; when a chunk is brought into the cache, the hash of its corresponding triple is "subtracted" from RWHash. In essence, RWHash is the difference of WriteHash and ReadHash. If, at the end of the integrity-check operation, RWHash is equal to 0 (and assuming that all of the time checks have passed), the integrity check is successful.

Chapter 9

Application Scenarios

This chapter describes how the AEGIS secure processor can enable new security-critical applications. First, we briefly discuss how a secure processor can be introduced to remote parties so that the processor can be authenticated and trusted. All secure applications commonly require such key management protocols in order to trust remote processors. Then, we describe representative applications enabled by the AEGIS processor: certified execution, Digital Rights Management (DRM), and secure sensor networks.

9.1 Key Management

The AEGIS secure processor enables remote parties to trust the entire computing system only based on the trustworthiness of the processor even under physical attacks. To trust a processor, however, remote parties need to either know the public key of the processor or share a symmetric key with the processor so that the processor can be authenticated.

This section discusses how users can obtain the public key of a secure processor or share the symmetric key with the processor. Here, we consider two different scenarios. First, public-key infrastructure (PKI) is used to certify a processor's public key if a user interacts with an unknown computer on the Internet. Second, a user himself can directly bootstrap a secret if he has physical possession of the processors before deploying them in a potentially hostile environment.

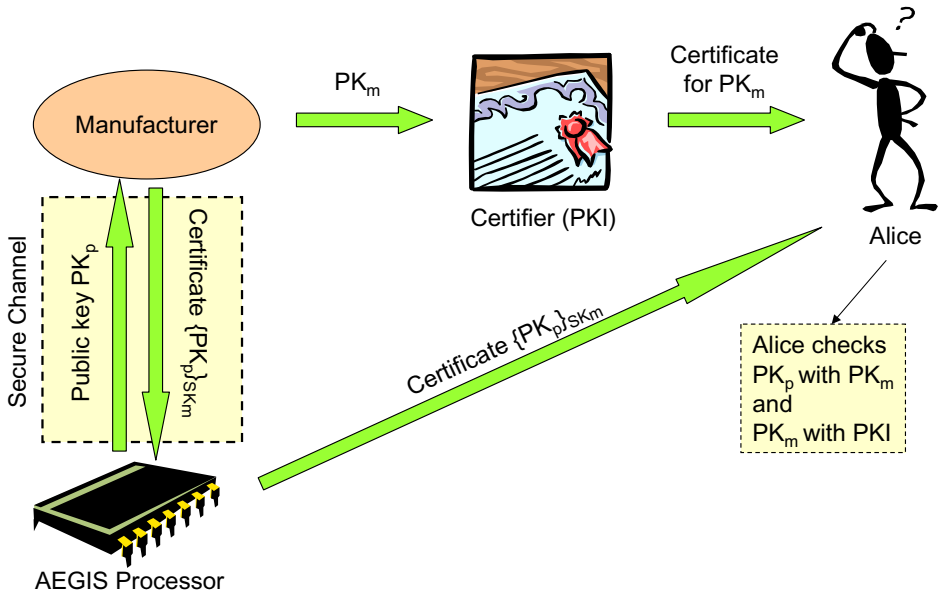


Figure 9-1: Remote introduction of the secure processor using PKI.

9.1.1 Remote Introduction

In common cases when remote users do not have physical access to the processor, a trusted third party must introduce the secure processor to the users remotely. Figure 9-1 illustrates one possible solution that the processor manufacturer acts as the trusted third party.

Before selling a processor, the manufacturer expresses a random private-public key pair (SK_p, PK_p) for the processor using the `l.puf.pksave` instruction, which returns the private key encrypted with a PUF response $E_R\{SK_p\}$, the syndrome S, and the public key PK_p. Then, the manufacturer creates a certificate by signing the processor's public key PK_p with his private key SK_m. Finally, the encrypted private key, the syndrome, and the certificate are provided to a system vendor so that they can be stored in non-volatile memory such as boot flash.

Now let us say the processor is deployed in the field and owned by Bob. Alice wants to interact with the processor but does not trust Bob. Therefore, Alice needs to know the processor's public key and authenticate the processor. For this purpose, the processor sends the certificate of its public key to Alice. Then, Alice can verify the certificate with the manufacturer's public key. Alice can also verify the manufacturer's public key using conventional public key infrastructure. Given the processor's public key, Alice can authenticate the processor, which can obtain its private key using the `l.puf.pkload` instruction.

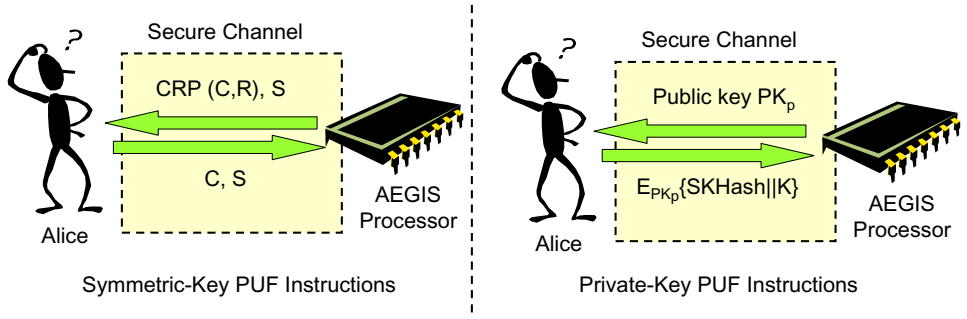


Figure 9-2: Direct bootstrapping of the secure processor.

9.1.2 Direct Bootstrapping

In some cases, secure processors may be used because systems operate physically exposed in hostile environments, not because the owners cannot be trusted. For example, in sensor networks, owners physically possess sensor devices, configure them, and deploy them in the field. Because the sensors are physically exposed, they need to be authenticated by the owner and other sensors configured by the same owner. In such cases, the owner can directly share a symmetric key with each processor without a trusted third party.

Figure 9-2 illustrates this approach to directly share a secret with the secure processor using the symmetric-key PUF instructions (see Section 8.2). Here, Alice has direct access to the secure processor before it gets deployed. For example, Alice can be the owner of the processor. Before deployment, Alice uses the `l.puf.response` instruction to obtain a unique challenge-response pair (CRP) (C, R) and the syndrome S over a secure channel. Alice keeps the response R as a secret to herself. The challenge C and the syndrome S are stored in the device, which gets deployed in a hostile environment. In the field, the security kernel can use the `l.puf.secret` instruction to obtain a secret key $K = H(SKHash \| \texttt{puf\_regenerate}(C, S))$. Alice can also compute the same secret $K = H(SKHash \| R)$ from the response. Now Alice shares a symmetric key with the kernel on a particular processor.

It is also possible to use the private key instructions to directly share a symmetric key with the processor. For example, Alice can express a private-public key pair using the `l.puf.pksave` instruction. Then, Alice chooses a random symmetric key K, and encrypts the key with the processor's public key for a particular kernel $E_{PK}\{SKHash \parallel K\}$. Finally, both the encrypted private key and the encrypted symmetric key are stored on the device

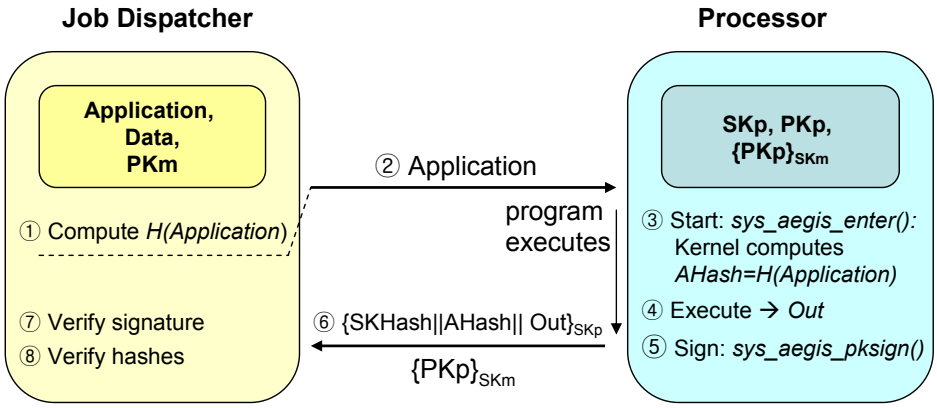


Figure 9-3: Certified Execution (Distributed Computation) by the AEGIS processor.

with the secure processor. Now only the particular security kernel on the processor can obtain that symmetric key.

9.2 Certified Execution

A typical example of certified execution is the distributed computation or grid computing example that we have used throughout this thesis. 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 [118].

Using a TE environment, 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 9-3 outlines a protocol that could be used by a job dispatcher to certify execution of an application program on a remote computer. Here, we assume that the manufacturer provided the certificate of the processor's public key in a way described in the previous section. First (1) the job dispatcher needs to know the hash of the application that it is sending out. For simplicity, we assume that the application includes all the necessary code

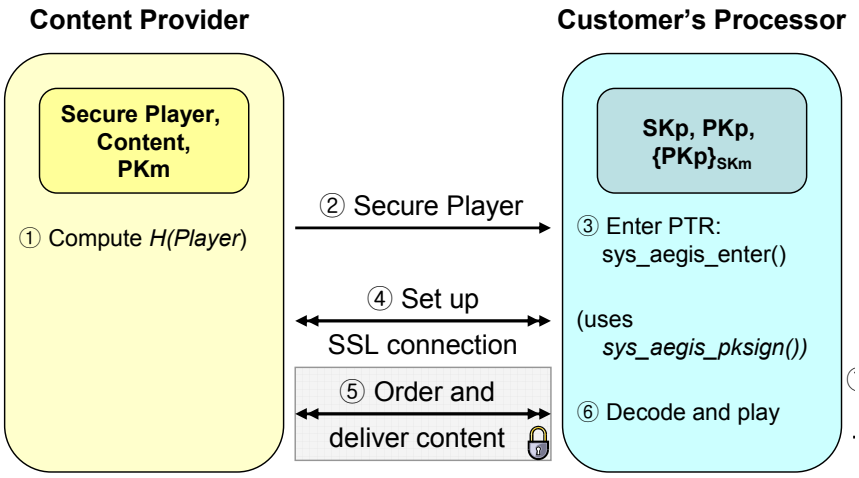


Figure 9-4: A simple Digital Rights Management (DRM) on the AEGIS processor.

and data for the run. The application is sent to the secure processor (2), which proceeds to run it. At this point, the processor is executing the security kernel, which is identified by its hash $SKHash$. The application enters TE mode by using the `sys_aegis_enter()` system call (3), at that time, a hash of the application gets computed for later use. The application executes and produces a result (4). The result gets concatenated with the program hashes ($SKHash$ and $AHash$) 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 hashes (8) before accepting the application's output as correct.

9.3 Digital Rights Management

Digital Rights Management (DRM) has become increasingly important since the advent of large scale sharing of copyrighted media over the Internet. 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.

In Figure 9-4, we show how a bidirectional private and authentic channel can be created between a content provider, and a trusted player, 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.

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 `sys_aegis_enter()` system call 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) [109], with client authentication, to establish a bidirectional private and authenticated channel with the content provider (4), the `sys_aegis_pksign()` system call being used to authenticate the client. In order to perform the SSL handshake, the player program requires a secure source of randomness, which is provided by the `l.aegis.random` instruction. 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).

Here, we note that our current architecture does not allow the player to implement the DRM policy that limits the number of times that a particular content can be played only using local storage. Our architecture cannot prevent replay attacks on off-chip non-volatile storage such as hard-disks.

9.4 Secure Sensor Networks

Sensor networks are being developed for a wide range of applications including environmental monitoring, intrusion alarms, and military surveillance. Thousands of nodes are distributed in a potentially hostile environment, collect sensor inputs, and communicate through wireless ad-hoc routing.

In sensor networks, the integrity of individual sensor inputs cannot be guaranteed because the environment around an individual node can always be manipulated by an adversary. Fortunately, there are often many nodes that are monitoring the same place. Thus,

unless the attacker can fool the majority of nodes at the same time, the corruption of a few inputs is not a concern. Also an ad-hoc network is quite robust from a few misbehaving nodes.

Attacks that compromise the entire network can occur when a single node sends many fake messages to the network. To counter this, work such as TinySec [58] proposes a shared key model where legitimate nodes attach a message authentication code (MAC) to every outgoing message. With the MAC, only messages sent by legitimate nodes will be accepted by other nodes or base stations. However, in order for this approach to work, the MAC key must be securely protected even under physical attacks.

PTR mode in our processor enables secure MAC computation on sensor nodes. As described in Section 9.1, the owner bootstraps unique symmetric keys for each sensor node before deployment. Using each node's key, the owner also embeds a common symmetric key shared by the entire network. Then, the nodes are deployed in the field. Now software in each node can enter PTR mode using the `l.aegis.enter` instruction, obtain the secret key, and compute a MAC. The protection mechanisms in the AEGIS processor guarantee that the key and the MAC computation is secure even when attackers can physically access the nodes.

Chapter 10

Implementation

We implemented the secure processor on an FPGA to validate and evaluate our design. All processor components including the processing core and protection modules are written in Verilog RTL. Our current implementation runs at 25 MHz on a Xilinx Virtex2 FPGA with 256-MB off-chip SDRAM (PC100 DIMM). We simply chose the relatively low frequency to have short hardware synthesis time; the current operating frequency does not reflect a hard limit in our implementation. This chapter discusses issues related to the embedded processor implementation of our design. We first give an overview of the implementation, and describe each component separately in more detail.

10.1 Implementation Overview

Figure 10-1 illustrates our secure processor implementation. The processor is based on the OR1200 core from the OpenRISC project [96]. OR1200 is a a simple 4-stage pipelined RISC processor where the EX and MEM stages of the traditional 5-stage MIPS processor are combined into one stage that can either take one cycle or two cycles. Even though our implementation is based on the 4-stage pipeline, the following discussion is based on more popular 5-stage pipelined processors.

Most security instructions such as `l.aegis.enter` perform rather complex tasks and are only used infrequently. As a result, in our implementation, all security instructions are implemented in firmware. The processor enters AT mode, called AEGIS Trap (AT) mode, on those software-implemented instructions, and executes the appropriate code in the on-chip code ROM. The processor also has an on-chip scratch pad that can only be accessed

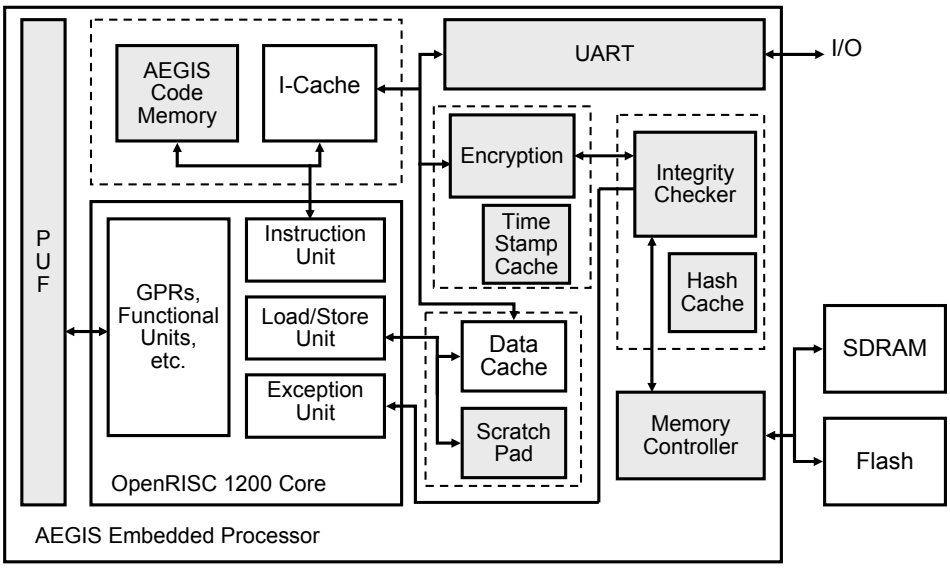


Figure 10-1: The overview of our processor implementation.

within AT mode, so that private computations can be done for the security instructions.

The PUF delay circuit is accessible through special-purpose registers so that the firmware can implement the PUF instructions. The PUF is only accessible in AT mode.

Finally, in addition to the processing core, the secure processor has hardware modules for off-chip integrity verification and encryption between on-chip caches and memory controller. The hash tree mechanism protects the read-write IV region, and a simple MAC scheme protects the read-only region. For encryption, the one-time-pad encryption scheme is implemented.

While our implementation is based on a simple processing core, we believe that it allows useful studies about secure processors in general. All additional hardware modules, except the ones for the special trap implementing the security instructions, are extensions independent to the processing core. Therefore, these protection modules can be combined with more complex processing cores in the same way. Moreover, embedded computers are likely to be one of the main applications for secure processors because they often operate in potentially hostile environments that require physical security. Thus, the implementation of embedded secure processors is in itself interesting.

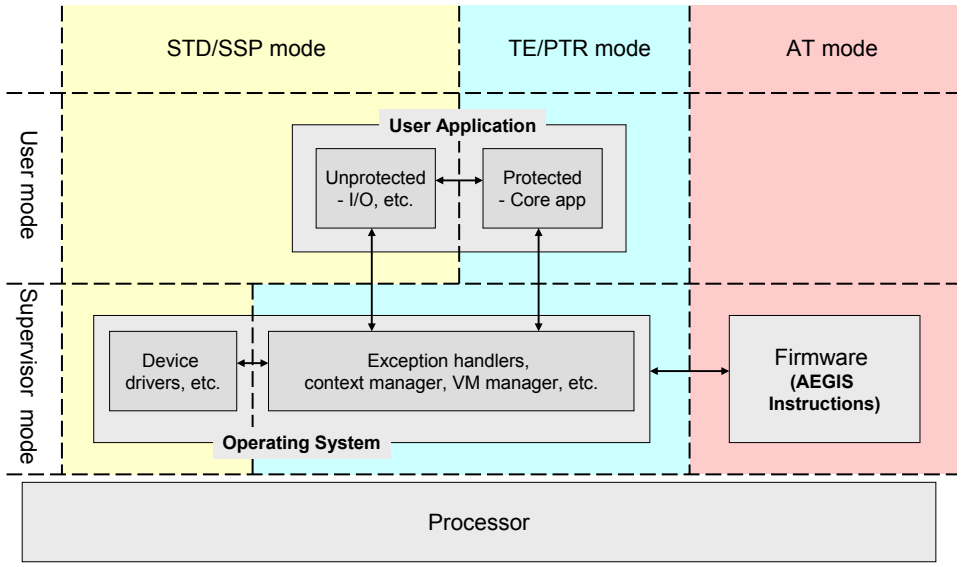


Figure 10-2: Execution modes in the AEGIS processor implementation.

10.2 Processor Core

This section describes the modifications required in the processor core to accommodate the additional security features in our secure processor as compared to conventional processors. The main change to the processor core is related to adding additional security instructions. Here, we explain the approach of implementing the instruction in a special trap mode. It is also possible to implement all instructions in hardware. However, the pure hardware approach is likely to take more hardware resources due to complex operations such as hashes and error correction.

10.2.1 AEGIS Trap Mode

In addition to the existing supervisor and user modes, the AEGIS processor design has four execution modes; STD, TE, PTR, and SSP. In the implementation, our processor has one additional mode called AEGIS Trap (AT) mode. The processor enters AT mode when it encounters one of the new AEGIS instructions that are implemented as firmware executing upon a trap. Figure 10-2 summarizes these five new execution modes in our processor implementation.

Because AT mode is designed to implement security instructions, it has higher privilege than any other execution mode including supervisor PTR mode. Firmware in AT mode can

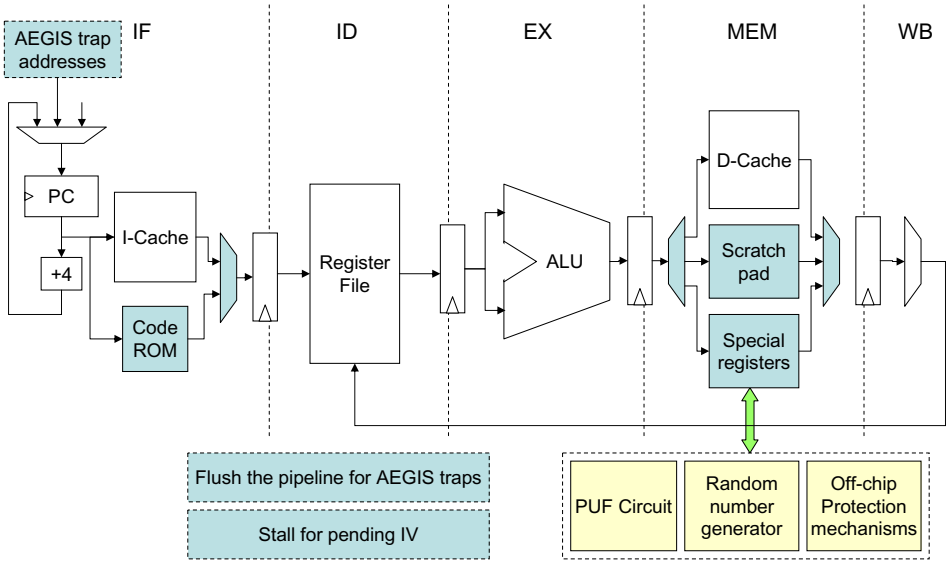


Figure 10-3: The modifications to the processor core for the AEGIS security features. The dark components indicate the new additions for the secure processor.

directly access the PUF delay circuit, control protection mechanisms such as encryption and integrity verification, and access special security registers such as the program hash ($SKHash$) register, private key register, etc.

The processor has a 1-bit register indicating whether it is in AT mode or not. This AT mode bit is cleared after a reset so that the processor starts in a regular supervisor mode. Also, the processor maintains two AEGIS control registers for the user-mode and the supervisor mode, which contain the mode bits (SM, UM) as well as the protection settings.

10.2.2 AEGIS Trap Support

Now let us consider the modifications required to support AT mode, in which most security instructions are implemented. Entering AT mode only needs minimal modifications to the core because it can be considered virtually identical to an exception for illegal instructions. The processor detects the security instructions that should be serviced in AT mode, waits till the MEM stage, and flushes all following pipeline stages using the existing exception mechanism. The difference is that the processor sets the AT mode bit, stores the instruction and the address of the next instruction to execute in special registers accessible in AT mode, and jumps to a pre-determined address in the code ROM as shown in Figure 10-3.

To execute the firmware, a code ROM and a scratch pad are added to the core. The code ROM contains the firmware implementing the security instructions. In AT mode, the processor fetches instructions from the code ROM rather than from the instruction cache. The scratch pad is a private space for the firmware to carry out its computations. For loads and stores in AT mode, the processor accesses either the scratch pad or the data cache based on the memory address. In our implementation, the scratch pad is mapped to the top of the memory space, which is used for I/O outside AT mode.

The firmware in AT mode also needs to control the off-chip memory protection mechanisms, access the PUF circuit, and set the secure execution modes. In our implementation, all these functions are mapped to special purpose registers. For example, the PUF circuit can be accessed by setting the PUF input register and reading the PUF output register. Also, changing the execution mode can be done by setting the AEGIS control register. To allow accesses to these special purpose registers in AT mode, the processor provides two instructions in AT mode, `l.aegis.mtspr` (Move To Special Purpose Register) and `l.aegis.mfspr` (Move From Special Purpose Register). As seen in Figure 10-3, the special purpose registers are mapped into their own memory space and accessed in the MEM stage.

To return from AT mode after completing a security instruction, the firmware uses a special instruction called `l.aegis.rfat` (Return From AEGIS Trap). On this instruction, the processor clears the AT mode bit and jumps back to the address following the security instruction that caused the AEGIS trap. This address is saved by the processor in a special register when the processor enters AT mode.

Finally, an exception may occur inside AT mode because some security instructions access memory. For example, `l.aegis.enter` computes the hash by reading the program in memory. On an exception in a regular mode, the processor saves the address of the instruction that caused the exception as well as the current execution state in special registers, enters the supervisor (PTR) mode, and jumps to a fixed exception handler location. In AT mode, the processor handles an exception in a slightly different fashion because it must appear to a handler in the security kernel as if the exception is caused by the security instruction that is being simulated in AT mode. The processor saves the address of the security instruction and the state before the AEGIS trap. On an exception in AT mode, the processor jumps to a firmware handler in AT mode, not directly to the handler in the security kernel. Then, the firmware properly aborts its operation and jumps to the handler

in the security kernel using `l.aegis.rfat`.

10.2.3 Pipeline Stalls

As discussed in Section 5.2.4, our processor design allows instructions to speculatively use values from off-chip memory in most cases before the integrity verification is completed. However, there are cases where the processor must stall and ensure that the integrity verification has been successful. When a security instruction is in the MEM stage or an exception other than the IV failure takes place, the processor must stall all instructions in and before the MEM stage. In PTR mode, a store instruction to the memory region outside the Private regions also causes a stall to ensure that the store does not write any private information into the memory (caches) until it is verified. These stalls can use the same datapath for conventional stalls for data hazards. To determine the stall condition, the processor core receives two signals from outside. First, `iv_pending` from the IV module indicates that there is an outstanding memory read that has not been verified. Second, `to_public` from MMU indicates that the memory access is to a public region.

10.2.4 Instructions Manipulating Off-Chip Protection

The security instructions that manipulate the off-chip protection mechanisms such as encryption and integrity verification require special attention. Namely, on such instructions, the processor must flush memory chunks, which are within regions that are affected by the changes in off-chip memory protection, from on-chip caches. For integrity verification, re-initialization requires flushing on-chip caches as described in Section 4.4.2. Flushing the caches is critical for the correctness and the security of off-chip encryption as well. First, without flushing, the processor may read a stale value from on-chip cache. For example, if the `l.aegis.setreg` instruction sets a decryption key for a read-only ME region, the subsequent instructions must be able to read decrypted values in the read-only ME region. However, if the processor accessed an encrypted value before (for example, due to the pipelining), on-chip caches may contain the stale value. Also, flushing is crucial for security. Say that a private value is modified in the on-chip cache, and later the address is excluded from the ME region. Unless the block is written back to memory before the change in the ME region, this private block will be written back in a plaintext form.

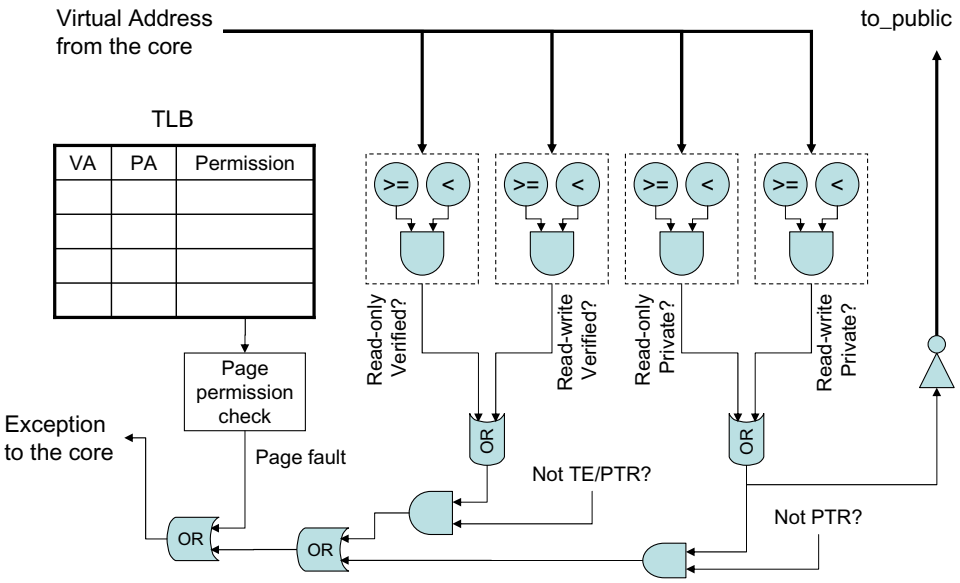


Figure 10-4: The memory management unit (MMU) with the security extensions.

10.3 Memory Management Unit

Given the processor core, let us consider the modification to the memory subsystem. Figure 10-4 illustrates the memory management unit (MMU) with additional access permission checks. The MMU has a conventional TLB, which translates a physical address to a virtual address and checks the page permissions. This TLB look-up takes one cycle in the MEM stage in the processor pipeline. As a result, the page fault will be detected by the end of the MEM stage.

In addition to the TLB look-up, the MMU checks if the virtual address from the processor core is within one of the four protection regions (read-only/read-write, Verified/Private). If so, the MMU verifies whether the memory access is permitted in the current execution mode. Also, the MMU generates the `to_public` signal back to the core if the access is *not* in a Private region. These operations mainly consist of simple bound checks that can be performed in parallel to the TLB look-up. Therefore, the results will be ready by the end of the MEM stage.

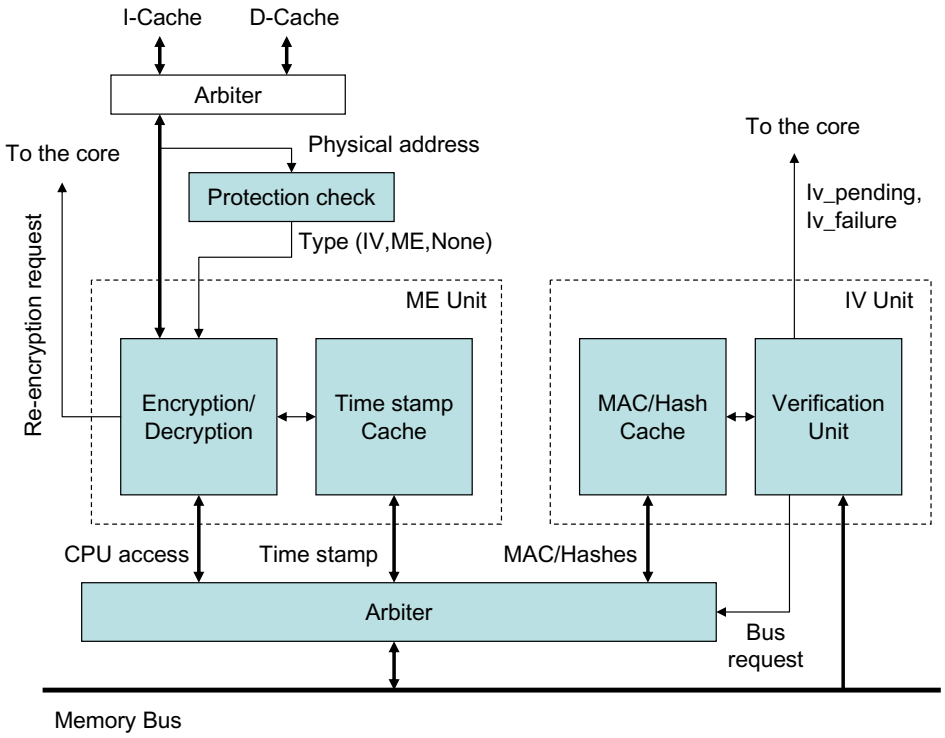


Figure 10-5: An overview of the off-chip protection modules.

10.4 Off-chip Protection Modules

This section describes the implementation of the off-chip memory protection mechanisms such as encryption and integrity verification whose algorithms are discussed in Chapter 4. Besides the new security instructions in the processor core and the permission check in MMU, the off-chip protection mechanisms are the only major components of our secure processor that remain to be discussed. We first describe how the two protection modules are integrated into the processor's memory hierarchy. Then, the internals of each module are discussed in more detail.

10.4.1 Overview

Figure 10-5 illustrates the overview of our off-chip protection modules in the memory hierarchy. In conventional processors without off-chip protection mechanisms, memory accesses from the instruction cache (I-Cache) and the data cache (D-Cache) normally pass through an arbiter and go directly to the memory bus connected to a memory controller. In the secure processor, the memory hierarchy up to the I/D-cache arbiter is identical to the con-

ventional system. However, there are four new additional components between the arbiter and the memory bus.

First, the processor needs to identify whether a memory access should be protected by IV or ME units. Therefore, the physical address of each access is checked to be within the four off-chip protection regions: read-only/read-write IV and read-only/read-write ME. This check can be performed in a way that is identical to the checks on the virtual addresses in MMU (see Figure 10-4). The result of this check is passed to the protection units. To signify this protection type, our bus carries additional bits as a type tag. Each access from the processor caches has four bits as a type tag, each of which indicates whether the access is within the read-only IV, read-write IV, read-only ME, and read-write ME, respectively. Accesses for ME time stamps and IV hashes are also marked with their own tags. MAC accesses from the IV unit are treated as unprotected.

Once the required protection is determined, all accesses from the instruction and data caches are handled by the Memory Encryption (ME) unit. The ME unit properly encrypts and decrypts the ME protected accesses. Unprotected accesses are simply passed to memory. In our implementation, the ME unit includes a separate cache for time stamps. Therefore, the ME unit produces memory accesses from the time stamp cache as well as the accesses from the instruction and data caches. In a high performance processor, an L2 cache can handle both processor data and ME time stamps. However, our embedded processor only has a relatively small L1 cache, which cannot be shared for time stamps.

The Integrity Verification (IV) unit monitors each access on the memory bus, and checks the integrity of each value if the corresponding tag indicates that it is IV protected. Because the verification process happens in the background, the IV unit simply monitors the values accessed, but does not interfere with memory accesses. For the processor core, the IV unit produces two signals, `iv_pending` and `iv_failure`, that indicate the status of the verification process. Similar to the ME unit, the IV unit possesses a separate cache for MACs and hashes. As a result, this MAC/hash cache can produce its own memory accesses.

Because three separate sources of memory accesses (I/D-caches, the time stamp cache, the MAC/hash cache) need to share the same memory bus, our implementation requires a second arbiter between the protection units and the memory bus. This arbiter allocates bus cycles based on a simple priority policy. In a normal case, the priority follows the order of the time stamp accesses, the CPU accesses, and the MAC/hash accesses. The time

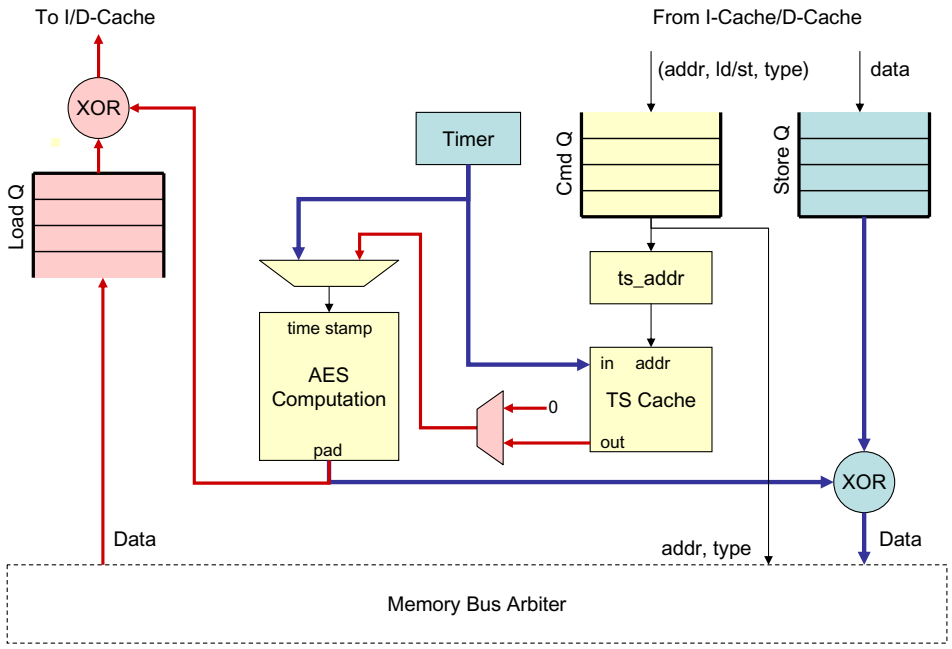


Figure 10-6: The Memory Encryption (ME) module. The blue represents the components used only for loads, and the red represents the ones for stores.

stamps are given the highest priority because they are required for the AES computation to decrypt a data block. The MACs and hashes have the lowest priority because the integrity verification does not delay the processor core in most cases. Obviously, with the given priority, MAC/hash accesses can be indefinitely delayed causing buffer overflows in the IV unit. To avoid this problem, the arbiter gets the `bus_request` signal from the IV unit. If this signal is asserted, the MAC/hash accesses get the highest priority.

10.4.2 Encryption Unit

Now let us consider the internals of each protection unit. Figure 10-6 illustrates the implementation of the ME unit in a simplified form. The memory accesses from the instruction and data caches are passed to the ME unit on the top of the right side. The ME unit buffers these accesses in the command queue and the store queue. For a read from the I/D-cache, the ME unit computes the address of the corresponding time stamp, reads the time stamp from the time stamp (TS) cache, and computes the decrypted pad using AES. At the same time, the ME unit issues a request for the data to the memory bus. Once the data arrives from memory, it is buffered in the load queue (on the left side of the figure) to wait for the

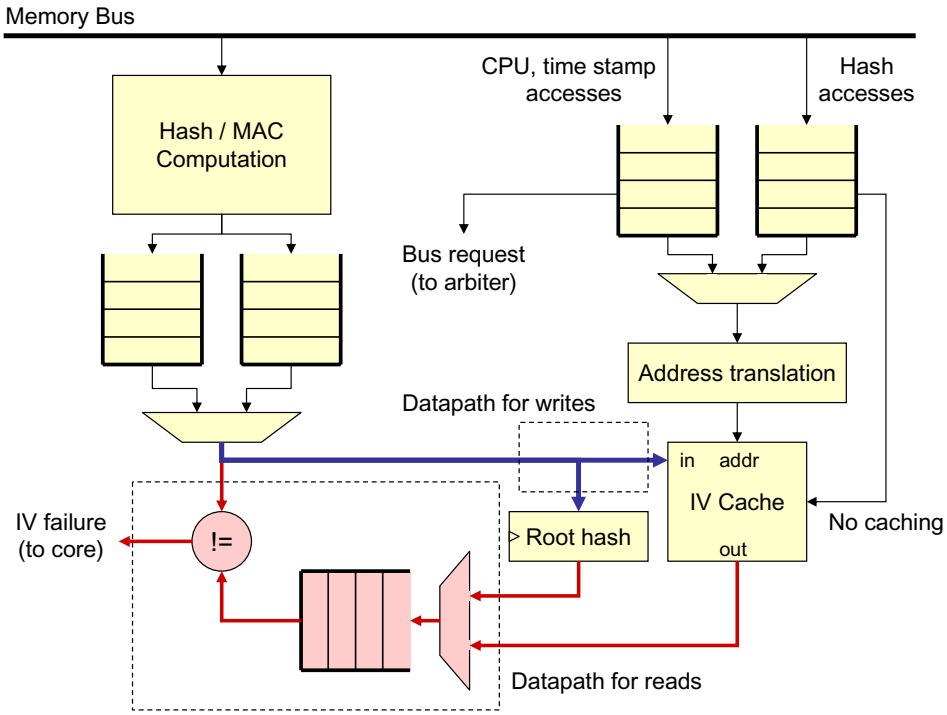


Figure 10-7: The Integrity Verification (IV) module. The blue represents the components used only for loads, and the red represents the ones for stores.

AES computation to complete. When the decryption pad is ready, the data is decrypted by XOR'ing with the pad, and returned to the processor cache.

For a write from the I/D-cache, the ME unit uses the time stamp from its timer register and computes the encryption pad. At the same time, the new time stamp is stored into the time stamp cache. Then, the timer register is incremented by one. When the encryption pad is ready from the AES unit, the data from the processor is encrypted with the pad and sent to memory.

While it is not shown in the figure, the ME unit must perform a re-encryption (see Section 4.2) when the timer gets close to its maximum value. This re-encryption can be implemented either in software or in hardware. In our implementation, the ME unit sends a signal to the processor core when the timer reaches the threshold, which incurs an AEGIS trap for the re-encryption.

10.4.3 Integrity Verification Unit

Finally, the IV unit is depicted in Figure 10-7. The basic operations of the IV unit are relatively simple. The unit monitors each memory access on the memory bus. If an access is tagged as IV protected (read-only IV, read-write IV, time stamps, or hashes), the IV unit computes the MAC or the hash of the data value. At the same time, the address and the tag are buffered, and the corresponding meta-data address is computed. For a read access, the IV unit reads the parent hash or the MAC either from the root hash register or the IV cache. Then, this parent hash or MAC is compared to the one computed from the value on the memory bus. For a write access, the IV unit waits until the new hash computation gets completed, and stores the new value into the IV cache.

The implementation of the cached hash tree has one tricky aspect. In the cached hash tree algorithm, each memory read to service a cache miss needs to be buffered so that it can be verified. Similarly, a cache write-back should be buffered so that the hash tree can be updated with a new value. Unfortunately, reading a parent hash to verify a read value or writing a new parent hash can incur a miss in the IV cache and a write-back of a dirty block. In this case, two new entries must be enqueued in order to dequeue one entry. As a result, if accesses to parent hashes are always cached, the buffer must be extremely large in the worst case (potentially as large as the cache itself).

To avoid this worst case scenario, our implementation selectively caches hashes read from memory. First, for an access of verified processor data or ME time stamps, the parent hash is always brought into the IV cache. If necessary, a dirty block is replaced and written back to memory. Similarly, for a hash write on the bus, the parent hash is always cached and updated in the cache (we use a write-allocate and write-back cache). On the other hand, when the parent hash is read, to verify a hash read on the memory bus, the IV cache replaces an existing block only if there is enough buffer space to enqueue a potential write-back. Otherwise, the parent hash is read from memory, but not put into the IV cache. (Figure 10-7 shows a simplified view of the real implementation.)

There is one final point of note regarding the MAC computation unit. To compute MACs, the ME unit utilizes the same hashing hardware that computes cryptographic hashes. A standard way to compute a MAC using a hash function is HMAC [65]. However, HMAC requires two hashing computations. For example, if one SHA-1 computation takes 80 cycles,

the HMAC computation takes 160 cycles, which results in significantly longer latency. To avoid this excessive latency for MAC computations, our implementation uses a simpler MAC construction based on NMAC [10]. In NMAC, the secret key is used as an initial vector of the hash compression function. Therefore, the length of the input string to a hash function is the same for both hash and MAC computations. Mihir Bellare, one of the developers of the NMAC and HMAC schemes, also suggested that only one hash compression round (80 cycles) is required if a MAC is computed for a fixed length input (personal communication, July 27, 2005). The only reason why HMAC and NMAC perform two hash computations is to prevent attacks exploiting variable length inputs. Therefore, for secure processors that protect fixed size cache blocks, a single hardware unit that performs one round of a hash compression function can be used for both MACs and hashes.

Chapter 11

Evaluation

This chapter evaluates various aspects of the secure processor design and implementation. First, Section 11.1 discusses the memory space overheads for the meta-data used by the off-chip protection mechanisms. Second, the additional hardware resource (silicon area) usage for the new security features is studied in Section 11.2. Finally, the impact of the protection mechanisms on processor performance is evaluated.

The memory overheads are independent of the implementation of the processor core, and can be determined simply from the design of the protection mechanisms. The additional hardware resource usage also mostly stems from the off-chip protection mechanisms, not the modification to the core. However, the performance of the secure processor heavily depends on the processor core implementation. Therefore, we show two separate analyses for embedded processors and high performance processors in Section 11.3 and Section 11.4, respectively.

11.1 Memory Space Overhead

The off-chip protection mechanisms require memory space for meta-data such as time stamps and hashes. Therefore, a part of the physical memory cannot be used by the processor core, effectively reducing the amount of off-chip memory in the system. This section summarizes the additional memory space required for the off-chip memory protection schemes based on the equations given in Section 4.4.1.

Protected region	Meta-data type	Overhead (meta-data/data)	Typical (%)
Read-only ME	None	-	-
Read-write ME	ME time stamps	mets_size / chunk_size	6.25
Read-only IV	MACs	mac_size / chunk_size	25
Read-write IV + ME time stamps	Hashes (`CHTree`)	1 / (chunk_size / hash_size -1)	33.3
IV with `LHash`	IV time stamps	ivts_size / chunk_size	6.25

Table 11.1: The summary of the memory space overhead for off-chip memory protection schemes. *chunk_size* represents the size of a cache block. The typical overheads are computed based on 64-B cache blocks, 32-bit time stamps, and 128-bit MACs/hashes.

11.1.1 Encryption

The one-time-pad (or counter-mode) encryption uses time stamps to ensure that each encryption pad is unique. Fortunately, for the read-only ME regions, the time stamps are always zero and do not need to be stored off-chip. For the read-write ME region, however, the time stamps must be stored in off-chip memory along with encrypted data and read back when decrypting the data. Therefore, as shown in Table 11.1, memory encryption consumes extra memory space to store time stamps, one for each chunk (or cache block) in the read-write ME region. For typical size of 32-bit (4-B) time stamps and 64-B cache blocks, the memory overhead for encryption is 6.25% of the read-write ME region.

11.1.2 Integrity Verification

Our baseline architecture uses MACs for the read-only IV regions and the cached hash tree (`CHTree`) for the read-write IV region to check the integrity of off-chip memory. The integrity checking schemes need memory space for MACs and hashes in addition to the data they verify. As summarized in Table 11.1, the additional memory space compared to data chunks is approximately $1/(m-1)$ for `CHTree` with a m-ary hash tree where m represents how many hashes fit into one chunk (or a cache block). For the MAC scheme, the overhead is a MAC per chunk (cache block). For typical values ($hash\_size = 16$ Bytes, $chunk\_size = 64$ Bytes, and $mac\_size = 16$ Bytes), the overheads are 33% of the read-write region and 25% of the read-only region.

In Section 8.4, we introduced an alternative way of verifying the off-chip memory integrity where a long sequence of operations need to be checked. This `LHash` scheme can

replace both MACs and the hash trees, and require time stamps to be stored along with the verified chunks in memory. Here, note that these time stamps are different from ME time stamps and require separate memory space. Therefore, the memory space overhead of `LHash` is a time stamp per verified chunk, which is 6.25% for the 32-bit time stamps and 64-B chunks. Therefore, `LHash` requires significantly less memory space compared to `CHTree`.

11.1.3 Summary

Overall, the off-chip protection mechanisms consumes 25% and 33% additional memory space for the MAC and the hash tree integrity verification, respectively. The memory encryption scheme adds 6.25% to the 33% overhead of the read-write regions. Therefore, to protect the privacy and the integrity of the entire read-write memory, only around 60% of the memory can be used for real processor data.

There are a few ways to reduce this memory overhead. The simplest solution is to increase the chunk size (or the cache block size). The above numbers are computed assuming 64-B cache blocks. If the blocks are 128 Bytes, the overheads reduce roughly by half. Also, the `LHash` scheme has significantly less memory overheads as it only require a time stamp per chunk. In fact, if chunks are encrypted with a randomized encryption scheme, even the time stamps are unnecessary for the `LHash` scheme. Therefore, when only a long sequence of operations need to be verified, the memory space overheads can be dramatically reduced (to 6.25%). Finally, the 128-bit MACs are fairly conservative given our application. Because attackers must forge a MAC during a processor's execution time, and cannot perform a birthday attack on the MAC, smaller MACs (64-bits) will be sufficient for our system, which can reduce the MAC overhead by half.

11.2 Hardware Resource Usage

This section studies the additional hardware resource usage (silicon area) for the security features in our processor. Our processor architecture requires additional instructions, access checks in MMU, and off-chip memory protection mechanisms as described in Chapter 10. Because the modifications to the processor core and the MMU are relatively simple, the off-chip protection schemes consume the majority of the additional hardware resources. Here,

Parameters	Specification
Processor	OR1200 core, 25 MHz
Off-chip SDRAM	64MB, 64bits/cycle, 12.5MHz
I/D cache	32KB, direct-mapped, 64B line
ME unit	4KB, direct-mapped, 64B line cache 3 AES blocks (3 * 128bits/12 cycles)
IV unit	16KB, direct-mapped, 64B line cache 5 SHA-1 blocks (5 * 512bits/80 cycles)

Table 11.2: The default processor parameters.

we first evaluate the silicon area usage of our embedded processor implementation. Then, we briefly discuss the area usage for high performance processors based on the analysis of cryptographic primitives and the overheads in the embedded processor case.

11.2.1 Processor Parameters

As described in Chapter 10, the AEGIS processor implementation is based on the OR1200 core from the OpenRISC project [96], which is a simple 4-stage embedded processor. All processor components including the processing core and protection modules are written in Verilog RTL. The current implementation runs on a Xilinx Virtex2 FPGA with 256-MB off-chip SDRAM (PC100 DIMM).

In the evaluation, we use the parameters in Table 11.2 as the default for our processor on an FPGA. The clock frequency of the off-chip DIMM (PC100) is chosen to be one half of the processor's clock frequency. This gives us a realistic memory latency in clock cycles as we would see with actual ASIC processors that have a much higher clock frequency. For example, we expect an ASIC implementation to run at a few hundreds of mega-hertz where the off-chip SDRAM operates at 100-133MHz. Therefore, the processor core will be two to three times faster than the off-chip DRAM.

The parameters of IV and ME units are selected to match the off-chip bandwidth. In our implementation, the processor can read or write 32 bits per core cycle, or 64 bits per off-chip bus cycle (2 processor cycles). For the ME unit, one 128-bit AES computation takes 12 cycles in our implementation. Therefore, in order to process 32 bits per cycle, three copies of the AES unit are required. Similarly, for the IV unit, one SHA-1 computation with a 512-bit input takes 80 cycles. Thus, we need five copies of SHA-1.

Instruction	Code size (B)	Memory req. (B)
`l.aegis.enter` (1)	576	256
`l.aegis.exit`	24	0
`l.aegis.csm`	28	0
`l.aegis.cpmr`	212	4
`l.aegis.suspend`	36	0
`l.aegis.resume`	72	0
`l.puf.response` (1,2,4)	596	1,236
`l.puf.secret` (1,3,4)	600	1,240
(1) *SHA1 hash*	1,960	120
(2) *bch_encode*	468	20
(3) *bch_decode*	5,088	1,100
(4) *get_puf_resp*	880	88
common handler	540	0
All	11,080	1,240

Table 11.3: The memory requirements of security instructions.

11.2.2 Security Instructions

The new security instructions incur both space and performance overheads. Because they are implemented as firmware executing in AT trap mode, the instructions require an on-chip code ROM and data scratch pad, consuming more transistors and on-chip space. Also, some of the instructions involve rather complex operations that require many cycles. In this subsection, we focus on the additional space usage. The performance of the security instructions is discussed in the following section.

Our embedded processor implements the symmetric-key PUF instructions in Section 8.2 rather than the private-key instructions (`l.puf.pksave`, `l.puf.pkload`) described in our baseline architecture. We believe that the symmetric-key instructions are more appropriate to embedded processors as they are likely to be bootstrapped by the owner without a trusted third party (see Section 9.1). Also, the overheads of the private-key PUF instructions are almost identical to the overheads of the symmetric-key instructions because the majority of the overhead comes from the BCH error correction (`puf_calibrate` and `puf_regenerate`).

Table 11.3 summarizes the memory requirements of our security instructions. For each instruction, the table shows the size of the code in the code ROM, and the size of the on-chip scratch pad (SRAM) required to execute the code. The space overhead of our instructions is quite small. The entire code ROM is only 11,080 Bytes, and the scratch pad for data needs to be only 1,240 Bytes. The instructions listed in the upper half of the table also share

the lower four routines labeled (1)-(4) during execution, thereby reducing code size. The `l.puf.secret` instruction uses the most resources, as it requires BCH decoding. However, note that the memory requirement of BCH decoding depends on the length of the codeword, and can be reduced by using a smaller codeword. While our current implementation uses a 255-bit codeword, it is always possible to trade the space requirement with the number of secret bits we obtain; we can use a set of smaller codewords and greatly reduce the scratch pad size and execution time. In the following subsection, we see how much silicon area that the 11-KB code ROM and the 2-KB ($> 1,240$ Bytes) scratch pad consume.

11.2.3 Silicon Area Usage

Our processor requires additional hardware modules for the security features. In this subsection, we compare the additional hardware resources used in our secure processor implementation with the hardware usage of the baseline processor that does not have any security features.

To analyze these overheads we performed an ASIC synthesis of both the baseline OpenRISC CPU and the AEGIS secure processor. Using TSMC $0.18\mu m$ libraries, we compare the size of major components in Table 11.4. The gate count is an approximation based on an average NAND2 size of $9.97\mu m^2$, and ROMs are implemented as combinational logic.

As seen in the table, the processor with all the protection mechanisms is roughly 1.8 times the size of the baseline. However, much of this overhead is due to the extreme simplicity of the OpenRISC core, which only uses $51,408$ gates. (The SDRAM controller and UART make up 43% of this.) Our logic gate count is $313,569$ versus the baseline $81,945$. Thus, the logic overhead is around $231,624$ gates ($2.3mm^2$), which is fairly reasonable for modern microprocessors. For example, the core size of the embedded PowerPC 440 ($0.18\mu m$) is reported to be about $4mm^2$. If we consider high performance CPUs, die size of the PowerPC 750CXr (G3) ($0.18\mu m$) is $42.7mm^2$, and the die size of the Pentium 4 ($0.18\mu m$) is $217mm^2$.

As expected, the majority of the area is consumed by the off-chip memory protection units. For example, more than 80% of the logic overhead and the total area overhead come from the IV and ME modules. The additional logic area for both processor core and the PUF circuit is only $0.25mm^2$, which is almost negligible compared to $1.9mm^2$ consumed by the IV and ME modules. Within the off-chip protection modules, the cryptographic

Module	Resource Usage (AEGIS / Base)		
	Logic μm^2	Gate Count	Mem. μm^2 (Kbit)
CPU Core (tot)	598,525 / 512,541	60,032 / 51,408	0 / 0
- SPRs	61,538 / 13,728	6,166 / 1,376	0 / 0
Code ROM	138,168 / 0	13,845 / 0	0 / 0
Scratch Pad	2,472 / 0	248 / 0	260,848 (16) / 0
Access Checks	115,735 / 0	11,597 / 0	0 / 0
IV Unit (tot)	1,075,320 / 0	107,756 / 0	1,050,708(134) / 0
- 5 SHA-1 units	552,995 / 0	55,415 / 0	0 / 0
- Hash cache	17,076 / 0	1,711 / 0	1,050,708(134) / 0
ME Unit (tot)	864,747 / 0	86,655 / 0	503,964 (34) / 0
- Key SPRs	108,207 / 0	10,843 / 0	0 / 0
- 3 AES units	712,782 / 0	71,426 / 0	0 / 0
- TS cache	14,994 / 0	1,502 / 0	503,964 (34) / 0
PUF	26,864 / 0	2,691 / 0	0 / 0
I-Cache*	18,932	1,897	1,796,458 (272)
D-Cache*	27,321	2,737	2,484,464 (274)
Debug Unit*	36,118	3,619	0
UART*	73,663	7,381	0
SDRAM Ctrl*	148,423	14,873	0
AEGIS Totals	3,126,288	313,569	6,096,442 (730)
Base Totals	816,998	81,945	4,280,922 (546)
Full Chip Area	9,222,730 μm^2 vs. 5,097,920 μm^2 (1.8x larger)		

Table 11.4: The hardware resource usage of our secure processor. * Denotes that values are identical on both AEGIS / Base

primitives such as a block cipher (AES) and a hash function (SHA-1) consume the most amount of logic. The SHA-1 units occupy more than 50% of the IV module's logic area, and the AES units utilize more than 80% of the ME module's logic area. Therefore, it is possible to reduce the area of the IV and ME modules by either reducing the number of AES and SHA-1 units used (giving lower performance), or by replacing them with "weaker" algorithms such as MD5 [110] or RC5 [112] which consume less area.

11.2.4 Discussion on High Performance Processors

The AEGIS processor implementation provides an estimate of the area overheads for embedded processors. Now let us consider how this area usage may change for more complex high performance processors. In short, our protection mechanisms are almost independent of the processor core, and the area usage of our embedded processor implementation should provide a fairly good estimate for high performance processors as well. We expect our protection mechanisms to only require a few hundred thousand gates even for high performance processors.

The main area overhead for the processor core comes from the additional code ROM and the scratch pad, whose sizes are independent of the processor core. While the processor is also required to support a special trap mode (AT mode) and additional stalls to wait for pending integrity verifications, these modifications are minimal because they can use pre-existing exception mechanisms or stall mechanisms.

Similarly, there is no reason for the access checks in MMU to be different for high performance processors. If the processor can perform multiple memory accesses simultaneously, the access check hardware must be duplicated. However, this duplication can only change the area consumption by a small factor. Given that the access checks only require 11K gates in our embedded processor implementation, the overhead in MMU will still be negligible even if the access check unit is replicated a few times.

Finally, we also expect the area usage of our off-chip protection mechanisms to stay at the same order of magnitude for high performance processors. However, there are a few points that need to be mentioned. First, in a high-end processor with a large on-chip L2 cache, the off-chip protection modules can share that cache for time stamps, MACs, and hashes instead of having separate caches. As a result, the overheads of the IV and ME caches ($1.5mm^2$ in our implementation) can be eliminated in high-end processors. While the datapaths of the IV and ME modules may have to slightly change for high performance processors, this change should not significantly change the area consumption as these modules are already designed for relatively high performance matching the maximum off-chip bandwidth. On the other hand, the AES and SHA-1 units may become slightly larger to operate with a higher clock frequency. We briefly discuss the implementation of AES and SHA-1 below.

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 [90]. 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 multiple rounds of four transformations: `SubByte`, `ShiftRows`, `MixColumns` and `AddRoundKey` along with key expansion. The number of rounds required for each execution is 10, 12, and 16 for the key size of 128, 192, and 256 bits respectively.

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 [66, 119]. 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. The key expansion of this ASIC implementation is identical to two rounds of the AES key expansion because the ASIC implementation supports 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.

In one ASIC implementation [119], a 128-bit block AES encryption optimized for high performance costs approximately 75,000 gates. This gate count is a few times more than the estimate of 21,000 gates in our implementation. However, this example demonstrates that high performance AES engines can be built with reasonable gate counts.

Cryptographic Hash Functions

To evaluate the cost of the hash units, we consider the MD5 [110] and SHA-1 [32] hashing algorithms. The algorithms take a 512-bit block, and produce a 128-bit or 160-bit (for SHA-1) digest. In each case, simple 32-bit operations such as additions and shifts are performed over 80 rounds. In our implementation, one round of the SHA-1 algorithm along with appropriate registers consumes about 10,000 gates. Because the operations in each round are very simple, it should be easy to perform each round with a very high clock frequency. Therefore, the size of the hash unit should not change much even for high performance implementations.

11.3 Performance I: Embedded Processors

This section evaluates the performance of the AEGIS implementation. The performance overheads of our secure processor come from two sources. First, the new security instructions take multiple cycles to execute because they are implemented in firmware. Especially,

Instruction	Execution cycles
`l.aegis.enter` (1)	22,937+2m+n(1)
`l.aegis.exit`	25
`l.aegis.csm`	18
`l.aegis.cpmr`	196+2m
`l.aegis.suspend`	43
`l.aegis.resume`	48
`l.puf.response` (1,2,4)	48,299+2(1)+(2)+(4)
`l.puf.secret` (1,3,4)	57,903+2(1)+(3)+(4)
(1) $SHA1\_hash$	4,715
(2) bch_encode	161,240
(3) bch_decode	2,294,484
(4) get_puf_resp	932,159
common handler	46 − 92

Table 11.5: The number of cycles to execute security instructions.

complex instructions such as `l.puf.secret` that include BCH decoding can take many cycles. Second, the two off-chip memory protection mechanisms, namely integrity verification and encryption, can degrade the off-chip memory performance. While we have modified the processor core and the MMU for access checks, those changes are carefully designed not to affect the clock frequency. We first discuss the performance of the new security instructions, and the run-time performance with the off-chip protection mechanisms. All experiments in this section are based on the RTL implementation and executions on an FPGA board.

11.3.1 Security Instructions

Table 11.5 shows the number of cycles that each security instruction takes to execute. Here, the execution cycle does not include the overhead of flushing the pipeline for a trap, which is only a few cycles per instruction. In the table, we give the execution time for each instruction in terms of the instruction's base cycle count and the number of times it uses a subroutines (where n is the number of 64 Byte chunks of memory which make up the program hash, and m is the number of 64 Byte chunks that are to be protected by integrity verification).

The results show that the performance of our instructions is not a concern. All instructions that would be used frequently take a small number of cycles. While some instructions such as `l.aegis.enter` and `l.puf.secret` can take a large number of cycles, they will be used infrequently in applications. The `l.aegis.enter` instruction should appear only once

at the start of an execution, and secret generation will likely occur only a handful of times throughout the entire execution of an application. Thus, over a long execution period, the overhead of these slow instructions should be negligible.

The performance of the slow instructions can also be improved, if desired. First, the execution time of $SHA1\_hash$ could be reduced greatly from 4,715 cycles to 80 cycles if we use a hardware SHA-1 unit. Similarly the get_puf_resp function could reduce its execution cycle time to $5,665$ if we added a hardware linear feedback shift register which could be used instead of a software pseudo-random number generator. Finally, the cycle time of the BCH decoding heavily depends on the length of the codeword. Therefore, we can trade performance for the number of secret bits we obtain.

11.3.2 Off-Chip Protection

Integrity verification and encryption affect the processor performance in two ways. First, they share the same memory bus with the processor core to store meta-data such as hashes and time stamps. As a result, these mechanisms consume off-chip memory bandwidth. Second, the encrypted data cannot be used by the processor until it is decrypted. Therefore, the encryption effectively increases the memory latency.

Table 11.6 shows the performance of the AEGIS processor under TE and PTR modes when run on an FPGA. In TE mode only integrity verification is enabled. In PTR mode both IV and ME are enabled. PTR mode experiments encrypt both the read-write data and read-only program instructions, however, we found that nearly all of the slowdown comes from the encryption of read-write data. These benchmarks do not use SSP mode, and therefore show worst-case performance overheads, where an entire application is protected.

The `vsum` program is a simple loop which accesses memory at varying strides to create different data cache miss rates. Since the processor suffers much of its performance hit during cache misses, this benchmark attempts to demonstrate worst-case slowdown. Table 11.6 shows that the performance degradation not prohibitive for programs with reasonable cache miss rates. For realistic cache miss rates of less than 10%, the slowdown is less than 20% for TE mode, and 25% for PTR.

One other major factor which affects performance overhead is the size of the protected read-write integrity verification region. Table 11.6 uses 4MB as a typical protected region size, but also breaks down the effects of smaller and larger protected regions using the `vsum`

	STD cycles	TE slowdown	PTR slowdown
Synthetic "vsum" (4MB Read-write IV Region, 32KB IC/DC, 16KB HC)			
- 6.25% DC miss rate	8,598,012	3.8%	8.3%
- 12.5% DC miss rate	6,168,336	18.9%	25.6%
- 1MB Dyn. IV Rgn.	1,511,148	18.8%	25.3%
- 16MB Dyn. IV Region	25,174,624	19.2%	25.9%
- 25% DC miss rate	4,978,016	31.5%	40.5%
- 50% DC miss rate	2,489,112	62.1%	80.3%
- 100% DC miss rate	1,244,704	130.0%	162.0%
EEMBC (4MB Read-write IV Region, 32KB IC/DC, 16KB HC)			
routelookup	397,922	0.0%	0.3%
ospf	139,683	0.2%	3.3%
autocor	286,681	0.1%	1.6%
conven	138,690	0.1%	1.3%
fbital	587,386	0.0%	0.1%
EEMBC (4MB Read-write IV Region, 4KB IC/DC, 2KB HC)			
routelookup	463,723	1.4%	21.6%
ospf	183,172	26.7%	73.1%
autocor	288,313	0.2%	0.3%
conven	166,355	0.1%	5.2%
fbital	820,446	0.0%	2.9%

Table 11.6: Performance overhead of TE and PTR execution.

example with a 12.5% miss rate. Increasing and decreasing the size of the protected region has only moderate effect on the TE and PTR overheads. This is because the hash cache hit-rate is consistently poor for the `vsum` benchmark. For the `vsum` benchmark specifically, the overhead only noticeably reduces once the entire hash tree fits in the hash cache, resulting in almost zero overhead.

For a more realistic evaluation of the protection mechanism overheads, we ran a selection of EEMBC kernels from the embedded microprocessor benchmark consortium [33]. Each EEMBC kernel was run using its largest possible data set, with two different cache configurations. We also chose to run each kernel for only a single iteration to show the highest potential slowdown. Hundreds and thousands of iterations lower these overheads to negligible amounts because of caching. Using an instruction and data cache size of 32KB (IC/DC) and hash cache of 16KB (HC), Table 11.6 shows that our protection mechanisms cause very little slowdown on the EEMBC kernels. Reducing the cache to sizes which are similar to embedded systems causes a greater performance degradation, however, most kernels still maintain a tolerable execution time. Overall, the performance overhead of our secure processor is reasonable for embedded applications with realistic cache miss-rates.

11.3.3 Suspended Secure Processing

The AEGIS processor allows only a part of the application to be trusted and protected using SSP mode. To illustrate the usefulness of this feature, we investigate how SSP mode can be used in a simple sensor network.

In sensor networks thousands of nodes are distributed in a potentially hostile environment, collect sensor inputs, and communicate through wireless ad-hoc routing. Attacks that compromise the entire network can occur when a single node sends many fake messages to the network. To counter this, work such as TinySec [58] proposes a shared key model where legitimate nodes attach a message authentication code (MAC) to every outgoing message. As with the distributed computation example, provided that the messages are signed before communication, only the MAC computation must be run in a secure execution mode. We note, however, that the integrity of sensor node *inputs* can only be guaranteed by numerous nodes monitoring the same location.

We analyzed simple sensor network software that collects a sensor input, computes a MAC, and sends the message to the base station. This application is constructed based on the `Surge` application in the `TinyOS` package [70]. The application consists of three main parts: sensor input processing, ad-hoc networking, and MAC computation. As discussed above, only MAC computation needs to be protected under PTR mode.

We measured the size of the code corresponding to each function under the OpenRISC platform. At the current time, we do not have a sensor network deployed whose nodes are based on the AEGIS secure processor. Therefore, the only way of obtaining realistic numbers corresponding to the amount of time the sensor node spends executing each function was to run a simulation using the TOSSIM simulator, which is a functional sensor network simulator.

We found that of the $52,588$ Bytes of instruction in this application, HMAC only consisted of $2,460$ Bytes. Therefore only 4.7% of the program instructions need to be protected by integrity verification. During execution, HMAC consumed an average of 32.5% of the cycles in the main program loop. Reducing the security overhead of an application by one third can certainly have a dramatic effect on performance.

11.4 Performance II: High-End Processors

This section further evaluates the performance of our secure processor through simulations. Because the security instructions are used infrequently in applications, this section focuses on the performance overheads incurred by the off-chip memory protection mechanisms. Evaluations in this section are different from the emulation study in the previous section in two ways. First, the simulation results demonstrate the performance of high performance secure processors rather than embedded processors. Second, through simulations, we evaluate various off-chip protection mechanisms in more detail, not just the one set of mechanisms implemented. For integrity verification, we compare `CHTree` and `LHash` (see Section 4.3 and Section 8.4). We also show the advantage of OTP encryption compared to the conventional direct encryption scheme (see Section 4.2).

11.4.1 Simulation Framework

Our simulation framework is based on the SimpleScalar tool set [15]. The simulator models speculative out-of-order processors with separate address and data buses. All structures that access the main memory including an L2 cache, the integrity checking unit, and the encryption unit share the same bus. The architectural parameters used in the simulations are shown in Table 11.7. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance. We used small buffers for time stamps (one for ME time stamps and another for `LHash` time stamps) to exploit spatial locality because time stamps are only 4 B while the memory bus is 8-B wide.

For all the experiments in this section, nine SPEC2000 CPU benchmarks [47] are used as representative applications. To capture the characteristics in the middle of computation, each benchmark is simulated for 100 million instructions after skipping the first 1.5 billion instructions.

Figure 11-1 shows the baseline performance of these benchmarks. In the graph, IPC stands for instruction per cycle, and is a standard performance metric for microprocessors. Benchmarks `mcf`, `applu`, and `swim` show poor L2 cache performance, and heavily utilize the off-chip memory bandwidth (`bandwidth-sensitive`). The other benchmarks are sensitive to cache sizes, and do not require high off-chip bandwidth (`cache-sensitive`).

We use the term "baseline" to refer to a standard processor without integrity verifi-

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
TLB latency	160
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 length	128 bits
Time stamps	32 bits
Time stamp buffer	32 8-B entries

Table 11.7: Architectural parameters.

cation or encryption. In the following discussions, we evaluate the overhead of security mechanisms compared to the baseline system with the same configurations. We note that these overheads are *optimistic* because the baseline could possibly be improved in performance using the larger hardware budget of the secure processor. Unfortunately, the current simulation framework does not allow us to simulate slightly larger caches that incorporate a few hundred thousand bits of extra space in lieu of the additional gates required for integrity verification and encryption. However, Figure 11-1 suggests that our estimation is not overly optimistic. Even if we consider the baseline with L2 caches that are twice as large, it only improves the baseline performance by 22% on the average. Therefore, we can consider the pessimistic overhead of the schemes to be at most 22% larger than the optimistic overhead.

11.4.2 Integrity Verification: TE Processing

This subsection evaluates the two integrity verification schemes proposed in this thesis: the hash tree scheme `CHTree` and the log hash integrity checking scheme `LHash`. In the experiments, we assume that the `CHTree` implementation uses the L2 cache to store internal hashes instead of having a dedicated IV cache. This implementation allows a large L2 cache

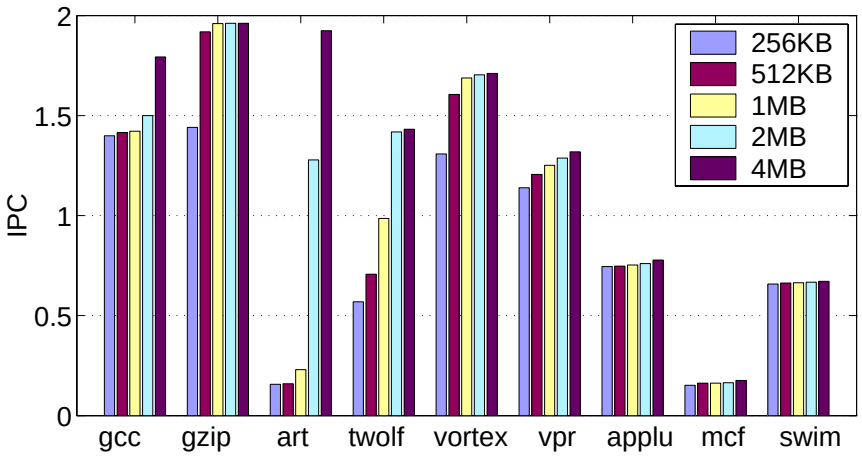


Figure 11-1: Baseline performance of simulated benchmarks for L2 caches with 64-B blocks.

to be shared between hashes and processor data, and eliminates the need of an extra IV cache.

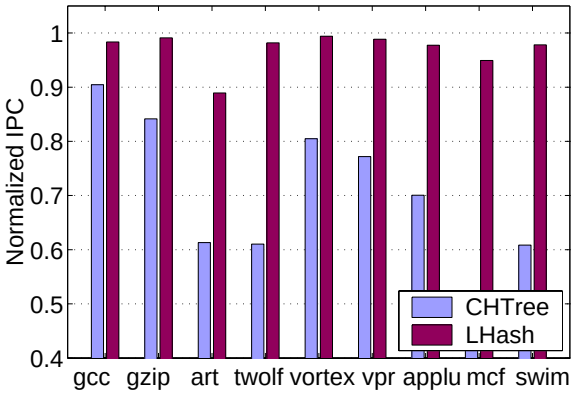
Run-Time Performance

We first investigate the performance of the integrity verification schemes ignoring the overhead of the integrity-check operation for `LHash`. For applications with very infrequent integrity checks such as certified execution, the overhead of the integrity-check operation is negligible and the results in this section represent the overall performance. The effect of frequent integrity checking is studied in the following subsection.

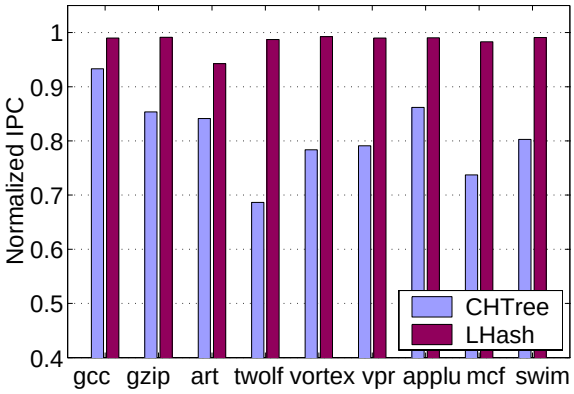
Figure 11-2 illustrates the impact of integrity checking on the run-time program performance. For four different L2 cache configurations, the normalized IPCs (instructions per clock cycle) of cached hash trees (`CHTree`) and log-hashes (`LHash`) are shown. The IPCs are normalized to the baseline performance that does not have any protection mechanisms.

The cached hash tree `CHTree` results in a 20-30% overhead in general, and has as much as 50% overhead in the worst case. This results match fairly well with the performance slowdown observed in our embedded processor implementation. On the other hand, the run-time overhead of the `LHash` scheme is often less than 5% and less than 15% even for the worst case. Only considering the run-time overhead and ignoring the integrity-check overhead, the results clearly show the potential advantage of the `LHash` scheme over the `CHTree` scheme.

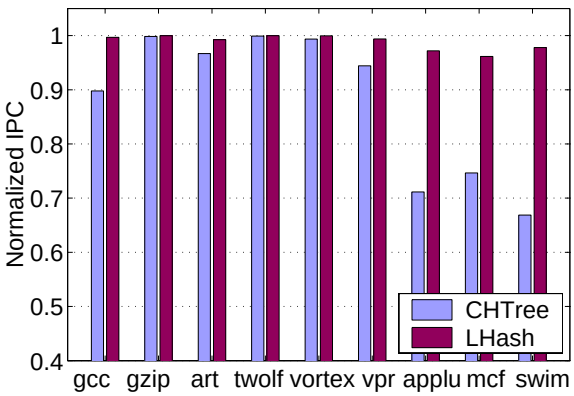
The figure also demonstrates the general effects of cache configuration on the memory



(a) 256KB, 64B



(b) 256KB, 128B



(c) 4MB, 64B

Figure 11-2: Run-time performance overhead of memory integrity checking: cached hash trees (`CHTree`) and log-hashes (`LHash`). Results are shown for two different cache sizes (256KB, 4MB) and cache block size of 64B and 128B. 32-bit time stamps and 128-bit hashes are used.

Bench -mark	L2 Data Miss-Rate (%)					
	256KB			4MB		
	Baseline	CHTree	LHash	Baseline	CHTree	LHash
gcc	2.92	3.46	2.93	1.06	1.74	1.06
gzip	16.04	23.77	16.04	1.10	1.10	1.10
art	63.19	63.77	63.19	0.91	0.91	0.91
twolf	36.10	52.05	36.10	0.65	0.65	0.65
vortex	9.07	14.85	9.07	1.30	1.31	1.30
vpr	30.24	41.95	30.24	16.65	18.28	16.65
applu	29.10	29.41	29.10	29.09	29.09	29.09
mcf	49.56	55.32	49.56	41.53	42.71	41.53
swim	24.18	27.29	24.18	23.68	23.69	23.68

Table 11.8: L2 miss-rates of program data for integrity verification schemes.

integrity verification performance. The overhead of integrity checking decreases as we increase either cache size or cache block size. Larger caches result in fewer memory accesses to verify and less cache contention between data and hashes. Larger cache blocks reduce the space and bandwidth overhead of integrity checking by increasing the chunk size. However, we note that increasing the cache block size beyond an optimal point degrades the baseline performance.

Integrity checking impacts the run-time performance in two ways: *cache pollution* and *bandwidth consumption*. The `CHTree` scheme stores its hash nodes in the L2 cache with program data. As a result, the cache miss-rate of the program data can be increased by the schemes. All integrity checking schemes use additional off-chip bandwidth compared to the baseline case, to access hashes or time stamps. Consuming more bandwidth may delay program memory accesses and increase effective latency.

Table 11.8 illustrates the effects of integrity checking on cache miss-rates. Since `LHash` does not store hashes in the cache, it does not affect the L2 miss-rate. However, `CHTree` can significantly increase miss-rates for small caches when it stores its hash nodes in the L2 cache with program data. In fact, the performance degradation of the `CHTree` scheme for *cache-sensitive* benchmarks such as `gzip`, `twolf`, `vortex`, and `vpr` in the 256-KB case (Figure 11-2) is mainly due to cache pollution. As the cache size increases, cache pollution becomes negligible as both data and hashes can be cached without contention.

The bandwidth consumption of the integrity checking schemes is shown in Figure 11-3. The `LHash` scheme theoretically consumes 6.25% to 12.5% of additional bandwidth compared to the baseline because it accesses the 32-bit time stamps for each 64-B cache block.

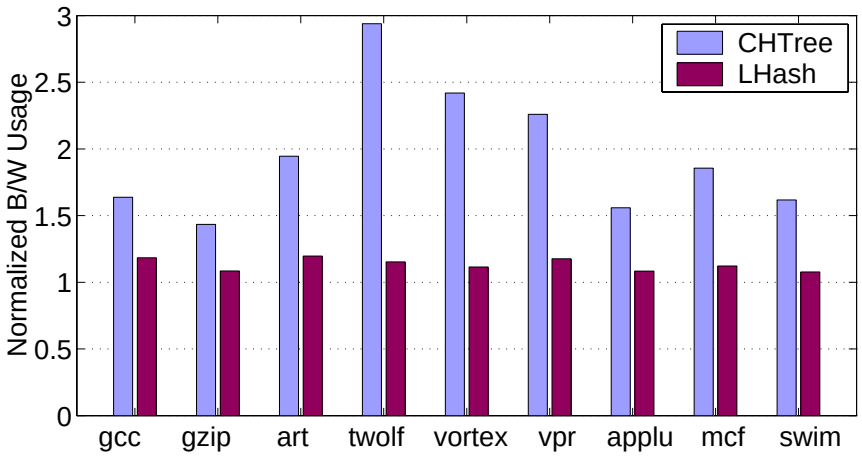


Figure 11-3: Off-chip bandwidth consumption of memory verification schemes for a 1-MB L2 with 64-B blocks. The bandwidth consumption is normalized to the baseline case.

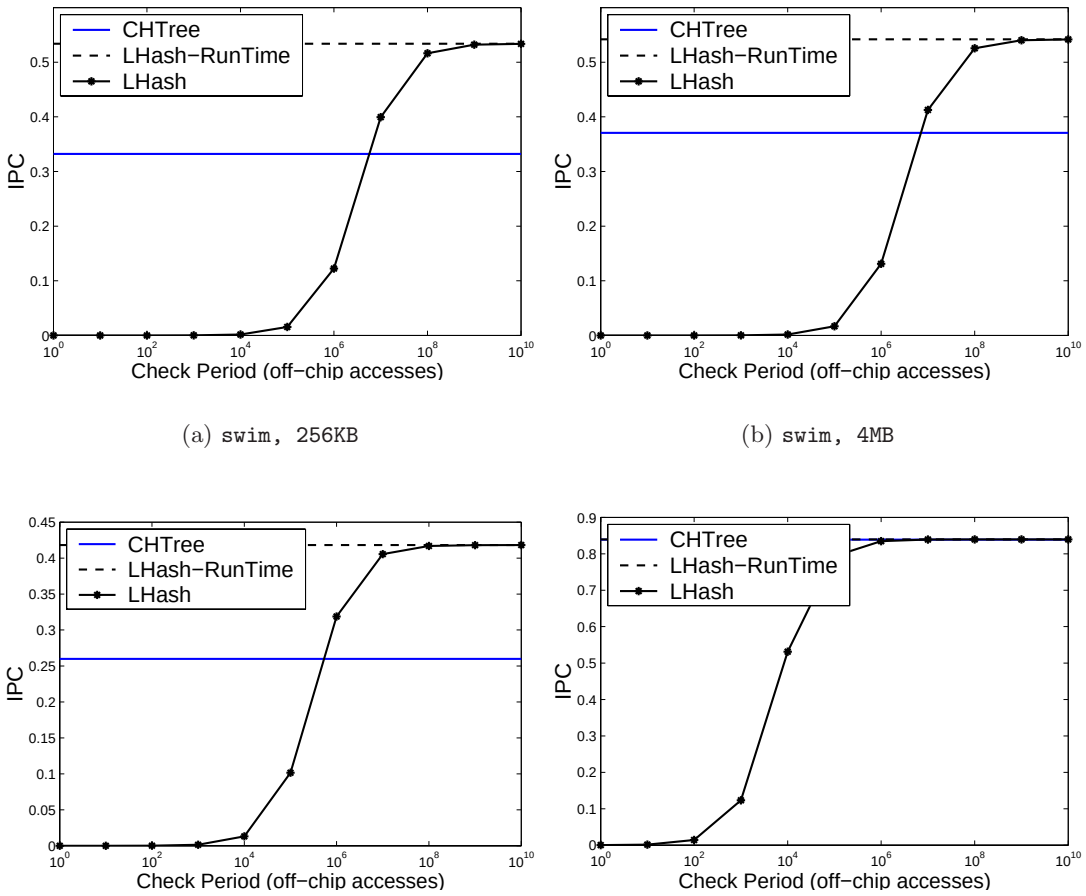
In our processor implementation, however, it consumed more (8.5% to 20%) because the bus width is 8B while the time stamps are only 4B. `CHTree` consumes additional bandwidth for hashes, which can depend significantly on the L2 cache performance. For *bandwidth-sensitive* benchmarks, bandwidth overhead directly translates into performance overhead. This makes the `LHash` scheme attractive even for processors with large caches where cache pollution is not an issue.

Overall Performance

The experiments in the last subsection demonstrated that the `LHash` scheme outperforms the hash tree scheme when integrity-check operations are ignored. In this subsection, we study the integrity checking schemes including the overhead of periodic integrity-check operations.

Let us assume that the `LHash` schemes check memory integrity every T memory accesses. A processor executes a program until it makes T main memory accesses, then checks the integrity of the T accesses by performing an integrity-check operation. Obviously, the overhead of the checking depends heavily on the characteristics of the program as well as the check period T. We use two representative benchmarks `swim` and `twolf` – the first consumes the largest amount of memory and the second consumes the smallest. `swim` uses 192MB of main memory whereas `twolf` uses only 2MB of memory. Here, we assume that the integrity-check operation only checks the memory space used by a program.

Figure 11-4 compares the performance of the memory integrity checking schemes for



(a) `swim, 256KB` (b) `swim, 4MB`

(c) `twolf, 256KB` (d) `twolf, 4MB`

Figure 11-4: Performance comparison between `LHash` and `CHTree` for various checking periods. `LHash-RunTime` indicates the performance of the `LHash` scheme without checking overhead. Results are shown for caches with 64-B blocks. 32-bit time stamps and 128-bit hashes are used.

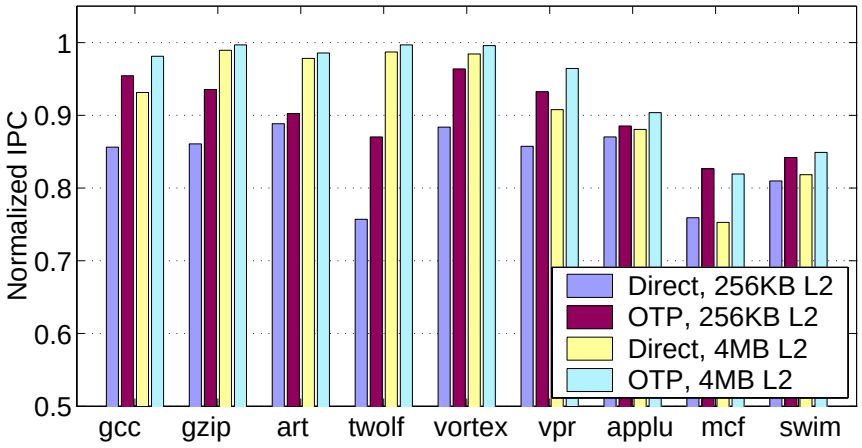


Figure 11-5: The overhead of direct encryption and OTP encryption (64-B L2 blocks).

varying check periods. The performance of the conventional `CHTree` scheme is indifferent to the checking period since it has no choice but to check the integrity after each access.

On the other hand, the performance of the log-hash scheme (`LHash`) heavily depends on the checking period. The `LHash` scheme is infeasible when the application needs to assure memory integrity after a small number of memory accesses. In this case, `CHTree` should be used. The performance of `LHash` converges to the run-time performance for a long period such as hundreds of millions to billions of accesses. Therefore, the `LHash` scheme would be a better solution when the secure processor is mainly used for applications that only need to verify the integrity of a long execution as in certified execution. The `CHTree` provides a more general solution that can be used for a wide range of applications.

11.4.3 Memory Encryption

For privacy (PTR mode), the processor also needs to encrypt and decrypt the off-chip memory accesses. In this subsection, we evaluate the OTP encryption scheme compared to conventional direct encryption. Here, we only compare the performance overheads from encryption without integrity verification. The next subsection discusses the overall performance overheads for PTR mode when both integrity verification and encryption are enabled.

Figure 11-5 compares the direct encryption mechanism with the one-time-pad (OTP) encryption mechanism. The instructions per cycle (IPC) of each benchmark is normalized to the baseline IPC. In the experiments, we simulated the 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 random vectors, and by delaying the data delivery for decryption.

As shown in the figure, the memory encryption for these configurations results in up to 18% performance degradation for the OTP encryption, and 25% degradation for the direct encryption. On average, the one-time-pad scheme reduces the overhead of the direct encryption by 43%. Our scheme is particularly effective when the decryption latency is the major concern. For applications with low bandwidth usage such as `gcc`, `gzip`, `twolf`, `vortex`, and `vpr`, the performance degradation mainly comes from the decryption latency, and our scheme reduces the overhead of the conventional scheme by more than one half.

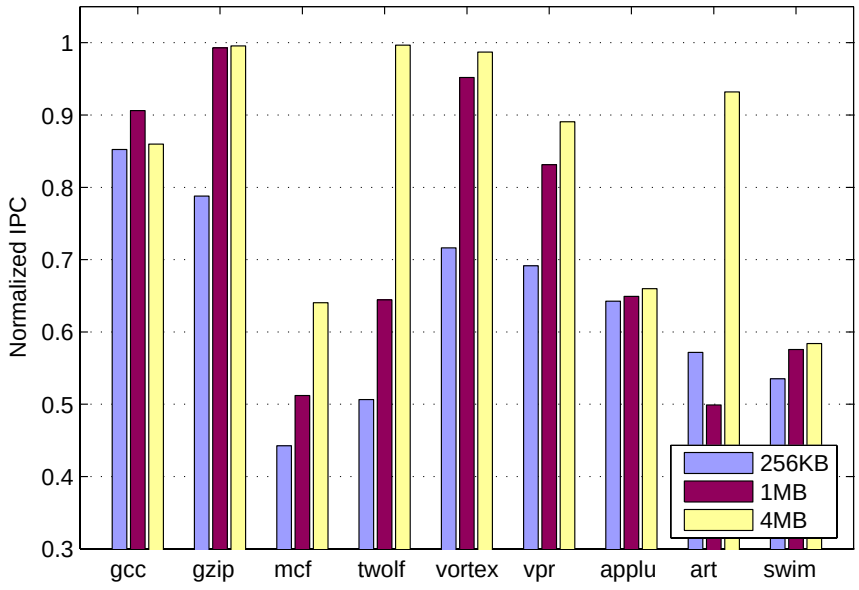
11.4.4 Re-Encryption Period

As noted in Section 4.2, the OTP encryption mechanism requires re-encrypting the memory when the global timer reaches its threshold value, which is close to the 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 re-encryption.

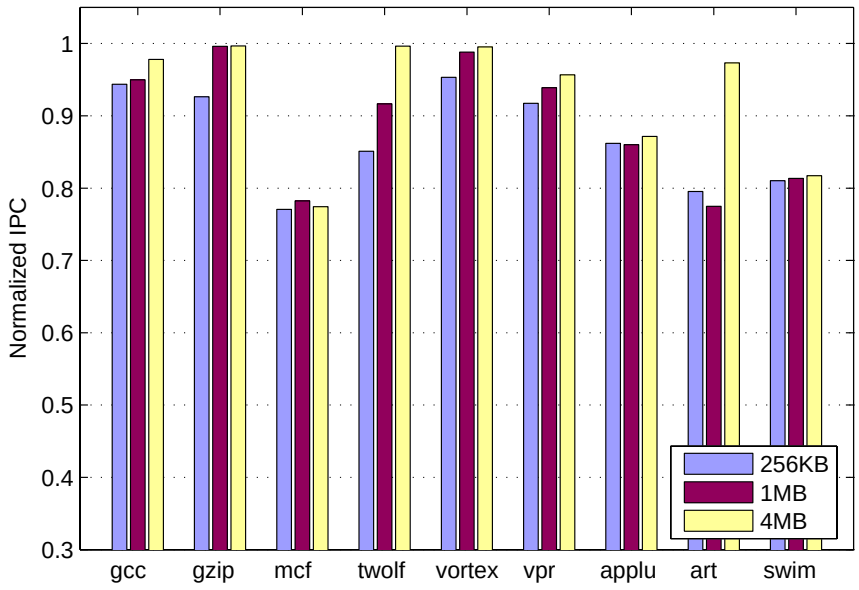
Fortunately, the simulation results for the SPEC benchmarks show that even 32-bit time stamps are large enough. In our experiments, the processor writes back to memory every 4800 cycles when averaged over all the benchmarks, and 131 cycles in the worst case of `swim`. Given the maximum time stamp size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

11.4.5 PTR Processing

Finally, we study the performance of the PTR processing by simulating integrity verification and encryption together. Figure 11-6 show the overhead of the PTR processing with the OTP encryption and the `CHTree` integrity verification as in our baseline architecture. The results demonstrate that PTR processing can be done with 60% overhead in the worst case (`mcf`), and less than 40% overhead in most cases. If the `LHash` scheme is used for



(a) OTP+`CHTree`



(b) OTP+`LHash`

Figure 11-6: The performance overhead of PTR processing. Results for three different L2 caches with 64-B blocks are shown.

applications that only require the integrity check at the end, the overhead can be reduced to 20% in the worst case. Given the trends 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 verified and encrypted.

Chapter 12

Related Work

12.1 Physical Random Functions

Researchers have proposed the addition of specific circuits that exploit manufacturing variations to uniquely identify each IC [76]. This work is similar to the PUF in a sense that the circuit exploits manufacturing variations. However, the focus of this work was simply on assigning a unique identifier to each chip, without having security in mind. As a result, this paper does not address issues such as preventing various attacks that try to extract the circuit responses or correcting errors to reliably re-generate the same responses, which are essential to authenticate an IC.

The concept of Physical Random Functions based on hidden circuit delay variations was originally developed in previous work [40, 41]. These papers introduce PUF circuits that directly measure the delay variations using ring oscillators rather than arbiters that relatively compare delays. A more recent paper [67] has introduced the arbiter-based circuit and shown the evaluation of an ASIC implementation that the discussions in this thesis are based on. The PUF protocols in Section 8.2 that manage Challenge-Response-Pairs (CRPs) as symmetric keys were also previously developed [39] without error correcting capabilities. Two Master theses [38, 74] provide good summaries of all previous work on PUFs.

This thesis extends previous work on PUFs in three ways. First, the PUF is enhanced with error correction so that the same secret key can be reliably re-generated on every evaluation of a PUF circuit. Second, a PUF protocol is developed to express private keys, and the existing protocol for symmetric keys is augmented with error correction. Finally, this thesis introduces an analytical model to understand PUF design issues and addresses

how the PUF circuit can be tested after manufacturing.

The circuit delay variation is not the only physical property that can be used for identification and authentication. Researchers have also proposed a technique called "Physical One-Way Functions" that use optical properties of each crystal to build a unique, unclonable key [98, 107]. This approach can be seen as another form of Physical Random Functions, and provides a high degree of physical security thanks to the complexity of crystals. However, crystals would be much more difficult to integrate into a single-chip processor as compared to the silicon PUFs in this thesis.

12.2 Off-Chip Memory Protection

12.2.1 Integrity Verification

The eXecute Only Memory (XOM) architecture [73] protects data stored in off-chip memory by appending the data blocks with a MAC of the address and the data value pair. The block's address is included in the MAC in order to prevent an adversary from copying blocks from one memory address to another. Similar integrity protection mechanisms have been applied to file systems such as the Protected File System (PFS) [133] and the Transparent Cryptographic File System (TCFS) [16]. The MAC-based integrity verification scheme in this thesis is essentially identical to the scheme used in XOM. However, because this approach is vulnerable to replay attacks [71, 126], the MAC scheme is only used for read-only memory regions in the AEGIS architecture.

Blum et al. [11] addressed the problem of securing various data structures in untrusted memory. They proposed using a hash tree rooted in trusted memory to check the integrity of arbitrarily large untrusted RAM. Their approach has a $O(\log(N))$ cost for each memory access. This hash tree (or Merkle tree) scheme is widely used to protect read-write data in software storage systems such as file systems [83] and databases [78]. The cached hash tree scheme (`CHTree`) in this thesis is also based on Blum's hash tree. However, unlike previous work, this thesis addresses the issues in implementing the hash tree in hardware, and shows how integrating the hash tree machinery with an on-chip cache can significantly reduce the memory bandwidth overhead and improve the performance. More detailed simulation studies and variants of the cached hash tree algorithm can be found in our conference publication [42].

The log hash (`LHash`) integrity verification scheme in Section 8.4 is inspired by Blum's offline memory checker that detects random errors in a sequence of memory operations [11]. The offline scheme computes a running hash of memory reads and writes. We have used the Blum's offline checker as a basis for designing the log hash (`LHash`) scheme. However, there are key differences between the two schemes. First, Blum's checker uses ϵ-biased hash functions [88]; these hash functions can be used to detect random errors, but are not cryptographically secure. On the other hand, the `LHash` scheme uses incremental multiset hash functions [19], which are cryptographically secure. Furthermore, the `LHash` scheme can use smaller time stamps compared to Blum's checker, which leads to better performance.

The main weakness of the `LHash` scheme is the cost of a check (`integrity-check`) operation. In the algorithm described in this thesis, this check operation requires the entire protected memory to be read. As a result, the `LHash` scheme can significantly outperform the hash tree when checks are infrequent, but suffers from a severe performance slowdown when checks are frequent. Recently, the `LHash` scheme has been extended to alleviate its checking overheads. A hierarchical approach has been proposed to reduce the size of memory that needs to be read for a check operation [135]. Also, an adaptive hybrid checker that dynamically chooses between the `LHash` scheme and the `CHTree` scheme has been developed [18, 21] to combine the best of both integrity verification schemes.

12.2.2 Encryption

XOM [73] and an early design of the AEGIS processor [136] directly use a block cipher to encrypt and decrypt memory. Unfortunately, this approach can considerably increase the memory access latency of reads as discussed in Section 4.2. The One-Time-Pad (OTP, or counter-mode) encryption scheme in this thesis improves the performance of memory encryption by enabling cipher computations to be performed in parallel to data reads from off-chip memory.

The idea of using one-time pads to decouple the cipher computation and data accesses from memory has also been proposed by Yang et al. [153] independently from our OTP scheme [134]. While the basic idea is the same, there exists one key difference between the two efforts. In their scheme, each cache block has its own sequence number stored in memory, which is used to determine the next sequence number to encrypt a cache block. In our OTP scheme, the time stamps in memory are only used for decryption, but not for

encryption; a single global timer inside a processor chip determines a new time stamp for encryption.

In terms of overheads, their scheme with per-block sequence numbers has an advantage over our OTP scheme. Because the per-block sequence numbers are incremented slower than the global timer, the size of sequence numbers can be smaller than time stamps without causing frequent re-encryption. Their work uses a smaller (16 Byte) sequence numbers whereas the OTP scheme in this thesis uses 32 Byte time stamps. They also study cases with relatively large caches to store the sequence numbers on-chip whereas our work only simulates cases with a small buffer for time stamps. As a result, their work reports only 1.28% performance slowdown compared to 8% slowdown of the OTP scheme in this thesis.

On the other hand, their scheme allows the same encryption pad to be used multiple times, which results in potential security problems. In their original scheme, Yang et al. do not consider re-encryption when a sequence number reaches its maximum values and wraps around. Therefore, the same sequence number can be used multiple times to encrypt a cache block. Their recent journal publication [154] mentions that the re-encryption is required.

However, in Yang's scheme, there still exist cases when the same encryption pad can be used multiple times. First, because a sequence number is stored in off-chip memory, an adversary can change the sequence number so that the processor re-uses the same sequence number multiple times for encryption. Therefore, to protect privacy, the processor must first verify the integrity of sequence numbers read from memory before using it to determine the next sequence number. The OTP scheme in this thesis does not have this problem because the global counter is on-chip where it cannot be tampered with.

Second, their scheme generates the encryption pad by performing the AES encryption on a seed, which is an addition of the cache block's virtual address (VA) and the per-block sequence number ($seq\_num_{VA,i}$). Therefore, if two cache blocks with $VA_1 = 0$ and $VA_2 = 4$ are encrypted and the two corresponding sequence numbers are related by $seq\_num_{0,i} = seq\_num_{4,i} + 4$, the same encryption pad will be used for both cache blocks. An adversary can find the XOR of the two cache blocks by performing the XOR of the two encrypted blocks in memory. This problem can be fixed by concatenating the virtual address with a sequence number rather than adding the two. Our OTP scheme guarantees that each encryption pad is unique by always using a new time stamp value from an on-chip timer on

each encryption.

Recently, prediction of sequence numbers to further optimize the encryption performance has been proposed [129]. This prediction technique utilizes idle cycles in the AES units to pre-compute an encryption pad without waiting for the actual sequence number from off-chip memory. This work reports between 15-40% performance improvement compared to the case with a 128-KB sequence number cache. However, this technique targets an OTP scheme with per-block sequence number similar to the one proposed by Yang et al. [153]. Because their prediction scheme exploits that multiple cache blocks are likely to have the same sequence numbers, it is not clear how well this technique will perform for the OTP scheme in this thesis that uses a single global timer.

12.2.3 Symmetric Multi-Processors (SMPs)

The memory protection in this thesis focuses only on uniprocessor systems. However, high-end server systems often have multiple processors such as symmetric multiprocessors (SMPs) that share the same off-chip memory. Multiple processors complicate off-chip protection in two ways. First, an SMP system requires that each processor, when it reads a value, be able to distinguish whether that value was written by one of the other processors as opposed to being written by an adversary. In a uniprocessor system, there is only one processor that is allowed to update the values in memory. Second, there exist processor-to-processor communications such as cache coherence protocols that must be checked.

Recently, researchers have proposed various ways to extend the off-chip protection scheme in uniprocessor systems so that they can be applied to SMP systems [22, 128, 158]. However, these efforts only focus on small-scale SMPs where all processors share a single bus. Moreover, some of the security and performance issues still remain to be addressed even for such small SMPs. The following paragraphs discuss each effort in more detail.

Clarke et al. proposed integrity verification algorithms for shared-bus SMP systems [22]. They separately protect the integrity of a shared off-chip bus and the integrity of off-chip memory. For bus authentication, two algorithms are proposed to either check each bus transaction one at a time or a sequence of transactions. For off-chip memory protection, the paper discusses the extensions to the uniprocessor algorithms such as `CHTree` and `LHash` for an SMP system. Finally, they introduce a protocol for all participating processors to communicate and ensure the integrity of previous operations. Unfortunately, this study

focuses only on algorithms without further discussions on practical implementation issues. While the proposed algorithms are secure, their performance implications have not been evaluated. Also, this work only addresses integrity verification, not privacy protection.

More recently, SENSS [158] has been proposed to address both integrity and privacy issues of SMP systems. The main contribution of SENSS is a fast encryption scheme for processor-to-processor communication based on the OTP encryption method. For off-chip memory protection, SENSS simply makes each processor use the `CHTree` scheme and the OTP scheme for a uniprocessor with cache coherence protocols to ensure that all processor see the same hash tree and time stamps. In SENSS, however, a few security issues remain to be addressed. First, to encrypt bus transactions, SENSS uses an OTP scheme with the encryption pad determined by the previous bus transaction. If there are two identical bus transactions, the same encryption pad will be re-used. Second, to verify the integrity of bus transactions, a check operation is periodically performed where each processor broadcasts its record of previous transactions. Unfortunately, checks are performed based on the number of bus transactions, not based on processor operations. As a result, the processor may be allowed to perform irrevocable operations without verifying the integrity of previous memory operations.

The previous two approaches assume that processor chips are the only trusted hardware components. Shi et al. proposed a different approach by trusting a memory controller chip (north bridge) as well as main processor chips [128]. They assume that all processors access the shared memory through a single memory controller chip. Therefore, the memory controller can protect the off-chip memory using uniprocessor schemes as if it is a single processor that owns the memory. The communications on a shared bus connecting processors and the memory controller are protected using MACs. However, the proposed scheme has a few limitations. First, in terms of security, their bus authentication scheme is vulnerable to attacks that drop messages as pointed out by Zhang et al. [158]. Also, this approach require all chips including processors and the memory controller to be made by the same manufacturer and contain a single secret key.

12.3 Trusted Computing Platforms

12.3.1 Secure Co-Processors

Secure co-processors such as IBM 4758 [131] have been proposed that encapsulate processing subsystems within a tamper-sensing and tamper-responding environment where secrets can be safely maintained and security-sensitive computations can be carried out. A processing subsystem contains the private key of a public/private key pair [30] and uses classical public key cryptography algorithms such as RSA [111] to enable a wide variety of applications. Using the private key, applications on the secure co-processor can authenticate themselves to remote parties (outbound authentication). This IBM 4758 architecture was independently validated at FIPS 140-1 Level 4 [130]. Smith's book [132] provides a detail description of IBM 4758 along with its predecessors such as ABYSS [145, 147], Citadel [97, 146], and Dyad [156, 157]. Recently, it has been reported that IBM produced a new secure co-processor called IBM 4764 that has the same basic architecture with IBM 4758 but with more computing power [7].

The IBM 4758 secure co-processor provides the same basic functionality as AEGIS; applications execute in secure environments with a capability to authenticate themselves to remote parties. The secure co-processor work has established the features required for remote parties to be able to trust computing devices. On the other hand, the cost and the computational power of AEGIS are significantly different from those of IBM 4758. Due to its tamper-proof package, IBM 4758 is relatively expensive (around $3K) and large. Also, the computational power of 4758 is limited hindering its usefulness. To maintain performance of the entire application, IBM 4758 has invariably been used as a co-processor executing only a small part of the application. AEGIS is cheap and small because only a single processor chip needs to be protected, and the security mechanisms can be easily integrated into high-performance processors.

12.3.2 Emerging Industry Platforms

Recently, there have been significant efforts in industry to build a secure platform for personal and pervasive computing. The Trusted Computing Group (TCG), which succeeded the Trusted Computing Platform Alliance (TCPA), is an organization formed by many hardware and software vendors to develop open standards for hardware-enabled security

technologies. They have developed an architecture based on a Trusted Platform Module (TPM), which is a small chip soldered to the motherboard [44]. Using a scheme similar to the integrity-checking boot process [6], the TPM records the platform configuration. Initially, the hash of the BIOS is recorded in the TPM, the BIOS sends the next block of code to the TPM, and so on. This scheme serves the same purpose as the program hashes in AEGIS identifying the software stack on the system. Then, the TPM can also perform outbound attestation and provide protected storage using its private key operations, which corresponds to AEGIS' private key instructions. However, the TPM architecture does not provide secure execution environments.

The LaGrande initiative [52] from Intel extends the TPM architecture to provide a more comprehensive secure computing platform. In addition to the TPM's protected storage and attestation capabilities, LaGrande also provides protected execution, and protected I/O such as keyboard, mouse and graphics. LaGrande's protected execution aims to isolate applications so that no other unauthorized software can observe or compromise the protected application. For this purpose, the LaGrande architecture divides execution environments into standard and protected partitions, similar to secure and insecure modes in AEGIS. System components including the processor, chipset, and OS are aware of this partition and manage resources separately. As a result, even an operating system, if it is in the standard partition, cannot tamper with applications in the protected partition.

Next Generation Secure Computing Base (NGSCB) [86] is a trusted computing architecture from Microsoft. NGSCB can be seen as more of an operating system architecture based on the hardware features provided by LaGrande than a separate hardware architecture. In NGSCB, the Nexus is a security kernel in the protected partition and the Nexus Computing Agents (NCAs) are the protected user applications.

TrustZone [2] is a security architecture emerging from ARM that targets small embedded devices such as mobile phones, PDAs, and set top boxes. Similar to LaGrande and NGSCB, TrustZone partitions execution modes into normal and secure. A single secure state bit inside the processor determines the state, and the cache and the MMU are augmented to isolate memory of the secure partition. Unlike LaGrande where the TPM is implemented as a separate chip, TrustZone implements all security features inside the main processor chip.

The outbound attestation, protected storage, and isolated execution environments in the TPM, LaGrande, and NGSCB are very similar to the features of AEGIS. Also, TrustZone's

partition of normal and secure modes is similar to AEGIS' standard and secure modes. However, none of these industry architectures handle hardware-based physical attacks. The TPM does not consider physical attacks on communication channels between itself and the main processor. Both LaGrande and NGSCB advertise that they only target defense against software attacks, not against physical attacks. While TrustZone implements all security features on-chip, it does not protect off-chip memory components from physical attacks.

Because the protection mechanisms in this thesis are modular, it is also possible to enhance the security of LaGrande, NGSCB, and TrustZone by applying those protection mechanisms. For example, physical random functions can be used to express secret keys inside the TPM or a TrustZone enabled processor. With integrity verification, applications can obtain 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.

12.3.3 Execution Only Memory (XOM)

The eXecute Only Memory (XOM) architecture [71, 72, 73] is designed to handle a security model similar to ours where both software and physical attacks are possible. In XOM, security requiring applications run in secure compartments, where both instructions and data are encrypted and from which data can escape only on explicit request from the application itself. XOM assumes that operating systems are completely untrusted and potentially malicious.

Although the security model is similar, the AEGIS architecture is different from XOM in various ways. The baseline AEGIS architecture uses a security kernel to handle multi-tasking and provide security features for user applications, which significantly simplifies the architecture compared to XOM. Also, the AEGIS architecture provides flexibility for applications to use protection mechanisms such as encryption and integrity verification only when they are necessary, avoiding unnecessary performance degradation whereas XOM requires all instructions in a secure compartment to be encrypted and verified.

This thesis also describes a variant architecture without any security kernel. This variant architecture has drawn insight from XOM, notably for the on-chip data tagging mechanism and the saving of contexts. However, our implementation of the context manager is different

from XOM because we use hash-trees to verify process state, which can be stored in off-chip memory. This allows AEGIS to support a much larger number of secure processes. Compared to XOM, AEGIS also has different on-chip tagging mechanism preventing replay attacks on on-chip cache data, and does not allow a malicious OS to fork secure processes.

Besides architectural differences, physical random functions (PUFS) and off-chip memory protection mechanisms in this thesis can be separately applied to XOM to enhance its security. XOM requires a private key inside the processor where PUFs can be used. Also, XOM's off-chip integrity verification mechanism is vulnerable to replay attacks, which was pointed out by Shapiro [126]. In particular, XOM will not notice if writes to memory are sometimes ignored. XOM can be fixed by using the integrity verification mechanisms in this thesis.

12.3.4 SP Architecture

Lee et al. at Princeton recently proposed a secret-protected (SP) architecture [69]. In the SP architecture, the goal is to protect critical secrets and securely perform sensitive computations such as encryption/decryption and signing with the secrets. Unlikely other trusted platforms, the SP processor does not contain any permanent secret key. Instead, a user master key is generated by hashing a pass-phrase typed in by a user. A special I/O devices protect the pass-phrase. Once the master key is generated, it can be used for various cryptographic operations. The SP processor provides a concealed execution mode, which is isolated from standard user and supervisor modes, and protects its data in caches and memory. Therefore, critical secrets can be protected from malicious software. Finally, the off-chip memory is protected using integrity verification and encryption mechanisms introduced in this thesis.

Compared to AEGIS, the SP processor includes secure I/O and partitions the execution mode differently without a security kernel because it is specialized to solve a very different problem of key management. On the other hand, the SP processor cannot solve the problem of trusting remote computation, which is the main focus of this thesis.

12.3.5 Software Virtual Machine

Garfinkel et al. proposed Terra [37], a virtual machine based architecture for trusted computing. Using a trusted virtual machine monitor (TVMM), Terra provides each application

with the semantics of running on a separate, dedicated hardware platform. In addition to the strong isolation of each application from software, Terra provides three additional capabilities for trusted computing that are not found in traditional virtual machines; privacy and isolation guarantees that cannot be violated even by the platform administrator, attestation, and trusted I/O paths. However, Terra does not replace the hardware trusted computing platforms. In fact, Terra relies on the presence of various hardware supports such as hardware attestation of the booted operating system, protected storage, secure I/O, etc. Therefore, Terra can be seen as a software layer that virtualizes a trusted hardware platform such as AEGIS.

12.3.6 Software Attestation and Tamper Resistance

Recently, researchers have investigated the possibility of providing trusted computing features purely in software without special hardware support. Horne et al. [48] proposed dynamic self-checking as a way to improve tamper resistance in software. In their technique, a program checks itself during an execution to ensure that it has not been modified. Combined with code obfuscation techniques, the goal is to require an attacker to reverse-engineer a significant portion of a program in order to modify the program's code.

Software-based attestation schemes have also been proposed as a way to authenticate remote hardware and software without secure hardware support. Kennell et al. [59] proposed a software technique called Genuinity that determines whether a remote machine is a real computer running the expected software or not based on checksums that are difficult to quickly simulate. Seshadri et al. [122] presented a similar technique called SWATT (SoftWare-based ATTestation) that verifies the memory contents of isolated embedded devices

Unfortunately, many limitations have been found in such software-only approaches without hardware support. For example, Wurster et al. have shown that dynamic self-checking mechanisms can easily be bypassed without reverse-engineering and removing the checks from the code if the operating system is malicious [141, 150]. They exploit the virtual-to-physical address translation mechanisms in modern processors to have modified code running while the checks are computed using the original code. Similarly, Shankar et al. [124] have demonstrated various attacks on Kennell's Genuinity technique. In general, hardware attestation support is required to authenticate remote hardware and software.

12.4 Physical and Side-Channel Attacks

Making an IC tamper-resistant to physical and side-channel attacks is a challenging problem [3]. Numerous attacks are described in the literature and have been used to break the security of smartcards [4, 5, 61, 62, 64, 105]. In general, these attacks can be categorized into invasive and non-invasive attacks. Invasive attacks on on-chip components typically require de-packaging and sophisticated micro-probing techniques [64, 105]. As a result, expensive equipment is often necessary for invasive attacks making them difficult to mount. On the other hand, non-invasive attacks do not open a chip and tend to be cheaper and more scalable.

The AEGIS security model in Chapter 2 assumes that the main processor chip is protected from physical and side-channel attacks. Either programs must be written in a way to prevent information leaks through side channels or the processor should be equipped with hardware protection mechanisms. This section summarizes most plausible side-channel attacks and protection methods. Because invasive attacks on on-chip components are expensive and unlikely for most adversaries, this section focuses on non-invasive attacks, especially side-channel attacks.

12.4.1 Memory Access Patterns

Off-chip memory access patterns can indirectly reveal sensitive information. For example, if a program accesses memory location A when a secret is 0 and accesses memory location B otherwise, the memory accesses leak that secret indirectly. Also, in intellectual property (IP) protection, the addresses of instructions fetched from memory can reveal the control flow of protected software even when it is encrypted.

Oblivious RAMs [43] introduce algorithms to completely stop any information leak through memory access patterns. The algorithms make the sequence of accessing memory locations equivalent for any two inputs with the same running time. Although this work shows theoretical solutions that are proven to be completely secure, the overhead of the proposed algorithms makes them impractical.

More recently, HIDE [159] proposed an architectural support to obfuscate memory access patterns. HIDE uses the idea of probabilistic oblivious RAM [43] and protects information leaks by randomly shuffling memory locations before the same location gets accessed twice.

However, in order to have reasonable performance overheads, HIDE only applies this technique within a small chunk of memory (usually 8 KB to 64 KB). Memory accesses within a single chunk is obfuscated, but access patterns for different chunks are observable by an adversary. While HIDE provides a practical way of obfuscating memory access patterns, it still remains an open question whether a practical and completely secure mechanism can be built.

12.4.2 Timing

Processors take time to carry out computation. Therefore, if the computation time depends on secret data, an adversary may be able to obtain the secret by monitoring the duration of computation. For example, Kocher has shown that an adversary can factor RSA keys by carefully measuring the amount of time required to perform private key operations [61]. More recently, researchers at Stanford has shown that even remote timing attacks are possible by extracting private keys from an OpenSSL-based web servers running in the local network [14].

One defense to the timing attack is to make the computation time independent of the sensitive input data. For RSA, Brumley's paper summarizes various counter measures that change the RSA computation itself. It is also possible to design the hardware to take constant time for each operation if the hardware is specialized for certain operations. For more general programs, Agat proposed program transformations that ensure all execution paths depending on sensitive data carry out the same number of operations [1].

12.4.3 Others

In addition to memory access patterns and computation time, computing devices also have other observable physical characteristics. For example, researchers have developed attacks that extract secrets from power traces of a device. In standard CMOS circuits, to a first approximation, power is only drawn from the power supply when zero to one transition occurs. Therefore, monitoring the power consumption can reveal the internal operations of an IC. For example, simple power analysis (SPA) tries to directly identify secret data from power traces [85]. On the other hand, a more advanced technique called differential power analysis (DPA) uses subtle statistical correlations between the secret and the dissipated power [23, 62].

As power analysis has become one of the major security concerns for small devices such as smartcards, various protection techniques have been proposed [24, 79, 103, 123, 138, 139, 140]. For example, Tiri et al. have developed a logic style called Wave Dynamic Digital Logic (WDDL) and a layout technique called differential routing to make circuits more resistant to DPA attacks [138]. They have implemented a security chip called ThumbPod with these technique and observed an improvement in DPA resistance of two orders of magnitude.

Computing devices also generate electromagnetic radiation that depends on their internal operations. It has been shown that adversaries with intimate knowledge of the chip layout can extract sensitive information using electromagnetic analysis [36, 104]. As natural hardware counter-measures, researchers have suggested an upper metal layer to contain the radiation, variable random currents to blur the radiation, and more advanced manufacturing technology to reduce the radiation [36].

Papers on embedded system security and smartcards [63, 106, 108] provide more details of the power and electromagnetic analysis as well as other types of side-channel attacks. So far, the side-channels attacks have mostly been studied for small devices such as smartcards, which are specialized for a particular task. It still remains to be seen whether such attacks can also apply to a general-purpose processor such as AEGIS that performs complex tasks with many more hardware components on-chip. Applying previously proposed counter-measures to the main processor also needs more study.

12.5 Software Vulnerabilities

The AEGIS processor provides secure execution environments that protects programs from both software and physical attacks. However, the secure processor assumes that protected programs are well-written, and does not prevent attacks exploiting software bugs such as buffer overflows [95, 148] or format string vulnerabilities [92, 121]. This section briefly summarizes some successful approaches to automatically detect and prevent such attacks. These techniques can be used to further secure the AEGIS system.

12.5.1 Safe Languages and Static Checks

Safe languages such as Java, and safe dialects of C such as CCured [91] and Cyclone [55] can eliminate most software vulnerabilities using strong type systems and run-time checks. However, programs must be completely rewritten in a safe language or ported to safe C in order to take advantage of the safe languages. Moreover, the safe languages are often less flexible and result in slower code compared to C.

Various static analysis techniques are proposed to detect potential buffer overflows [35] or format string vulnerabilities [125]. While these techniques can detect many errors at compile time, they all suffer from two weaknesses; they often cannot detect errors due to the lack of run-time information, and they tend to generate considerable false errors that need to be inspected by programmers. Run-time mechanisms are still required to protect undetected errors even after the programs are inspected by static analysis tools.

12.5.2 Dynamic Checks in Software

Many compiler patches are developed to automatically generate binaries with dynamic checks that can detect and stop malicious attacks at run-time. Early compiler patches such as StackGuard [27] and StackShield [142] checked a return address before using it in order to prevent stack smashing attacks. While these techniques are effective against stack smashing with near-zero performance overhead, they only prevent one specific attack.

PointGuard [26] is a more recent proposal to protect all pointers by encrypting them in memory and decrypting them when the pointers are read into registers. While PointGuard can prevent a larger class of attacks, the performance overhead can be high causing about 20% slowdown for openSSL.

Bound checking provides *perfect* protection against buffer overflows in instrumented code because all out-of-bound accesses are detected. Unfortunately, the performance overhead of this perfect protection while preserving code compatibility is quite high; checking all buffers incurs 10-30x slowdown in the Jones & Kelly scheme [56], and checking only string buffer incurs up to 2.3x slowdown [114].

On top of considerable performance overhead, compiler patches have the following weaknesses. First, they need re-compilation which requires access to source code. Therefore, they cannot protect libraries without source codes. Also, some techniques such as PointGuard

require users to annotate the source program for function calls to uninstrumented code.

Program shepherding [60] monitors control flow transfers during program execution and enforces a security policy. For example, program shepherding can ensure that only code that is originally loaded can be executed. Because most attacks need to re-direct the control flow, shepherding provides general protection against a large set of attacks exploiting software bugs. On the other hand, program shepherding can incur considerable overheads because it is implemented based on a dynamic optimization infrastructure, which is an additional software layer between a processor and an application. The space overhead is reported to be 16.2% on average and 94.6% in the worst case. Shepherding can also cause considerable performance slowdown: 10-30% on average, and 1.7x-7.6x in the worst case.

12.5.3 Library and OS Patches

Library patches such as FormatGuard [25] and Libsafe [9] replaces vulnerable C library functions with safe implementations. They are only applicable to functions that use the standard library functions directly. Also, FormatGuard requires re-compilation.

Kernel patches enforcing non-executable stacks [29] and data pages [99] have been proposed. However, code such as `execve()` is often already in victim program's memory space as a library function. Therefore, attacks may simply bypass these protections by corrupting a program pointer to point to existing code. Moreover, non-executable memory sections can also prevent legitimate uses of dynamically generated code.

12.5.4 Hardware Protection Schemes

Both AMD and Intel recently added No eXecute (NX) bits to their processors [46, 51], which support non-executable data pages in the x86 architecture. This hardware extension prevents a class of attacks that inject malicious code into data pages exploiting software bugs. On the other hand, the NX bits have the same limitations as the kernel patches enforcing non-executable stacks [29]. They cannot prevent attacks that simply change the control flow without injecting new code, and may prevent legitimate uses of dynamically generated code.

Recent works have proposed hardware mechanisms to prevent stack smashing attacks [152, 68]. In these approaches, a processor stores a return address in a separate return address stack (RAS) and checks the value in the stack on a return. Unfortunately, this

approach only works for very specific types of stack smashing attacks that modify return addresses.

Mondrian memory protection (MMP) [149] provides fine-grained memory protection, which allows each word in memory to have its own permission. MMP can be used to implement non-executable stacks and heaps. It can also detect writes off the end of an array in the heap. However, MMP implementations that place inaccessible words before and after every malloc'ed region incur considerable space and performance overheads. Even then, MMP cannot prevent many forms of attacks such as stack buffer overflows and format string attacks.

Dynamic information flow tracking [137] protects programs against malicious software attacks by identifying spurious information flows from untrusted I/O, tracking the flows at run-time, and restricting their usage. In this approach, the operating system identifies a set of input channels as spurious, and the processor tracks all information flows from those inputs. A broad range of attacks are effectively defeated by checking the use of the spurious values as instructions and pointers. Because the protection is done in hardware, this approach incurs relatively low overheads on average: a memory space overhead of 1.4% and a performance overhead of 1.1%.

Arora et al. [8] proposed a hardware-assisted method to protect embedded systems from malicious software attacks. In their approach, static program analysis extracts properties of a normal program behavior such as control flows. At run-time, a hardware monitor observes the processor's dynamic execution trace, checks whether the execution trace falls within the allowed program behavior, and flags any deviations. Therefore, this approach can prevent any attack that makes a program deviate from its normal behavior. On the other hand, this solution requires each program source code to be statically analyzed and the hardware monitor to be programmed for each program.

12.6 Application Studies

This thesis briefly describes how AEGIS, or trusted computing in general, can be applied to a set of applications such as distributed computation, mobile agents, DRM, sensor networks, etc. On the other hand, building a real application on AEGIS requires many details to be worked out. This section summarizes efforts made to apply trusted computing platforms

to strengthen the security of various applications. While most of these efforts use existing commercial platforms such as IBM 4758 and TPM, AEGIS can also be used to enable the same applications.

Yee, in his doctoral thesis [157], proposed host integrity checks, secure audit trails, copy protection, electronic currency, and secure postage as applications of secure co-processors. Also, Yee investigated the use of secure co-processors for mobile agents [155].

Recently, many more applications have been proposed for trusted computing. Itoi used an IBM 4758 to protect the key in Kerberos V5 [53], and Lorch et al. used the co-processor to secure credential repository for Grid computing (distributed computing) [77]. Perrig et al. proposed securing electronic auctions with secure co-processors [100]. Sandhu et al. proposed to implement access control in peer-to-peer environments using trusting computing technology [117]. Finally, Xia et al. have proposed to implement access control for enterprise networks using secure co-processors [151]. The Trusted Computing Group (TCG) also recently proposed using the TPM hardware and its attestation capability to control accesses to networks [45].

Sean Smith's group at Dartmouth has experimented with both the IBM 4758 secure co-processor and the TPM hardware. Using the IBM 4758, they implemented applications such as a hardened web server [54], protecting user privacy in archives of LAN traffic [49], and private information retrieval [50]. With the TPM hardware, they built a virtual secure co-processors [81, 82], and used that platform to harden three sample applications: Apache SSL web servers, OpenCA certificate authorities, and compartmented attestation [80]. Smith's book [132] provides a good overview of their work as well as some other related projects.

In addition to applying trusting computing on various applications, researchers have investigated the impact of trusted computing. Erickson [34] examined how the trusted computing environments with DRM will affect our personal use of information and pointed out the challenges and potential negative effects. Schechter et al. discussed the impact of trusting computing platforms on the entertainment industry [120]. While trusted computing technologies can better protect media from pirates, they point out that the trusted computing platforms can also be used to better protect pirates and their peer-to-peer distribution networks from the entertainment industry.

Chapter 13

Conclusion

This chapter concludes the thesis. The first section summarizes the security challenges addressed by AEGIS and the solutions that this thesis has proposed. Then, the second chapter discusses how the AEGIS secure processor can be improved in the future and applied in practice.

13.1 Summary

This thesis has described the AEGIS processor that can be used as a main processor to build a secure computing system. AEGIS enables a remote party to trust an AEGIS platform even when the platform is physically exposed to an adversary. More specifically, the security features in AEGIS allow remote parties to authenticate the hardware and software of the platform, and trust the integrity and the privacy of programs executing on it. These capabilities of outbound attestation and secure execution can be applied to many application scenarios including distributed computation, DRM, software IP protection, mobile agents, etc. As an extension, AEGIS also provides a secure booting feature that gives the hardware vendor control of software that can execute on each processor.

Unlike previous approaches to build a trusted computing platform, this thesis has investigated the approach where the AEGIS processor chip is the only trusted hardware component. At the same time, AEGIS still allows other off-chip components such as memory to be used for computation. Being a secure main processor, AEGIS enables a cheap, computationally powerful, and secure platform when compared to previous work in this area. For example, secure co-processors require multiple chips in a system to be protected

by expensive tamper proof packaging. Smartcards are cheap, but with limited computational power because all resources need to fit in a single chip. Finally, auxiliary security chips such as TPMs are insecure against physical attacks.

Building a secure platform with all security features in a single processor chip requires new security mechanisms. Physical random functions (PUFs) provide a cheap and secure way to authenticate each processor's hardware. Memory encryption and integrity verification mechanisms protect the confidentiality and the integrity of off-chip memory from physical attacks.

The thesis also contributes mechanisms to reduce the trusted computing base in software. First, the processor architecture supports suspended secure processing (SSP) which allows programs to be partitioned into secure and insecure parts, and isolates a secure part of a program from other potentially malicious or buggy software. Second, a variant processor architecture can ensure the security of user applications without trusting any part of an operating system.

In AEGIS, these security mechanisms are combined in a single processor to provide a strong security guarantee against both software and physical attacks. However, because the mechanisms are independent of each other, they can also be applied separately to applications that only require a subset of AEGIS' features. For example, PUFs can be separately applied to smartcards, RFIDs, and key cards. The encryption mechanism can protect the confidentiality of software IPs on off-chip flash memory or ROM even for small devices without off-chip DRAM.

The costs of these new security mechanisms are minimal in most cases. The mechanisms other than encryption and integrity verification do not incur any performance degradation or additional memory space usage. They also have minimal hardware resource usage; the PUF circuit consumes only a few thousand gates, and architectural support for SSP consumes about 12K gates. Off-chip protection mechanisms have more noticeable costs. However, the costs are modest for most applications. For example, encryption of read-write data incurs about 8% performance slowdown on average with 6.25% additional memory space usage. The integrity verification (`CHTree`) causes 22% performance slowdown on average and consumes 33.3% additional memory space. Both encryption and integrity verification require about 100K gates each in hardware.

Finally, the thesis has discussed the software support for the AEGIS processor. First,

the security kernel, which is a trusted core part of an operating system, handles the security issues related to multi-tasking in the baseline AEGIS architecture. The thesis has discussed the security functions required in the security kernel and how the security kernel can use the processor architectural features to provide protection for itself and user applications.

Second, in order for the secure processor to be useful in practice, programmers must be able to use the security features in a high level language. The thesis has described a programming model based on the C language, and the details of how the high-level programming abstractions can be implemented by a compiler for the AEGIS processor architecture.

13.2 Future Research

The AEGIS processor architecture and its security mechanisms in this thesis enable trusted remote interactions with reasonable overheads, while providing strong security guarantees. However, there are many research opportunities to enhance the security of the current design, reduce the overheads, and expand its applicability to a wider range of application scenarios. Some work on these research problems has already been described in the related work chapter.

13.2.1 Security Enhancements

The AEGIS architecture has been built upon a security model (see Chapter 2) which assumes that the processor chip is protected from invasive attacks and side-channel attacks such as timing attacks, power analysis, and electromagnetic radiation analysis. These attacks and their counter-measures have been extensively studied for smartcards. On the other hand, the studies tend to only focus on small circuits that are specialized for cryptographic operations. Whether these attacks are feasible for complex main processors, and if so, whether the existing counter-measures apply are questions that require further studies to provide a stronger foundation to build a singe-chip secure processor.

Besides common side-channel attacks on smartcards, the secure processor with off-chip memory must also protect against attacks exploiting memory access patterns observable on the memory address bus. Unfortunately, because smartcards do not generally have off-chip memory, the protection of the memory bus has not been studied as much as the counter-

measures against other physical attacks. Oblivious RAM [43] provides theoretical studies and HIDE [159] implements a variant of the idea in practice. However, it is still an open problem to develop a provably secure mechanism with practical overheads.

Secure environments in AEGIS ensure the integrity and the privacy of applications during execution. Also, the application can encrypt its data in non-volatile memory to ensure privacy while it is not executing. However, the current AEGIS architecture does not provide mechanisms that can ensure the freshness of data in local non-volatile storage. For example, let us say that a DRM player saves the number of times a particular media file has been played in local hard-disk. If the system gets rebooted and the player restarts, the player cannot guarantee that the number it reads from hard-disk is the most recent one that it saved. To guarantee the freshness of local non-volatile data even from physical attacks, the processor needs physically secure non-volatile storage, which AEGIS does not have.

Finally, the current AEGIS architecture does not address the problem of software vulnerabilities such as buffer overflows. While the secure modes within a process can isolate a secure part of a program from vulnerabilities in an insecure part, the processor cannot prevent attacks exploiting bugs within the secure part of an application. It is an interesting research question as to how the processor architecture can protect applications from such malicious software attacks.

13.2.2 Protection Overheads

Off-chip memory protection mechanisms in AEGIS incur performance overheads because they add additional latencies to memory accesses and consume additional bandwidth for meta-data. Recently, there have been some efforts to improve the performance of the OTP encryption scheme using either on-chip caches [154] or prediction of time stamps (sequence numbers) [129]. With these optimizations, the slowdown due to encryption can be reduced to a few percent. On the other hand, integrity verification still remains to be the most expensive protection scheme costing up to 52% slowdown when `CHTree` is used. One promising approach to improve the integrity verification seems to be a hybrid of `CHTree` and `LHash` called tree-log [21]. It remains open as to how the tree-log algorithm can be applied to a secure processor to reduce performance overhead.

The evaluations in this thesis have mainly focused on the performance and memory space overheads of protection schemes. However, the security mechanisms also consume

additional energy, which is probably the most important overhead in mobile and embedded devices. For example, encrypting off-chip traffic is likely to increase the number of bit flips on the memory bus consuming more energy. Thus far, there has not been any study on the energy overhead of the secure processor or optimization techniques to reduce the energy consumption.

13.2.3 Additional Features

The AEGIS architecture in this thesis focuses on uniprocessor systems and the problem of trusted remote interactions. However, the secure processor can be applied to more application scenarios if it is extended with more security features. Also, it is an interesting research challenge to improve the attestation mechanisms in the current design.

First, modern high-end servers often use symmetric multi-processors (SMPs) that share the same memory. There are two challenges that need to be addressed for AEGIS to be applied for such high performance server farms. In terms of security, AEGIS' off-chip protection mechanisms need to be augmented to handle multiple processors that access shared memory resources. Further studies are required also for the usage model. Unlike a uniprocessor case, in large server farms, multiple processors need to share secrets to authenticate each other. Also, application programs can also migrate from processor to processor requiring remote parties to interact with a group of processors rather than a single processor.

Second, securing local I/O channels pose new research challenges. For example, imagine a public terminal that can be accessed by a large number of people. AEGIS can provide secure environments for application programs to execute even under physical attacks. To trust computation on such a system, however, local users must also be able to trust I/O channels such as keyboards, mouse, and display.

Finally, there are many interesting research opportunities in outbound attestation mechanisms. The current AEGIS design simply uses a cryptographic hash to uniquely identify each application. However, this scheme implies that remote parties must be able to trust the whole application and know which applications can be trusted. Ideally, remote parties should be able to authenticate only the part of an application that they are interacting with using fine-grained attestation mechanisms. Also, attestation of an application's security property showing that it is trustworthy rather than simply telling the identity will be

more desirable.

13.2.4 Application Studies

This thesis has briefly discussed application scenarios such as distributed computation and DRM. However, such discussions are limited because details of each application are not worked out. Further studies on real implementations will provide much more insight into the potential and limitations of the current AEGIS design. There have been a number of applications implemented for the IBM 4758 secure co-processor. AEGIS can replace the secure co-processor in those applications as it provides the required security features. However, what kinds of new applications AEGIS' additional computation power enables needs to be further investigated.

For AEGIS, or trusted computing platforms in general, to be widely deployed, a key challenge to be addressed is an infrastructure to authenticate the processor hardware and software on it. The attestation mechanism in AEGIS allows remote parties to know which hardware and software they are interacting with. However, it is up to the remote parties to decide whether the platform is trustworthy or not. For that purpose, in most scenarios, a trusted third party is required. Building such an infrastructure is a challenging problem especially when hardware and software once trusted may be later found buggy.

Finally, trusted computing platforms may result in side effects that need to be investigated. First, the attestation capability of AEGIS poses a privacy issue. For example, remote parties may be able to relate various activities an individual engages in on the Internet by looking at the identity of the machine. Finally, the attestation may also allow remote hackers to easily identify the vulnerabilities to exploit by giving away information about the exact software running on the system.

Bibliography

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

[2] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. ARM white paper, July 2004.

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

[4] Ross Anderson and Markus Kuhn. Tamper resistance - a cautionary note. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11, November 1996.

[5] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop on Security Protocols, LNCS*, 1997.

[6] 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.

[7] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptographic co-processor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, 2004.

[8] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 178–183, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO '96*, volume 1109 of *LNCS*. Springer-Verlag, 1996.

[11] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

[12] Duane S. Boning and James E. Chung. Statistical metrology: Understanding spatial variation in semiconductor manufacturing. In *Proceedings of SPIE 1996 Symposium on Microelectronic Manufacturing*, 1996.

[13] Keith A. Bowman, Steven G. Duvall, and James D. Meindl. Impact of die-to-die and within die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Journal of Solid-State Circuits*, 37(2):183–190, February 2002.

[14] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[15] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[16] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, Boston, MA, 2001.

[17] Joris Claessens, Bart Preneel, and Joos 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, February 2003.

[18] Dwaine Clarke. *Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data*. PhD thesis, Massachusetts Institute of Technology, May 2005.

[19] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity

checking. In *Advances in Cryptology - Asiacrypt 2003 Proceedings*, volume 2894 of *LNCS*. Springer-Verlag, 2003.

[20] Dwaine Clarke, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas. Offline integrity checking of untrusted storage. In *Technical Report MIT-LCS-TR-871*, November 2002.

[21] Dwaine Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, 2005.

[22] Dwaine Clarke, G. Edward Suh, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Checking the Integrity of Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System. In *MIT CSAIL CSG Technical Memo 470*, July 2004.

[23] C. Clavier, J. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In *CHES*, volume 1965 of *LNCS*, pages 252–263. Springer-Verlag, August 2000.

[24] J. Coron, P. Kocher, and D. Naccache. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES*, volume 1717 of *LNCS*, pages 292–302. Springer-Verlag, August 1999.

[25] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *the* 10^{th} *USENIX Security Symposium*, 2001.

[26] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[27] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[28] Robert Davies. Hardware random number generators. In 15^{th} *Australian Statistics Conference*, July 2000.

[29] Solar Designer. Non-executable user stack.
`http://www.penwall.com/linux/`.

[30] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[31] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: how to generate strong keys from biometrics and other noisy data. In *Advances in Cryptology - Eurocrypt 2004*, 2004.

[32] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), September 2001.

[33] The Embedded Microprocessor Benchmark Consortium (EEMBC).
`http://www.eembc.org/`.

[34] John S. Erickson. Fair use, DRM, and trusted computing. *Commun. ACM*, 46(4):34–39, 2003.

[35] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security*, 2003.

[36] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261, London, UK, 2001. Springer-Verlag.

[37] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Priciples (SOSP 2003)*, 2003.

[38] Blaise Gassend. Physical Random Functions. Master's thesis, Massachusetts Institute of Technology, January 2003.

[39] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled Physical Random Functions . In *Proceedings of* 18^{th} *Annual Computer Security Applications Conference*, December 2002.

[40] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the Computer and Communication Security Conference*, November 2002.

[41] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Delay-based circuit authentication and applications. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, March 2003. Extended version to appear in Concurrency and Computation: Practice and Experience.

[42] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, February 2003.

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

[44] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. `http://www.trustedcomputinggroup.com/home`, 2004.

[45] Trusted Computing Group. Open standards for integrity-based network access control. TCG white paper, 2005.

[46] HardwareCentral. CPU-based security: The NX bit, May 2004.

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

[48] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dyanmic self-checking techniques for improved tamper resistance. In *Proceedings of the* 1^{st} *ACM Workshop on Digital Rights Management (DRM 2001)*, volume 2320 of *LNCS*, pages 141–159. Springer-Verlag, 2002.

[49] A. Iliev and S. W. Smith. Prototyping and armored data vault: Rights management for big brother's computer. In *Privacy Enhancing Technologies (PET 2002)*, volume 2482 of *LNCS*, pages 144–159. Springer-Verlag, 2003.

[50] A. Iliev and S. W. Smith. Protecting client privacy with trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, March-April 2005.

[51] Intel. Execute disable bit functionality.

[52] Intel. LaGrande technology architecture overview, September 2003.

[53] N. Itoi. Secure coprocessor integration with kerberos v5. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, pages 113–128, 2000.

[54] S. Jiang. WebALPS implementation and performance analysis: Using trusted co-servers to enhance privacy and security of web interactions. Master's thesis, Dartmouth College, June 2001.

[55] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[56] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the* 3^{rd} *International Workshop on Automatic Debugging*, 1997.

[57] Benjamin Jun and Paul Kocher. The Intel Random Number Generator. Cryptography Research Inc. white paper, April 1999.

[58] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *Second ACM Conference on Embedded Networked Sensor Systems (SensSys 2004)*, November 2004.

[59] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 295–310, August 2003.

[60] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, San Francisco, California, August 2002.

[61] Paul Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS and other systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996.

[62] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[63] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41th Conference on Design Automation (DAC'04)*, pages 753–760, 2004.

[64] O. Kommerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Procedings of USENIX Workshop on Smartcard Technology (Smartcard'99)*, pages 9–20, May 1999.

[65] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC: Keyed-Hashing for Message Authentication, February 1997. Status: INFORMATIONAL.

[66] Henry Kuo and Ingrid 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.

[67] J-W. Lee, D. Lim, B. Gassend, E. G. Suh, M. van Dijk, and S. Devadas. A Technique to Build a Secret Key in Integrated Circuits with Identification and Authentication Applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, June 2004.

[68] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the 2003 International Conference on Security in Pervasive Computing*, 2003.

[69] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 2–13, 2005.

[70] Philip Levis, Sam Maddena, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[71] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[72] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[73] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark 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.

[74] Daihyun Lim. Extracting Secret Keys from Integrated Circuits. Master's thesis, Massachusetts Institute of Technology, May 2004.

[75] Helger Lipmaa, Phillip Rogaway, and David Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000.

[76] K. Lofstrom, W. R. Daasch, and D. Taylor. IC Identification Circuit Using Device Mismatch. In *Proceedings of ISSCC 2000*, pages 372–373, February 2000.

[77] M. Lorch, J. Basney, and D. Kafura. A hardware-secured credential repository for grid PKIs. In *Proceedings of the* 4^{th} *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2004.

[78] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[79] S. Mangard. Hardware countermeatures against DPA - a statistical analysis of their effectiveness. In *CT-RSA*, volume 2964 of *LNCS*, pages 222–235. Springer-Verlag, February 2004.

[80] J. Marchesini, S. W. Smith, O. Wild, and A. Barsamian. Open-source applications of TCPA hardware. In *Proceedings of the* 20^{th} *Annual Computer Security Application Conference (ACSAC)*, December 2004.

[81] J. Marchesini, S. W. Smith, O. Wild, and R. Macdonald. Bear: An open-source virtual secure coprocessor based on TCPA. Technical Report TR2003-471, Dartmouth College Department of Computer Science, August 2003.

[82] J. Marchesini, S. W. Smith, O. Wild, and R. Macdonald. Experimentin with TPCA/TCG hardware, or: How I learned to stop worrying and love the bear. Technical Report TR2003-476, Dartmouth College Department of Computer Science, December 2003.

[83] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

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

[85] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining smart-card security under the threat of power analysis attakcs. *IEEE Transactions on Computers*, 51(5):541–552, 2002.

[86] Microsoft. Next-generation secure computing base. `http://www.microsoft.com/resources/ngscb/defaul.mspx`.

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

[88] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In 22^{nd} *ACM Symposium on Theory of Computing*, pages 213–223, 1990.

[89] Sani R. Nassif. Modeling and forecasting of manufacturing variations. In *Proceedings of ASP-DAC 2001, Asia and South Pacific Design Automation Conference 2001.* ACM, 2001.

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

[91] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the* 29^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[92] Tim Newsham. Format string attacks. Guardent, Inc., September 2000. `http://www.securityfocus.com/guest/3342`.

[93] NIST. FIPS PUB 180-2: Secure Hash Standard, August 2002.

[94] Charles W. O'Donnell, G. Edward Suh, and Srinivas Devadas. PUF-Based Random Number Generation. In *MIT CSAIL CSG Technical Memo 481 (http://csg.csail.mit.edu/pubs/memos/Memo-481/Memo-481.pdf)*, November 2004.

[95] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[96] OPENCORES.ORG. OpenRISC 1000 Project. `http://www.opencores.org/projects.cgi/web/or1k`.

[97] E. Palmer. An introduction to Citadel - a secure crypto coprocessor for workstations. Technical Report RC18373, IBM T. J. Watson Research Center, 1992.

[98] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical One-Way Functions. *Science*, 297:2026–2030, 2002.

[99] PaX Team. Non executable data pages. `http://pageexec.virtualave.net/verb`pageexec.txt .

[100] A. Perrig, S. W. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. *eJETA.org: The Electronic Journal for E-Commerce Tools and Applications*, 1, January 2002.

[101] Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Commun. ACM*, 47(6):53–57, 2004.

[102] C.S. Petrie and J.A. Connelly. A noise-based ic random number generator for applications in cryptography. *IEEE TCAS II*, 46(1):56–62, January 2000.

[103] N. Pramstaller, F. Gurkaynak, S. Hane, H. Kaeslin, N. Felber, and W. Fichtner. Towards an AES crypto-chip resistant to differential power analysis. In *ESSCIRC*, pages 307–310, September 2004.

[104] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Proceedings of Smart Card Programming and Security (E-smart 2001), LNCS 2140*, pages 200–210, September 2001.

[105] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley and Sons, January 2004.

[106] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *Transactions on Embedded Computing Systems*, 3(3):461–491, 2004.

[107] P. S. Ravikanth. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[108] M. Renaudin, F. Bouesse, Ph. Proust, J. P. Tual, L. Sourgen, and F. Germain. High security smartcards. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10228, Washington, DC, USA, 2004. IEEE Computer Society.

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

[110] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, April 1992. Status: INFORMATIONAL.

[111] 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.

[112] Ronald L. Rivest. The RC5 Encryption Algorithm, from Dr. Dobb's Journal, January, 1995. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996*. 1996.

[113] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. WCB McGraw-Hill, 1999.

[114] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the* 11^{th} *Annual Network and Distributed System Security Symposium*, 2004.

[115] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[116] Ishan Sachdev. Development of a programming model for the AEGIS secure processor. Master's thesis, Massachusetts Institute of Technology, May 2005.

[117] Ravi Sandhu and Xinwen Zhang. Peer-to-peer access control architecture using trusted computing technology. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 147–158, New York, NY, USA, 2005. ACM Press.

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

[119] Patrick R. Schaumont, Henry Kuo, and Ingrid M. Verbauwhede. Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor. In *Design Automation Conference 2002*, June 2002.

[120] Stuart E. Schechter and Michael D. Smith. Trusted computing, peer-to-peer distribution, and the economics of pirated entertainment. In *Proceedings of the Second Workshop on Economics and Information Security*, May 2003.

[121] Scut. Exploiting format string vulnerabilities. TESO Security Group, September 2001. `http://www.team-teso.net/articles/verb`formatstring .

[122] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: SoftWare-base ATTestation for Embedded Devices. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 272–282, May 2004.

[123] A. Shamir. Protecting smart cards from passive power analysis with detached power supplies. In *CHES*, volume 1965 of *LNCS*, pages 71–77. Springer-Verlag, August 2000.

[124] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 295–310, August 2003.

[125] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, 2001.

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

[127] Weidong Shi, Hsien-Hsin Lee, Chenghuai Lu, and Mrinmoy Ghosh. Towards the Issues in Architectural Support for Protection of Software Execution. In *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[128] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, and Chenghuai Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT-04)*, pages 123–134, September 2004.

[129] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 14–24, 2005.

[130] S. W. Smith, R. Perez, S. H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *Proceedings of the* 22^{nd} *National Information Systems Security Conference*, October 1999.

[131] 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.

[132] Sean Smith. *Trusted Computing Platforms: Design and Applicaitons*. Springer Science+Business Media, Inc., 2005.

[133] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, 2001.

[134] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 339–350, December 2003.

[135] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Hardware mechanisms for memory integrity checking. In *Technical Report MIT-LCS-TR-872*, November 2002.

[136] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing.

In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, June 2003.

[137] G. Edward Suh, Jaewook Lee, David X. Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the* 11^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[138] K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. A side-channel leakage free coprocessor ic in 0.18um CMOS for embedded aes-based cryptographic and biometric processing. In *DAC '05: Proceedings of the 42nd Annual Conference on Design Automation*, pages 222–227, New York, NY, USA, 2005. ACM Press.

[139] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *DATE*, pages 246–251, feb 2004.

[140] Kris Tiri and Ingrid Verbauwhede. Simulation models for side-channel information leaks. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 228–233, New York, NY, USA, 2005. ACM Press.

[141] Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2), April-June 2005.

[142] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`.

[143] Xiaoyun Wang. Collisions for some hash functions MD4, MD5, HAVAL-128, RIPEMD. In *CRYPTO '04*, 2004.

[144] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *CRYPTO '05*, 2005.

[145] S. H. Weingart. Physical security of the uABYSS system. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[146] S. White, S. H. Weingart, W. Arnold, and E. R. Palmer. Introduction to the Citadel architecture: Security in physically exposed environments. Technical Report RC16672, IBM T. J. Watson Research Center, 1991.

[147] S. R. White and L. D. Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[148] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the* 10^{th} *Annual Network and Distributed System Security Symposium*, 2003.

[149] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.

[150] Glenn Wurster, Paul C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[151] Haidong Xia, Jayashree Kanchana, and Jose Carlos Brustoloni. Using secure coprocessors to protect access to enterprise networks. In *Proceedings of the Networking 2005 Conference*, volume 3462 of *LNCS*, pages 154–165. Springer-Verlag, 2005.

[152] Jun Xu, Zbigniew Kalbarczjk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Proceedings of the* 2^{nd} *Workshop on Evaluating and Architecting System dependability (EASY)*, 2002.

[153] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 351–360, Dec 2003.

[154] Jun Yang, Lan Gao, and Youtao Zhang. Improving memory encryption performance in secure processor. *IEEE Transactions on Computers*, 54(5), May 2005.

[155] B. S. Yee. A sanctuary for mobile agents. In *Secure Internet Programming*, volume 1603 of *LNCS*, pages 261–274. Springer-Verlag, 1999.

[156] B. S. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the* 1^{st} *USENIX Electronic Commerce Workshop*, pages 155–170, 1995.

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

[158] Youtao Zhang, Lan Gao, Jun Yang, Xiangyu Zhang, and Rajiv Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 352–362, february 2005.

[159] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 72–84, New York, NY, USA, 2004. ACM Press.