AEGIS: A Single-Chip Secure Processor

G. Edward Suh
Cornell University
Ithaca, NY 14853
suh@csl.cornell.edu

Charles W. O'Donnell and Srinivas Devadas
Massachusetts Institute of Technology
Cambridge, MA 02139
{cwo,devadas}@mit.edu

***Abstract*—This article presents the AEGIS secure processor architecture, which enables physically secure computing platforms with a main processor as the only trusted component. The AEGIS architecture ensures private and authentic program execution even in the face of physical attack using two new security primitives. First, Physical Unclonable Functions (PUFs) generate cryptographic keys in a highly secure yet inexpensive manner, exploiting random manufacturing variations. Second, off-chip memory protection mechanisms ensure the integrity and the privacy of off-chip memory. AEGIS, 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 the AEGIS processor and show that it is reasonable.**

I. 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 [3] 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 Execution Technology [6], formerly named LaGrande Technology, uses a Trusted Platform Module (TPM) from Trusted Computing Group (TCG) [18], to provide authentication mechanisms. Next Generation Secure Computing Base (NGSCB) from Microsoft [10] and TrustZone from ARM [1] also incorporate similar 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 specific 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 Unclonable 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 Section II, we compare our secure computing model with other approaches. Section III describes Physical Unclonable Functions, and Section IV gives an overview of the AEGIS architecture with its memory protection mechanisms. Section V briefly discusses resource and performance costs of our protection mechanisms, and we conclude in Section VI.

II. 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.

A. Tamper-Proof Packages

The conventional approach to building physically secure systems [14], [19] 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 [2] and active instrusion 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.

B. Multi-Chip Approach

Recent efforts to build secure computing platforms implement security functionality in an auxiliary chip. 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.

C. AEGIS Approach

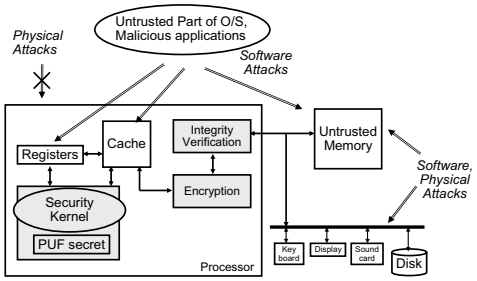

Fig. 1. AEGIS secure computing model.

Figure 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 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 Section III.

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 Section IV.

In this article, we do not consider the attacks using side-channels such as memory access patterns or power supply voltage [8]. To prevent side-channel attacks, the processor must be equipped with additional counter-measures similar to ones that are developed for smartcards. We also do not handle security issues caused by flaws or bugs in software. Finally, we assume the processor has a hardware random number generator [7], [11], [13] to defeat possible replay attacks on communication.

III. Physical Unclonable 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 [2]. 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 may also increase the cost and the complexity of manufacturing even for applications where physical security is a low concern. On-chip EEPROMs require more complex fabrication processes compared to standard digital logic. Fuses do not require more manufacturing steps, but contain a single permanent key and easy to read out.

A Physical Random Function or Physical Unclonable Function (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. Therefore, the PUF output can be used as a unique secret for each AEGIS chip. 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 [4], [9]. Even with identical layout masks, the variations in the manufacturing

process cause significant delay differences among different ICs.

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. Another advantage of PUFs is that they do not require any special manufacturing process or programming steps.

In this section, we describe an implementation of a silicon PUF based on ring oscillators, and discuss how the PUF can be used to express a secret in a secure processor.

A. Ring Oscillator PUF

Figure 2 illustrates a PUF delay circuit that is comprised of many *identically* laid-out delay loops (ring oscillators). This PUF design is called RO PUF. Each ring oscillator is a simple circuit that oscillates with a particular frequency. Due to manufacturing variation, each ring oscillator oscillates with a slightly different frequency. In order to generate a fixed number of bits, a fixed sequence of oscillator pairs is selected, and their frequencies are compared to generate an output bit. The output bits from the same sequence of oscillator pair comparisons will vary from chip to chip. Given that oscillators are identically laid out, the frequency differences are determined by manufacturing variation and an output bit is equally likely to be one or zero if random variations dominate.

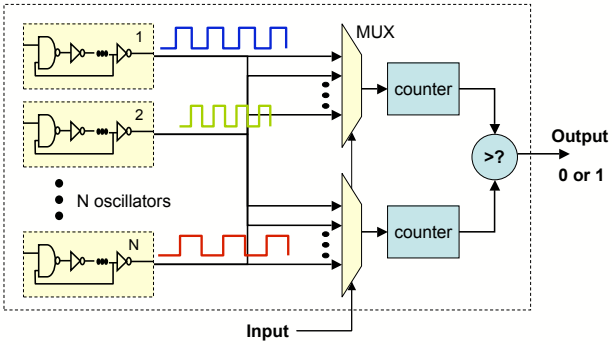

Fig. 2. Ring oscillator based PUF circuit.

Note that it is very easy to duplicate a ring oscillator as a hard-macro and ensure that all oscillators are identical. There is no need for careful layout and routing. For example, the paths from oscillator outputs to counters do not need to be symmetric. By counting many oscillator cycles, the difference in oscillator frequencies can be amplified and will dominate any skews in routing. Therefore, the RO PUF design is easy to implement.

Now let us consider how many bits we can generate from this circuit. Each comparison of a pair of oscillators generates a bit. There are $N(N-1)/2$ distinct pairs given N ring oscillators. However, the entropy of this circuit, which corresponds to the number of independent bits that can be generated from the circuit, is clearly less than $N(N-1)/2$ because the bits obtained from pair-wise comparisons are correlated. For example, if oscillator A is faster than oscillator B, the comparison will yield a 1. If B is in turn faster than C, the comparison will yield a 1. It is clear that when A is compared with C that the comparison will yield a 1 - these bits are correlated.

Fortunately, it is possible to derive the maximum entropy of this circuit assuming pair-wise comparisons, i.e., the number of independent bits that can be generated by the circuit as a function of N, the number of oscillators. There are $N!$ different orderings of ring oscillators based on their frequencies. If the orderings are equally likely, the entropy will be $log_2(N!)$ bits. For example, 35 oscillators can produce 133 bits, 128 oscillators can produce 716 bits, and 1024 oscillators can produce 8769 bits.

For simplicity, it is also possible to use each oscillator only once to generate a single bit and avoid any correlation. For example, 128 pairs of oscillators (256 oscillators total) can be used to generate 128 independent bits.

B. Reliability Enhancement

Ring oscillator frequencies change significantly as environmental conditions such as temperature and voltage change. Of course, we are not using absolute frequencies but rather doing relative comparisons. The PUF output changes only if the ordering of the two oscillators being compared changes.

Figure 3 shows how errors ("bit-flips") could occur due to environmental changes. Say that ring oscillator Blue is faster than ring oscillator Green at room temperature. However, when the temperature increases, both oscillators slow down, with Blue slowing down faster than Green, due to different device or physical parameters. These ring oscillators "flip" when the temperature changes substantially. This flip causes an error in the generated bit.

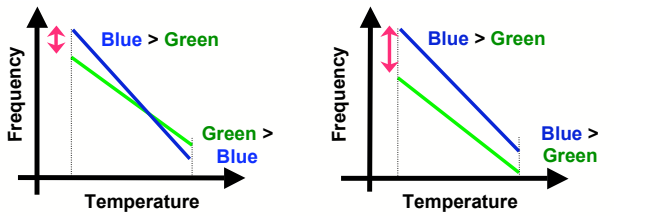

Fig. 3. The relationship between the ring oscillator frequency distance and the probability of a PUF output flip.

The insight from Figure 3 is that ring oscillators whose base frequencies are far apart are much less likely to flip than ring oscillators whose frequencies are close together. This insight can be used to dramatically reduce the error rate of generated key bits by judiciously selecting ring oscillator pairs that will be compared. Specifically, we can remove a significant portion of errors if we only compare ring oscillator pairs, whose frequencies are far apart, to generate key bits.

As mentioned earlier, a fixed sequence of ring oscillator pairs is generated, this sequence now needs to be k times longer than the desired number of bits to be generated. Then,

for each k ring oscillator pairs, we choose the pair that has maximum distance. The bit vector indicating these selections is saved so that the same pairs can be used to re-generate the output. Other masking schemes such as picking n out of m, or using a distance threshold are also possible.

C. Cryptographic Key Generation

In secure processors, PUFs must be used for cryptographic primitives such as encryption and digital signatures. Unfortunately, outputs from the PUF circuits as described are inappropriate as cryptographic keys. Because of noise, the outputs are likely to be slightly different on each evaluation, even if the masking is performed. On the other hand, cryptographic primitives require that every bit of a key stays constant. Moreover, some primitives such as RSA require keys to satisfy specific mathematical properties whereas the PUF outputs are randomly determined by manufacturing variations.

Here, we discuss how PUFs can generate volatile secret keys that can be used for cryptographic operations. There are two components. First, the error correction process, which consists of initialization and re-generation, ensures that the PUF can consistently produce the same output even if there are significant environmental changes such as voltage and temperature fluctuations. Second, the key generation process converts the PUF output into cryptographic keys. The overall process is shown in Figure 4.

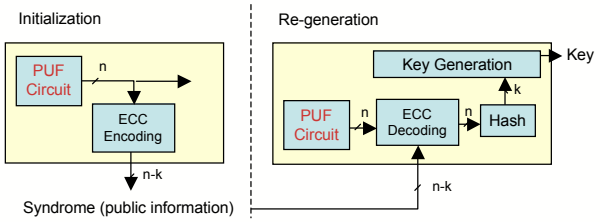

Fig. 4. Cryptographic key generation with PUFs.

In the initialization step, an output is generated from the PUF circuit and the error correcting syndrome for that output is computed and saved for later. For example, the BCH code can be used to compute the syndrome. The syndrome is information that allows for correcting bit-flips in re-generated PUF outputs. If a masking scheme is used, the bit vector that selects oscillator pairs will also be stored along with the syndrome. Note that the syndrome and this bit vector are public information and can be stored anywhere (on-chip, off-chip, or remotely on a server).

To re-generate the same PUF output, the PUF first produces an output from the circuit. If there is a saved bit vector, then that is used to select pairs. Then, the PUF uses the syndrome from the initialization step to correct any changes in the circuit output. In this way, the PUF can consistently reproduce the output from the initialization step.

Clearly, the syndrome reveals information about the PUF delay circuit output. In general, however, 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. For example, we can use the BCH (127,64,21) code to reliably generate 64-bit secrets. The BCH (n, k, d) code can correct up to $(d-1)/2$ errors out of n bits with an $(n-k)$-bit syndrome ($b = n - k$).

While the mask may reveal information about what ring oscillator frequencies are far apart, it does not reveal information about the sign of comparisons, i.e., the bits generated. If a ring oscillator is used many times to generate bits, then it is conceivable that information about ordering of ring oscillators can be extracted from the mask. This can be easily precluded by using each oscillator only once to generate a single bit.

For cryptographic operations that use a randomly selected number as a key, the output of the error correcting code (ECC) can be simply hashed down to a desired length and used as a cryptographic key. For example, symmetric key primitives such as AES can use the hashed PUF output.

For cryptographic operations whose keys need to satisfy special properties (for example, an RSA key pair), the hashed PUF output is used as a seed for a key generation algorithm. In this way, the PUF can generate keys for any cryptographic operation. We note that PUFs simply generate keys that can be used with standard algorithm. There is no change required in cryptographic algorithms.

D. Experimental Validation

The RO PUF circuit has been tested on 15 Xilinx Virtex4 LX25 FPGAs (90nm) [16], where all FPGAs are exactly the same model and therefore identical designs. For the experiments, 1024 ring oscillators are placed on each FPGA and the 1-out-of-8 mask scheme is used.

The experimental result show that two identical PUF circuits on two different FPGAs produce a different output bit with a probability of 46.15% on average (inter-chip variation), which is pretty close to the ideal average of 50%. On the other hand, multiple measurements on the same chip are different only with 0.48% probability (intra-chip variation) even in the worst-case environmental change. The intra-chip variation was studied by changing temperature from -20C to 120C and voltage from 1.08V to 1.32 (+/-10%). This results show that the intra-chip variation is much lower than the inter-chip variation even in the worst-case environmental change.

From the inter-chip and intra-chip variations, we can estimate how reliable the PUF-generated cryptographic keys will be. For example, if the BCH (127,64,21) code is used to generate a key, 10 errors in a 127-bit PUF output can be corrected and the probability of failing to re-generate the same key is less than 5×10^{-11}.

IV. 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 mode (STD) which has no additional security measures, a Tamper-Evident mode (TE) which ensures the integrity of program state, a Private Tamper-Resistant mode (PTR) which additionally ensures privacy, and a Suspended Secure Processing mode (SSP).

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 & 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
- **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

A. 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 Section III). 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 [17].

B. 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” subsections which correspond to read-only data (such as application instructions) and read-write data (such as heap and stack variables).

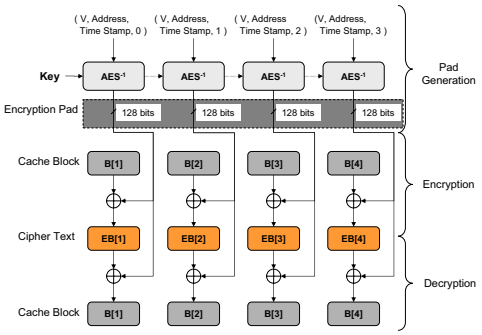

Fig. 5. One-time-pad (counter-mode) encryption mechanism.

Memory encryption is handled by encrypting and decrypting all off-chip data transfers in the ME regions using a One-Time-Pad (OTP, or counter-mode) encryption scheme [15]. Figure 5 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.

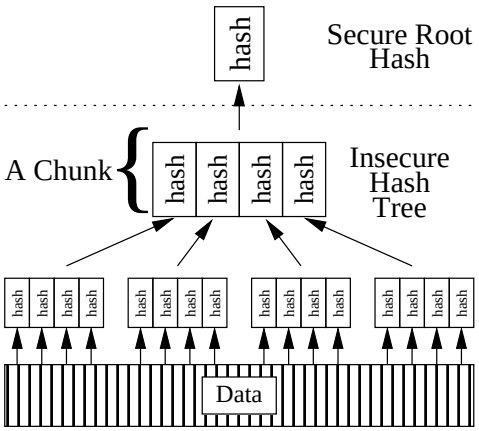

Fig. 6. Hash tree protection of IV region.

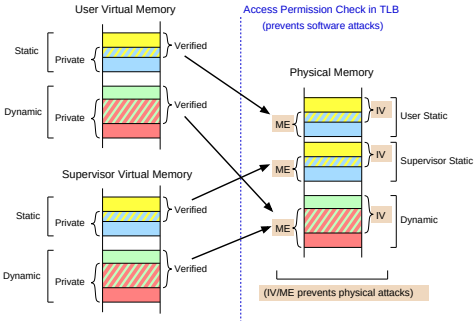

Fig. 7. Protected regions in virtual and physical Memory.

The processor protects the dynamic IV region by creating a hash tree for the region, and saving the root hash node on-chip [5] (Figure 6). 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 access any of the IV or ME protected memory regions.

C. Multitasking

Secure multitasking on the AEGIS processor can be ensured with the help of a trusted security kernel handling such things 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

Figure 7 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 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.

D. 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 [10].

E. 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 prevents illegal software accesses.

V. Overheads

The security capabilities discussed in Sections III and IV 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 project [12]. The PUF circuit, integrity verification mechanism, and memory encryption mechanism were added to the core as can be seen in Figure 8. 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 12KB.

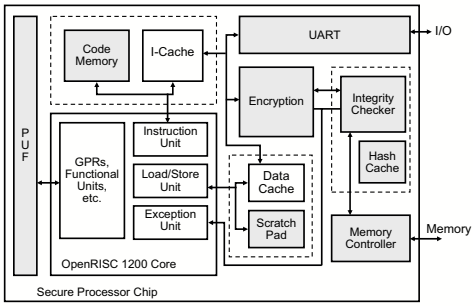

Fig. 8. The AEGIS core implementation overview.

A. PUF

The PUF circuit size is particularly small compared to the size of an unmodified OR1200 core. After running this AEGIS core and the OR1200 core through an ASIC synthesis tool, the PUF circuit size was only $2,691$ gates, or roughly 4.5% the size of the embedded OR1200 core. We note that the result is based on a previous PUF circuit design based on delay paths and an arbiter [17] instead of the RO PUF design presented in this paper. However, we expect the size of the RO PUF circuit to be comparable to the previous design.

The PUF initialization and the key generation are implemented in firmware, and take 1.1M and 3.2M 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.

B. 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.

C. 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.

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 [17].

VI. Conclusions

The AEGIS processor architecture can be used to build computing systems which are secure against both software and physical attacks. Physical unclonalbe 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

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

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

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

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

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

[6] Intel. Intel trusted execution technology. http://www.intel.com/technology/security/, 2007.

[7] B. Jun and P. Kocher. The Intel Random Number Generator. Cryptography Research Inc. white paper, Apr. 1999.

[8] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[9] J.-W. Lee, D. Lim, B. Gassend, G. E. 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*, New-York, June 2004. IEEE.

[10] Microsoft. Next-Generation Secure Computing Base. `http://www.microsoft.com/resources/ngscb/defaul.mspx`.

[11] C. W. O'Donnell, G. E. Suh, and S. Devadas. PUF-Based Random Number Generation. In *MIT CSAIL CSG Technical Memo 481*, November 2004.

[12] OpenRISC 1000 Project. `http://www.opencores.org/projects.cgi/web/or1k`.

[13] C. Petrie and J. Connelly. A Noise-based IC Random Number Generator for Applications in Cryptography. *IEEE TCAS II*, 46(1):56–62, Jan. 2000.

[14] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31(8):831–860, April 1999.

[15] 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, Dec 2003.

[16] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th Conference on Design Automation*, 2007.

[17] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *Annual International Symposium on Computer Architecture*, New-York, June 2005. ACM.

[18] Trusted Computing Group. TCG TPM Specification version 1.2, Revisions 62-94 (Design Principles, Structures of the TPM, and Commands). https://www.trustedcomputinggroup.org/specs/TPM/, 2003-2006.

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