Information Security Technical Report (2005) 10, 63–73

Information Security Technical Report

AEGIS: A single-chip secure processor

G. Edward Suh*, Charles W. O'Donnell, Srinivas Devadas

Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Abstract This article presents the AEGIS secure processor architecture, which enables new applications by ensuring private and authentic program execution even in the face of physical attack. Our architecture uses two new primitives to achieve physical security. First, we describe Physical Random Functions which reliably protect and share secrets in a manner that is cheaper and more secure than existing solutions based on non-volatile memory. Second, off-chip memory protection mechanisms ensure the integrity and the privacy of off-chip memory. Our processor, with its new protection mechanisms, has been implemented on an FPGA, and is fully functional. We briefly assess the cost of the security mechanisms in our processor and show that it is reasonable.

Introduction

As computing devices become ubiquitous and highly interconnected, two contradictory trends are appearing. On the one hand, the cost of security breaches is increasing as we place more sensitive information and responsibilities on the devices. On the other hand, computing elements are becoming small, disseminated, unsupervised, and physically exposed. Unfortunately, conventional software protection mechanisms do not address physical threats, presenting a significant vulnerability in future computing applications.

For example, in Digital Rights Management (DRM), the owner of a computer system is motivated to alter the system behavior in order to make illegal copies of protected digital content. Similarly, mobile agent applications (Claessens et al., 2003) require that sensitive electronic transactions be performed on untrusted hosts. The hosts may be under the control of an adversary who is financially motivated to compromise a mobile agent. In such scenarios, software-only protections can easily be bypassed because attackers have full control of the operating systems and applications such as DRM players or mobile agents.

To address these emerging threats, there have been significant efforts to build a secure computing platform that enables users to authenticate the platform and its software. The Trusted Platform

* Corresponding author.
E-mail addresses: suh@mit.edu (G.E. Suh), cwo@mit.edu (C.W. O'Donnell), devadas@mit.edu (S. Devadas).

1363-4127/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.istr.2005.05.002

Module (TPM) from Trusted Computing Group (TCG) (T.C. Group, 2004), Next Generation Secure Computing Base (NGSCB) from Microsoft, and TrustZone from ARM (Alves and Felton, 2004) all incorporate authentication mechanisms. If a DRM mechanism is implemented on these secure platforms, a content provider can encrypt its protected content just for a specific device executing trusted DRM software.

While the above systems can detect attacks that tamper with the operating systems or user applications, they cannot protect against physical attacks that tap or probe chips or buses in the system.

In this article, we introduce a single-chip secure processor called AEGIS. In addition to mechanisms to authenticate the platform and software, our processor incorporates mechanisms to protect the integrity and privacy of applications from physical attacks as well as software attacks. Therefore, physically secure systems can be built using this processor. Two key primitives, namely, Physical Random Functions and off-chip memory protection enable the physical security of our system. These primitives can also be easily applied to other secure computing systems to enhance their security.

The rest of the article is organized as follows. In the following section, we compare our secure computing model with other approaches. Then, we describe Physical Random Functions, which is followed by an overview of the AEGIS architecture with its memory protection mechanisms. Further, resource and performance costs of our protection mechanisms are discussed, and we conclude in the last section.

Secure computing models

A secure computing platform needs to contain a secret key so that remote parties can authenticate the platform. Also, the platform must protect the integrity and the privacy of applications during execution. In this section, we compare possible approaches to build a secure computing system based on the implementation of these authentication and protection mechanisms.

Tamper-proof packages

The conventional approach to building physically secure systems (Smith and Weingart, 1999; Yee, 1994) 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 (Anderson, 2001) and active intrusion detection circuitry must be continuously battery powered even when the device is off. In addition, these devices are not flexible, e.g., their memory or I/O subsystems cannot be upgraded easily. As a result, this type of tamper-proof package is not appropriate for pervasive computing devices that need to be cheap and flexible.

Multi-chip approach

Recent efforts to build secure computing platforms such as TPM from TCG (T.C. Group, 2004) and Microsoft NGSCB, implement security functionality 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 those used in smartcards, this chip is relatively simple and contains an embedded secret key which can be used to authenticate the platform. Even though these platforms use multiple chips when implementing the security features, they do not use expensive tamper-proof packages. They simply assume that physical attacks are difficult to carry out.

Some advantages of implementing security features in a separate chip are clear. 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. Unfortunately, communication between the main processor and the adjoining security chip (e.g. the TPM) can be easily tampered with. Similarly, communication between the main processor and main memory suffers from the same flaw. Therefore, this approach is *not* secure against physical attacks.

AEGIS approach

Fig. 1 illustrates the model that AEGIS is built upon. Put briefly, we only trust a single-chip secure processor that includes all security features and secret keys. The processor is protected from physical attacks whenever it is powered on, so that its internal state cannot be tampered with or observed directly by physical means. On the other

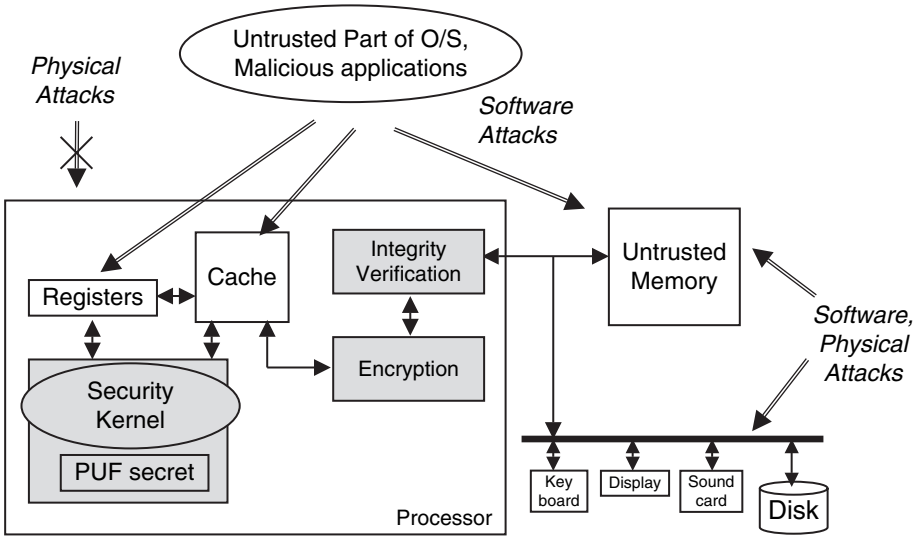

Figure 1 AEGIS secure computing model.

hand, all components outside the processor chip, including external memory and peripherals, are assumed to be insecure. They may be observed and tampered with by an adversary at will.

Trusting a single processor enables us to build a cheap and secure computing platform. Because only one chip needs to be protected when it is powered on, there is no need for the expensive battery-backed tamper-proof package. In fact, even without additional protection mechanisms, opening up a chip and tampering with on-chip memory while the processor is running is prohibitively expensive for most low-budget attackers. Active intrusion detection can also be used if necessary. However, unlike the tamper-proof package, the protection mechanism only needs to be active while the power is on. Finally, unlike the multi-chip approach, physical attacks on external buses cannot compromise the system security.

On the other hand, only trusting the processor chip brings new challenges. First, the secret key must be embedded in the main processor in a way that is secure without significantly increasing the cost of the processor. Unfortunately, existing non-volatile memory such as EEPROM is neither secure nor cheap to implement in the main processor. We address this problem using Physical Random Functions in the next section.

Second, off-chip memory is still vulnerable to physical attacks. The processor must check values read from memory to ensure the integrity of execution state, and must encrypt private data values stored in off-chip memory to ensure privacy. We briefly describe the off-chip memory protection mechanisms in the section on 'Processor architecture'.

We will also assume the processor has a hardware random number generator (Jun and Kocher, 1999; O'Donnell et al., 2004; Petrie and Connelly, 2000) to defeat possible replay attacks on communication. In this article, we do not consider the attacks using side-channels such as memory access patterns or power supply voltage (Kocher et al., 1999). Mechanisms that are commonly used in today's smartcards can prevent side-channel attacks.

Physical Random Functions

As noted in our security model, an AEGIS processor chip must contain a secret so that users can authenticate the processor that they are interacting with. One simple solution is to have non-volatile memory such as EEPROM or fuses on-chip. With this, the manufacturer programs the non-volatile memory with a chosen secret such as a private key, and introduces the corresponding public key to the users.

Unfortunately, digital keys stored in non-volatile memory are vulnerable to physical attacks (Anderson, 2001). Motivated attackers can remove the package without destroying the secret, and extract the digital secret from the chip.

Storing a digital key in on-chip non-volatile memory also has additional problems even for applications where physical security is a low concern. On-chip EEPROMs require more complex fabrication processes compared to standard digital logic. This would cause secure processors to be more expensive and difficult to manufacture. Fuses do not require more manufacturing steps, but contain a single permanent key. Finally, both

EEPROM and fuse storage need to be initially programmed and tested by a *trusted party* at a *secure location* before use.

A Physical Random Function (also called a Physical Unclonable Function or PUF) is a function that maps a set of challenges to a set of responses 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. While PUFs can be implemented with various physical systems, we use silicon PUFs (SPUFs) that are based on the hidden timing and delay information of integrated circuits.

PUFs provide significantly higher physical security by extracting secrets from complex physical systems rather than storing them in non-volatile memory. A processor can dynamically generate many PUF secrets from the unique delay characteristics of wires and transistors. To attack this, an adversary must mount an invasive attack *while the processor is running and using the secret*, a significantly harder proposition. Further, an attacker who attempts to measure the hidden timing information within the PUF must do so without changing any PUF wire delays.

This is extremely hard because fabricated oxide/metal layers need to be removed in order to measure transistor delays or to view the secret.

Another advantage of silicon PUFs is that they do not require any special manufacturing process or special programming and testing steps. Therefore, PUFs are cheap.

In this section, we now describe a candidate implementation of a silicon PUF, and how the PUF can be used to express a secret in a secure processor.

Silicon PUFs

Even when using identical layout masks, variations in a manufacturing process can 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.

Fig. 2 illustrates the SPUF delay circuit used in this article. While this particular design is used to demonstrate the technology, we note that many other designs are possible. The 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, which either switch or pass through the top and bottom signals. To evaluate the output for a particular input, a rising signal is given to both top and bottom paths at the same time, the two signals race through the two delay paths, and the latch at the end measures which signal is faster. The output is *one* if the top signal is faster, and *zero* if the bottom signal is faster.

This PUF delay circuit with 64 stages has been fabricated and tested in TSMC's 0.18 μm, single-poly, six-level metal process (Lee et al., 2004). 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). Thanks to the relative delay measurements, the PUF is robust against environmental variations. For realistic changes in temperature from 20 to 70 °C and regulated voltage changes of ±2%, the output noise is 4.8% and 3.7%, respectively. Even when unrealistically increasing the temperature by 100 °C and varying the voltage by 33%, the PUF output noise still remains below 9%. This is significantly less than the inter-chip variation of 23%, allowing for the identification and authentication of individual chips. (We note that

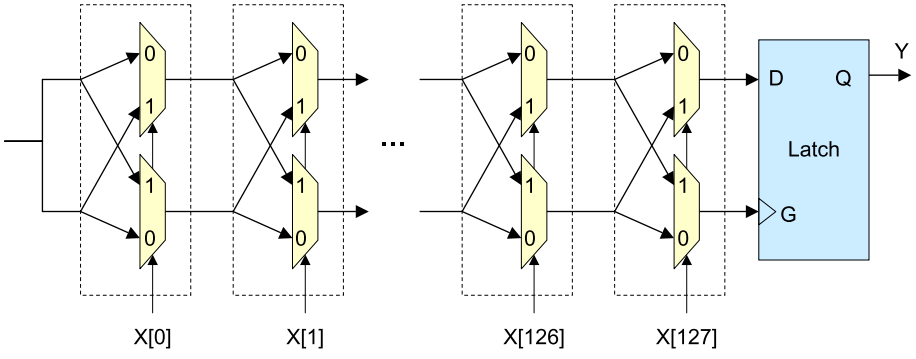

Figure 2 A silicon PUF delay circuit.

an ideally symmetric layout of the circuit in Fig. 2 would likely increase inter-chip variation to 50%.)

In the following discussion, we assume that a PUF circuit gets a 128-bit challenge C as an input and produces a k-bit response $R = \mathrm{PUF}(C)$. There are two ways to construct a k-bit response from the 1-bit output of the PUF delay circuit. First, one circuit can be used k times with different inputs.

Second, the single-output PUF circuit itself can be duplicated multiple times to obtain k bits with a single evaluation. As the PUF circuit has only a few hundred gates, the duplication incurs a modest increase in the overall processor gate count.

Preventing model building

Because our PUF circuit is rather simple, attackers can try to construct a precise timing model for a given PUF from many challenge–response pairs. In order to prevent model-building attacks, we hash (obfuscate) the output of the delay circuit to generate the k-bit response. Therefore, to learn the actual circuit outputs, attackers need to invert a one-way hash function, which is computationally intractable.

Expressing chosen secrets

PUF responses can be considered as secrets because they are randomly determined by manufacturing variations and are difficult to predict without access to the PUF. In this section, we discuss how a chosen private key or a symmetric key can be expressed with the PUF.

First, a manufacturer or a user who wants to put a chosen key into a device needs to obtain a secret response from the PUF. For this purpose, the processor provides the `getResponse` primitive. The input to this primitive is *PreC*, called a ''pre-challenge''. The PUF computes the challenge C by hashing *PreC* using a cryptographic one-way function such as SHA-1, and outputs the PUF response R for that challenge. Note that attackers cannot compute the pre-challenge for a specific challenge unless they can invert the one-way hash function.

In the absence of an eavesdropper, the user can use a randomly chosen *PreC*, and obtain a response in plaintext. This user can easily compute the challenge from the pre-challenge because the hash function is public. Here, *PreC* should be kept secret so that attackers cannot generate the same challenge–response pair (CRP).

If eavesdropping is permitted, the `getResponse` primitive can be modified to use private/public key cryptography. The user inputs his public key *PK* instead of *PreC*. The primitive uses *PK* as a PUF challenge, obtains the response, and encrypts the response with the user's public key *PK* before outputting it.

Even though an eavesdropper can see the encrypted response, only the user can decrypt the response using his private key.

Knowing a secret CRP (C, R), the manufacturer or the user can have the PUF express a chosen secret key SK by providing the challenge C and the key after it has been encrypted using the response ($E_R\{SK\}$). The processor obtains the response R from the PUF using the challenge C, and decrypts $E_R\{SK\}$ to obtain SK. For example, the manufacturer can choose a private key SK, and embed it in a device by storing C and $E_R\{SK\}$ in the device's non-volatile memory.

Note that both C and $E_R\{SK\}$ are public, and can be stored in any non-volatile memory such as flash memory or even hard-disk. Also, assuming that the response is random, a simple XOR can be used to encrypt the chosen secret key ($E_R\{SK\} = SK \oplus R$).

Reliable secret generation

Due to environmental noise, PUF responses are likely to be *slightly* different on each evaluation, even for the exact same challenge C. However, cryptographic primitives require that every bit of a key stays constant. Therefore, we need to securely add error correction capabilities to our PUF so that the same secret can be generated on every execution.

Fig. 3 summarizes the extended PUF bootstrapping and secret generation including error correction. For `getResponse`, in addition to the k-bit response, the processor also computes a BCH Code syndrome for the PUF delay circuit output. The BCH code is a popular error correcting code that is widely used for binary data. Now the primitive outputs the response R and the syndrome S, which are k-bit and b-bit values, respectively. A syndrome is redundant information that allows a BCH decoder to correct errors on the PUF delay circuit output. Other non-syndrome error correction schemes are also possible.

The processor generates a secret using three inputs: the challenge C, the syndrome S, and the encrypted key $E_R\{SK\}$. 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 previous `getResponse` primitive. Finally, the response is used to obtain a chosen secret SK.

Obviously, the syndrome reveals information about the PUF delay circuit output, which may

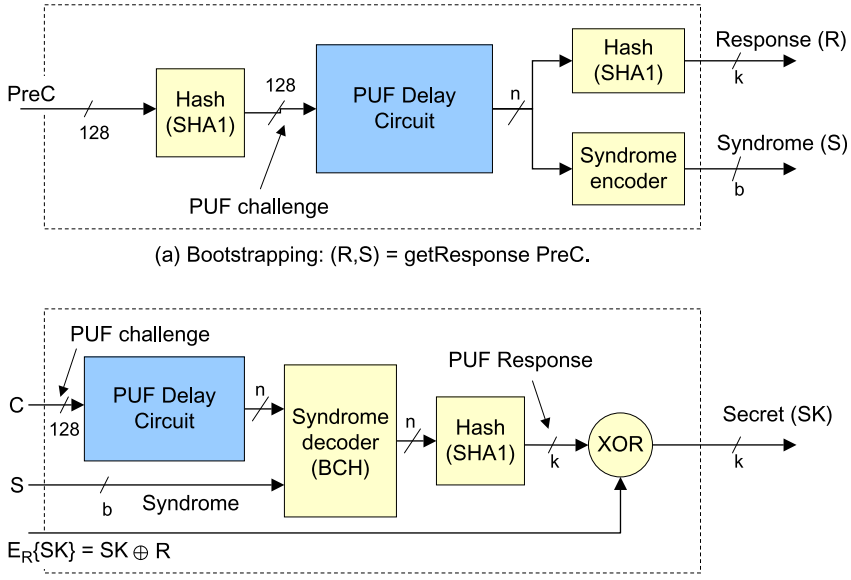

Figure 3 Reliable secret generation using PUFs.

be a security hazard. However, enough bits remain secret even after error correction. For example, the BCH (255,63,61) code can correct up to 30 errors out of 255 bits, and uses a 192-bit syndrome. Even if we assume that the syndrome reveals 192 bits of the PUF output, 63 bits remain secret. For some applications 63-bit secrets are enough. For higher security the PUF can be used twice to obtain 126-bit secrets.

Given that the PUF has a bit error rate of 4.8% under realistic environmental conditions, this error correcting capability also provides very high reliability. The probability for a PUF to have more than 30 errors out of 255 bits is 2.4×10^{-6}.

Security analysis

In this section, we discuss the most plausible attacks and show how our PUF design can defeat each of them.

- *Brute-force attacks*: attackers with access to the secure processor can try to completely characterize the PUF by obtaining all possible CRPs. However, this is infeasible because there are an exponentially large number of challenges. For example, given 128-bit challenges, the attacker must obtain 2^{128} CRPs.
- *Model building*: to avoid exhaustive enumeration, attackers may try to construct a precise timing model of a given PUF and learn the parameters from many CRPs. However, model building is not possible because the PUF primitives never directly return the PUF delay circuit output (see the section on 'Preventing model building').
- *Duplication*: to bypass the PUF primitives, attackers may fabricate the same PUF delay circuit that can be directly accessed. However, the counterfeit PUF is extremely unlikely to have the same outputs as the original PUF since 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.
- *Invasive attacks*: attackers can open up the package of the secure processor and attempt to read out the secret when the processor is running or attempt to measure the PUF delays when the processor is powered off. Probing the delays with sufficient precision (the resolution of the latch) is extremely 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, and destroying the secret. We note that it is possible to architect the processor in such a way that only part of the secret is present on the chip in digital form at any given time.

Processor architecture

The AEGIS processor is able to shield against software and physical attacks by protecting a

program before it is executed, protecting it during execution, and protecting it during processor mode transitions. When an application is initially run, the processor uses a program hashing technique to verify that the program was not corrupted while it was held in unprotected storage. During execution the processor uses integrity verification, memory encryption, and access permission checks to guarantee security under four different secure execution modes. Finally, the transition between secure execution modes is carefully structured and monitored.

Typical processors contain user and supervisor modes which control access to special functions such as virtual memory mechanisms. Within user and supervisor modes, AEGIS additionally provides a Standard (STD) mode which has no additional security measures, a Tamper-Evident (TE) mode which ensures the integrity of program state, a Private Tamper-Resistant (PTR) mode which additionally ensures privacy, and a Suspended Secure Processing (SSP) mode.

SSP mode allows an application which is running under TE or PTR mode to safely execute insecure regions of the program. This reduces the need for a large trusted amount of code and allows drivers and third party libraries to be run safely.

Here we summarize the protection capabilities of each of these modes. Note that TE mode has all the capabilities of STD mode, and PTR mode has all the capabilities of TE and STD mode.

- *STD and SSP modes:*
  - R/W access to unprotected memory
  - Standard code can be executed (in an unprotected fashion)
  - Only can call one of the security instructions which (re-)enters TE or PTR mode
  - STD mode can manage VM (SSP cannot)
- *TE mode:*
  - R/W access to verified memory
  - Access to most security instructions
- *PTR mode:*
  - R/W access to private memory
  - Access to PUF instructions

Authentication

Our processor allows users to authenticate the processor and software. For this purpose, each processor has a unique secret key securely embedded using a PUF (see the section on 'Physical Random Functions'). For example, each processor can have its own private key whose corresponding public key is known to users. Then, the processor can sign a message with the private key to authenticate itself to the users.

In order to support software authentication, our processor combines program hashes with a digital signature as in Microsoft NGSCB or TPM. When the operating system starts and enters a secure execution mode (TE or PTR), our processor computes the cryptographic hash of the trusted part of the operating system, which is called the security kernel. This program hash is stored in a secure on-chip register, and is always included in the signature. Therefore, when users verify a signature from the processor, they know that the message is from a particular security kernel running on a particular processor.

The security kernel provides the same authentication mechanism to user applications by computing their hashes when user applications enter a secure computing mode. While we described an authentication scheme using private/public keys, we note that it is also possible to use different protocols optimized for PUFs (Suh et al., 2005).

Memory protection

The TE and PTR security modes must guarantee the integrity and/or privacy of instructions and data in memory under both software and physical attacks. To defend against software attacks the processor performs additional access permission checks within the Memory Management Unit (MMU). To defend against physical attacks, Integrity Verification (IV) and Memory Encryption (ME) techniques are used. These defenses are not enabled at startup, but instead are initiated when a supervisor program switches into TE or PTR mode.

The processor separates physical memory space into regions designated ''IV protected'' or ''ME protected'' (allowing overlap) whose boundaries are specified upon entrance into TE or PTR mode. The processor has an integrity verification mechanism which detects any tampering that changes the content of the IV regions, and an encryption mechanism which guarantees the privacy of the ME regions. For efficiency reasons, the IV and ME regions are further divided into ''static'' and ''dynamic'' sections which correspond to read-only data (such as application instructions) and read-write data (such as heap and stack variables).

Memory encryption is handled by encrypting and decrypting all off-chip data transfers in the ME regions using a one-time-pad (OTP) encryption scheme (Suh et al., 2003). Fig. 4 shows how an evicted cache block is XOR'ed with an AES encryption of its memory address, a time stamp, and

some constant bit vector *V*. The time stamp is small and is also stored in memory. During a cache block fetch, decryption latency is hidden since the time stamp can be fetched and used to recompute a pad while the larger cache block is still being loaded from memory. For the static ME region, the pad computation can start even earlier as no time stamp is required.

The processor protects the dynamic IV region by creating a hash tree for the region, and saving the root hash node on-chip (Gassend et al., 2003; Fig. 5). In this way, any tampering of off-chip memory will be reflected by a root hash that does not match the saved one. The same hash tree also protects the encryption time stamps for the dynamic ME region that overlaps with the dynamic IV region. Static IV regions are protected differently. Because the static region is read-only, replay attacks (substituting the new value with an old value of the same address) are not a concern. In this case, cryptographic message authentication codes (MACs) are taken over the address and data values of the static IV region, and stored in a reserved portion of the unprotected memory.

To reduce verification latency, the IV mechanism runs in the background, only stalling main execution to catch up when a security instruction must be executed, or when a store occurs to non-private memory while in PTR mode. This guarantees that all security instructions have been verified and protects private data from leaking into non-private memory.

Finally, access permission checks guarantee that processes operating within either SSP or STD mode cannot tamper with any of the IV or ME protected memory regions.

Multitasking

Secure multitasking on the AEGIS processor can be ensured with the help of a trusted security kernel handling things such as Virtual Memory Management (VMM). In this model, a trusted security kernel is started after boot-up and transitions the processor into TE or PTR mode before starting the VMM system.

Both the security kernel and user applications can use four protected regions in virtual memory space which provide different levels of security.

1. Read-only (static) verified memory
2. Read-write (dynamic) verified memory
3. Read-only (static) private memory
4. Read-write (dynamic) private memory

Fig. 6 shows how the AEGIS processor separates physical memory to allow a security kernel to safely map virtual addresses.

We point out here that only single dynamic IV/ME regions are required since a security kernel can share this space with user processes. However, the processor separately provides user-level and supervisor-level static IV/ME regions since these regions depend upon specific decryption keys

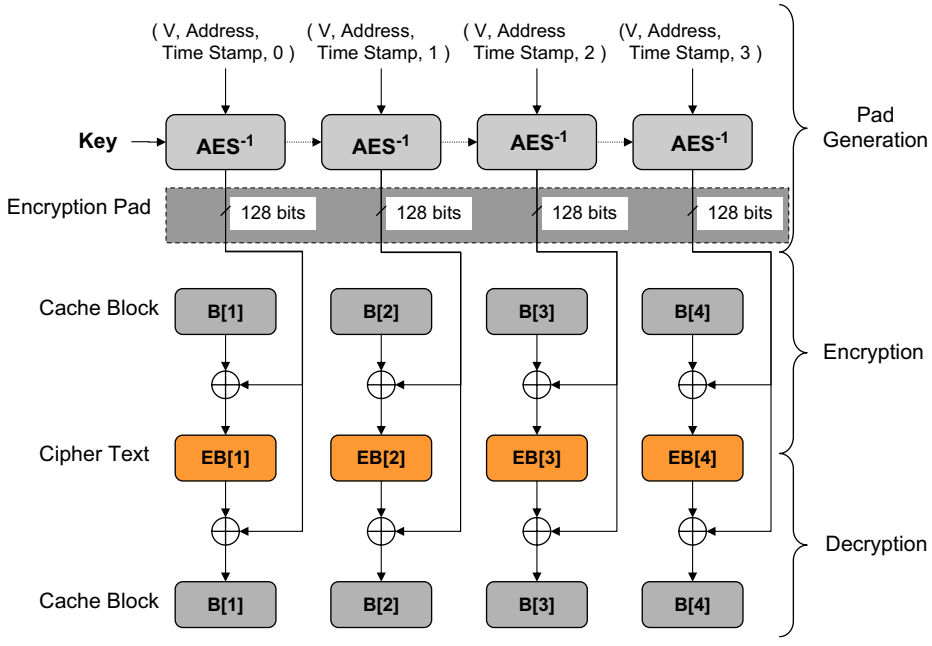

Figure 4 One-time-pad encryption mechanism.

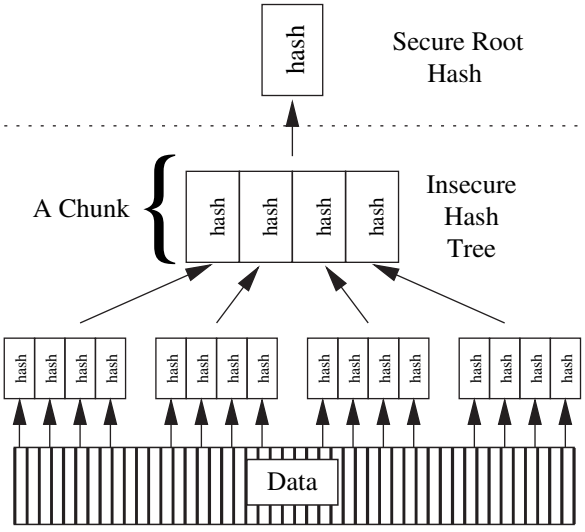

Figure 5 Hash tree protection of IV region.

which may differ between the security kernel and a user application.

The security kernel is also responsible for protecting against malicious programs by isolating the memory space of each user process. This includes separate regions within the dynamic IV/ME region as well as separations within the user processes' static IV/ME region.

Finally, on a context switch, the security kernel is responsible for saving and restoring the user's secure mode and the memory protection regions as a part of process state.

Debugging support

The AEGIS processor supports full debugging by default while in STD mode, but requires it to be specifically enabled while in protected modes. The processor includes whether debug is enabled or not when it computes the program hash. Thus, the security kernel will have different program hashes depending on whether 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.

Protection summary

In summary, any attacks before program execution, such as executing an untrusted security kernel, are detected by a difference in program hashes. During the 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.

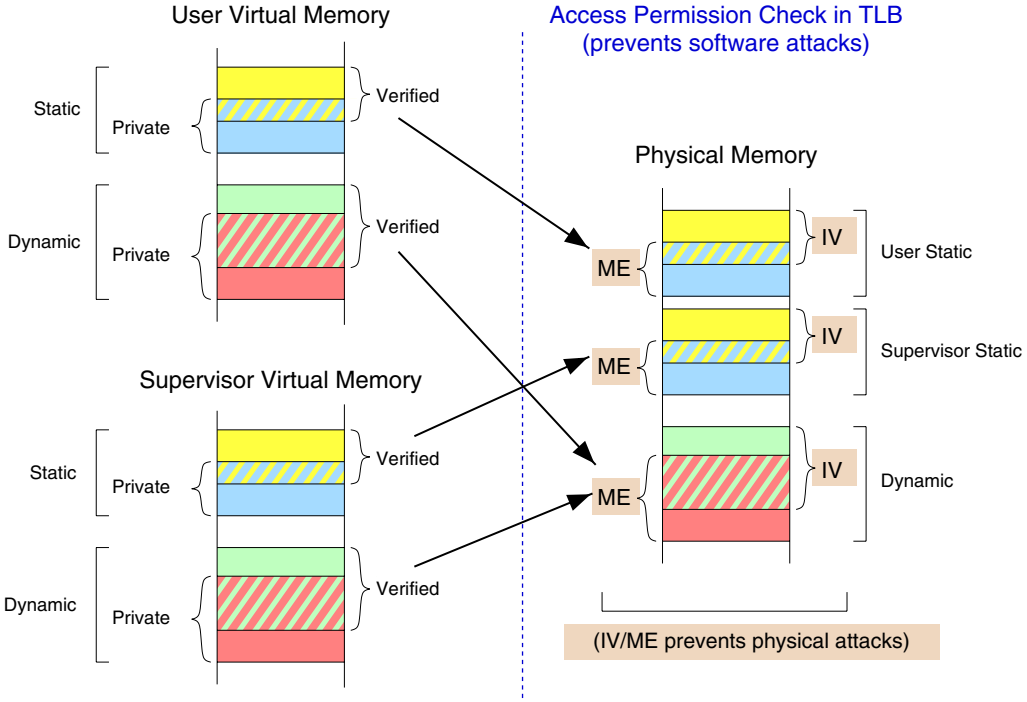

Figure 6 Protected regions in virtual and physical memory.

Overheads

The security capabilities discussed in the sections on 'Physical Random Functions' and 'Processor architecture' do not come for free. These added hardware mechanisms increase the size of the processor core and marginally degrade program performance.

To analyze these overheads we implemented an embedded AEGIS processor core on a Xilinx Virtex2 FPGA based on the OR1200 processor core from the OpenRISC projects. The PUF circuit, integrity verification mechanism, and memory encryption mechanism were added to the core as can be seen in Fig. 7. Security instructions are implemented in firmware software since they are complex and infrequently used, however the embedded memory requirement to hold and execute these instructions is only about 12 KB.

PUF

The PUF circuit size is particularly small compared to the size of an unmodified OR1200 core, although its placement and layout requires special attention. After running this AEGIS core and the OR1200 core through an ASIC synthesis tool, the PUF circuit size was only 2691 gates, or roughly 4.5% the size of the embedded OR1200 core.

The bootstrapping (`getResponse`) and the secret generation are controlled in firmware, and take 1.1 M and 3.2 M cycles, respectively. While this overhead may seem high, these operations will only be performed a few times within an entire program. Therefore, the overhead is negligible when compared to the long execution times of typical programs.

Hardware costs

The integrity verification mechanism, memory encryption mechanism, and permission access checks within the MMU are the only other modifications which required additional logic to be added to the processor core. Using an ASIC synthesis tool, we found that the IV mechanism required 107,756 gates, while the memory encryption mechanism and access checks required 86,655 and 11,587 gates, respectively. All told, the hardware modifications are quite modest when compared with the size of current commercial cores.

System performance

The main performance overhead of the AEGIS processor comes from the two off-chip memory protection mechanisms in two different ways.

1. *Bus contention*: the IV and ME mechanisms share the same memory bus to store meta-data such as hashes and time stamps.
2. *Memory latency:* encrypted data must be decrypted before it can be used by the processor.

Since bus traffic depends on the rate of cache block evictions, the performance overhead is also heavily dependent on the cache miss-rate. A higher miss-rate will increase the amount of processor data which is sent off-chip and needs to be verified and encrypted.

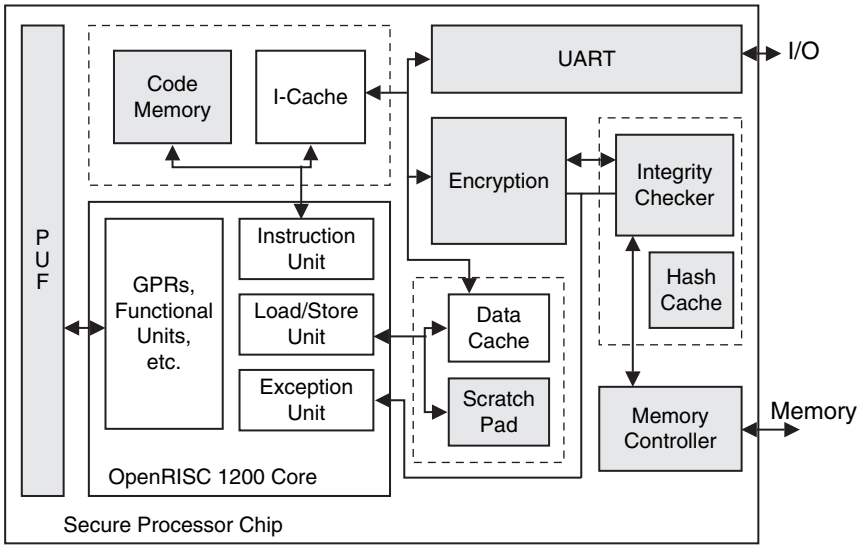

Figure 7 The AEGIS core implementation overview.

To estimate the worst-case overheads, we used a synthetic benchmark that simply reads a large array in the memory with varying cache miss-rates. We found that the percentage slowdown of a program while running in TE mode ranges from 3.8% for a data cache miss-rate of 6.25% to a maximum overhead of 130% when the processor has no cache at all. Similarly, PTR mode exhibits a slowdown of 8.3% and 162%.

More realistic embedded benchmarks, such as the EEMBC benchmark suite show an average percentage slowdown of only 0.1% for programs running in TE mode, and 1.3% for PTR mode. Results from a wider range of benchmarks are also promising and can be found in an MIT CSAIL CSG Technical Memo (Suh et al., 2005).

Conclusions

The AEGIS processor architecture can be used to build computing systems which are secure against both software and physical attacks. Physical Random Functions hold an important role in this, providing a way to reliably create, protect, and share secrets without the use of on-chip non-volatile memory. The four modes of secure execution, which AEGIS provides, enable new ways of creating applications, especially with the use of a suspended secure mode to reduce the trust base without sacrificing physical and software security. An embedded AEGIS architecture implementation has also shown that performance overheads are minimal given typical applications.

References

Alves T, Felton D. Trustzone: integrated hardware and software security. ARM; July 2004 [white paper].

Anderson RJ. Security engineering: a guide to building dependable distributed systems. John Wiley and Sons; 2001.

Claessens J, Preneel B, Vandewalle J. (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 February 2003;3.

Gassend B, Suh GE, Clarke D, van Dijk M, Devadas S. Caches and merkle trees for efficient memory integrity verification. In: Proceedings of ninth international symposium on high performance computer architecture; February 2003.

Jun B, Kocher P. The intel random number generator. Cryptography Research Inc; April 1999 [white paper].

Kocher P, Jaffe J, Jun B. Differential power analysis. In: Lecture notes in computer science, vol. 1666; 1999. p. 388–97.

Lee J-W, Lim D, Gassend B, Suh GE, van Dijk M, Devadas S. 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.

Microsoft. Next-generation secure computing base, <http://www.microsoft.com/resources/ngscb/defaul.mspx>.

O'Donnell CW, Suh GE, Devadas S. PUF-based random number generation. In MIT CSAIL CSG Technical Memo 481; November 2004 <http://csg.csail.mit.edu/pubs/memos/Memo-481/Memo-481.pdf>.

OpenRISC 1000 Project, <http://www.opencores.org/projects.cgi/web/or1k>.

Petrie C, Connelly J. A noise-based IC random number generator for applications in cryptography. IEEE TCAS II January 2000; 46(1):56–62.

Smith SW, Weingart SH. Building a high-performance, programmable secure coprocessor. In: Computer networks (Special issue on Computer Network Security), vol. 31; April 1999. p. 831–60.

Suh GE, Clarke D, Gassend B, van Dijk M, Devadas S. Efficient memory integrity verification and encryption for secure processors. In: Proceedings of the 36th Int'l symposium on microarchitecture; December 2003. p. 339–50.

Suh GE, O'Donnell CW, Sachdev I, Devadas S. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In: Proceedings of the 32nd annual international symposium on computer architecture. MIT-CSAIL-CSG-Memo-483 is an updated version available at. http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf; June 2005.

T.C. Group. TCG specification architecture overview revision 1.2, <http://www.trustedcomputinggroup.com/home>; 2004.

Yee BS. Using secure coprocessors. PhD thesis, Carnegie Mellon University; 1994.