Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions

G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{suh,cwo,isachdev,devadas}@mit.edu

ABSTRACT

Secure processors enable new applications by ensuring private and authentic program execution even in the face of physical attack. In this paper we present the AEGIS secure processor architecture, and evaluate its RTL implementation on FPGAs. By using Physical Random Functions, we propose a new way of reliably protecting and sharing secrets that is more secure than existing solutions based on non-volatile memory. Our architecture gives applications the flexibility of trusting and protecting only a portion of a given process, unlike prior proposals which require a process to be protected in entirety. We also put forward a specific model of how secure applications can be programmed in a high-level language and compiled to run on our system. Finally, we evaluate a fully functional FPGA implementation of our processor, assess the implementation tradeoffs, compare performance, and demonstrate the benefits of partially protecting a program.

1. INTRODUCTION

As computing devices become ubiquitous, interconnectivity and interdependability are escalating the need for secure and trusted computation. Public and private data must be certified and safeguarded while still allowing operating systems and applications the flexibility and performance users demand. To further complicate matters, the proliferation of embedded, portable devices is creating security risks which software-only solutions cannot handle. Even large-scale software security efforts can suffer from a single point of failure when an attacker has physical access to a trusted client PC or laptop. This attacker could snoop or compromise the client's hardware and bypass the software security measures. Secure processors must therefore be able to defend against physical attacks which could corrupt data, discover private data, or violate copy-protection, and defend against tampering that renders software protection useless.

Fundamental to almost any security question is the idea of a secret. Whether a secret is a cryptographic key, or merely a hidden certificate, a secure processor must be able to generate, protect, and share that secret with the outside world. Using this vital ability, it is possible for a processor to achieve several goals. First, an environment can be provided which ensures that any physical or software tampering which alters the behavior, state, or data of a program will be detected. This is called a *tamper-evident execution environment*. Second, a *private and authenticated tamper-resistant execution environment* can be provided which will protect the confidentiality of a program and its data as well as detect tampering by any software or physical process. However, for an application to execute securely, these environments need not be omnipresent, but merely available and persistent at different points of execution.

We will illustrate the benefit of these two environments with examples which demonstrate a new class of secure applications. One such application is a sensor network which monitors environmental conditions. These sensor nodes are vulnerable to physical attack, and even a single compromised node can propagate fake messages which bring down the entire network. Such attacks are preventable if the message passing routines of each node are protected by a tamper-evident environment. Trusted distributed computing is another application which is only possible if the remote system is providing a tamper-resistant environment. Finally, it is worth noting that it is possible to use private tamper-resistant environments to enable copy-protection of software and media content in a manner which is resistant to both software and physical attacks.

We present in this paper the AEGIS single-chip secure processor architecture which uses Physical Random Functions (or Physical Unclonable Functions, PUFs), to veritably create and maintain secure secrets. PUFs have been previously proposed in [5]. We show how to use a PUF for *reliable* secret generation and how to use PUF secrets to design and build a secure processor.

Our secure processor architecture is able to provide the secure environments mentioned which are necessary for trusted and private computation. To guarantee a tamper-evident environment, our processor protects against physical attacks on memory and prevents any physical or digital tampering of programs before their execution. Physical memory tampering is precluded by a hardware integrity verification algorithm, and private tamper-resistant execution maintains privacy through efficient on-chip memory encryption and decryption methods. Other software attacks are countered with architectural modifications to the processor governing access rights, which are under the control of a security kernel.

Moreover, unlike prior secure architecture proposals, our processor does not require that an application remain in a secure execution environment at all times. We have added a new *suspended secure environment*, which is an environment that can be switched to from within a secure environment to allow tamper-evident or tamper-resistant state to persist in suspension while unprotected program code is executed. This unprotected execution is prohibited from accessing anything belonging to the secure environment, and is required to return control flow to a specific program location.

A suspended secure environment enables applications to have a more flexible trust model. For example, it allows applications to integrate library and API functionality into their program without requiring those libraries to be verifiably trusted. As it is commonly the case that only small regions of code need to be trusted to achieve the security requirements of the entire application, this method of execution is quite useful. Even applications which contain no libraries can benefit from this same feature, resulting in performance gains. (This reduces the amount of code which is run with any

inherent overheads of tamper-evident or tamper-resistant security.)

In this paper, we also describe means by which a program can be written in a high-level language and compiled to execute securely within the three environments discussed. In this programming model, procedures and data structures are considered unprotected, verified, or private, and a compiler can easily map these constructs to the architectural features provided.

Given this architectural definition of a secure processor, we have constructed a fully operational RTL implementation which is running on an FPGA. Results show that the overhead inherent with secure computing is reasonably low, and, in coordination with well designed applications and the availability of a suspended secure environment, the performance overhead is small.

In Section 2 we present our security model and overview. Sections 3 discusses a reliable implementation of a PUF and how to share secrets. Section 4 details the processor architecture, and Section 5 describes its programming model. Section 6 discusses the FPGA implementation of our processor. Section 7 presents resource and performance evaluation. Related work is described in Section 8, and we conclude in Section 9.

2. SECURE COMPUTING

Our secure computing platform aims to enable remote users and computers to *trust* the computation 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 a way for remote users and computers to authenticate hardware and software of the platform. In this section, we discuss our attack model and protection mechanisms required under that model.

2.1 Model

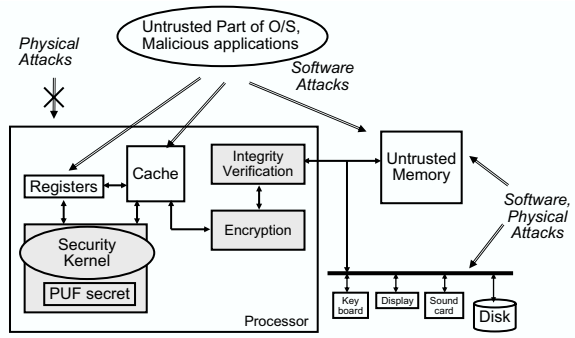

Figure 1: Our secure computing model.

Figure 1 illustrates our model. Put briefly, a secure processor is trusted and 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 at will by an adversary.

Because off-chip memory is 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 note that simply encrypting memory may not be sufficient to provide complete opacity of program operation. Information can be leaked via side-channels such as memory access patterns [24] or power supply voltage [11]. Here we assume that applications are well-written, using techniques to prevent information leaks [1, 7] and the processor is equipped with mechanisms that are commonly used in today's smartcards to prevent side-channel attacks. We also do not handle security issues caused by flaws or bugs in an application.

In our model, the processor contains secret information to authenticate itself and to allow it to communicate securely with the outside world. We will also assume the processor has a hardware random number generator [9, 16, 18] to defeat possible replay attacks on communication.

Finally, a part of the operating system, called the *security kernel*, is trusted and operates at a higher security privilege than the regular operating system. The security kernel is responsible for providing secure multitasking functionality to user applications. It is also given custody of virtual memory management and must handle permission exceptions. The processor must also protect against any modifications to the security kernel code itself since the kernel is stored in vulnerable external storage devices. This can be done using a *program hash*, which is discussed next.

2.2 Architecture Overview

In this subsection, we demonstrate the basic security features of our architecture, and discuss the protection mechanisms to implement them.

To illustrate our 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 following pseudo-code 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.

```
DistComputation()
{
  x = Receive();                // receive Alice's input
  result = func(x);             // compute
  key = sys_puf_secret(C);  // get a PUF secret (known C)
  mac = HMAC(key, (result,x));  // sign the result
  Send(result,mac);             // send the result
}
```

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 a third party could pretend to be Bob. If Bob is using our processor, Alice is guaranteed that the correct computation has been done by Bob's computer.

To certify that Alice's application is run on Bob's computer, our processor uses the combination of a program hash and an internal processor secret. This secret can be based on a PUF or conventional digital memory, see Section 3. Let us assume that Alice shares a unique secret with the processor (we explain the details of secret sharing in Section 3.3). When Bob starts his computer, our processor computes the cryptographic hash of his security kernel ($SKHash$). Alice trusts this kernel and knows its hash which the security kernel vendor makes public. This hash uniquely identifies the kernel because an ideal hash makes it cryptographically infeasible to write another kernel with the same hash value. When Bob starts the `DistComputation` program, the security kernel computes the hash of the application ($AHash$), which includes `func`, `Receive`, `Send`, and `HMAC`.

Within `DistComputation`, the `sys_puf_secret` system call generates a secret key (`key`) depending on $SKHash$, $AHash$, and the processor's internal secret (see Section 3.3 for more details). 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 `key`, she can verify, using the `mac` from Bob, that `DistComputation` was executed on Bob's computer.

Program hashing and secrets 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 security kernel have not been tampered with during execution.

Our processor ensures the integrity and the privacy of off-chip memory since physical attacks can modify and inspect memory contents. This is either done with a Tamper-Evident (TE) execution environment, which verifies values returning from off-chip memory, or with a Private Tamper-Resistant (PTR) execution environment which additionally encrypts stores to off-chip memory. Additionally, with time-sharing, malicious software may tamper with protected applications while they are interrupted. Our processor therefore protects interrupted applications by providing the security kernel with mechanisms to handle virtual memory management and context switching.

With the above protection mechanisms, Alice can trust Bob's results even when his time-shared computer is in a hostile environment. However, in this approach the entire `DistComputation` application and the entire operating system 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. Operating systems also have many potentially dangerous components such as device drivers and I/O interfaces. 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 `DistComputation`, 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 `HMAC` computation. 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.

Our processor supplies a special execution mode where untrusted code can be executed with no overhead, but cannot tamper with the protected code. We call this environment Suspended Secure Processing (SSP). SSP mode allows applications to have a more flexible trust model, and improves performance.

In summary, a secure processor should support remote authentication, off-chip memory protection, secure multitasking, and Suspended Secure Processing (SSP). In our approach, the processor protects initial state with program hashes and shares secrets using PUFs. Memory is protected for both user applications and the security kernel via TE and PTR execution, and the SSP environment is provided for flexibility and performance. A protected security kernel handles the computation of an application program hash and secures multitasking.

3. PHYSICAL RANDOM FUNCTIONS

As noted in our security model, a processor 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. The manufacturer programs the non-volatile memory with a chosen secret such as a private key, and authenticates the corresponding public key to the users.

Unfortunately, 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 digital secret from the chip. While it is possible to add various countermeasures to defeat the potential attacks on on-chip secrets, available protection mechanisms are quite expensive and need to be battery powered continuously (This is required to detect tampering even when the computing device is off.)

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 substantially 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 (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.

In this section, we describe a candidate implementation of a silicon PUF, and how the PUF can be used to share a secret between a secure processor and a user.

3.1 Silicon PUFs

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.

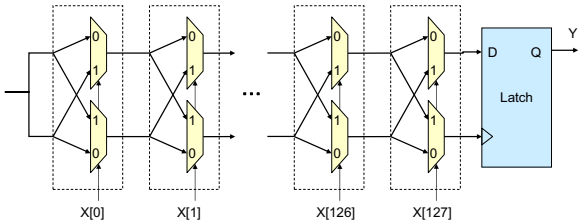

Figure 2: A silicon PUF delay circuit.

Figure 2 illustrates the SPUF delay circuit used in this paper. While this particular design is used to demonstrate the technology, 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 0.18μm, single-poly, 6-level metal process [12]. 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 Celsius and regulated voltage changes of $\pm 2\%$, the output noise is 4.8% and 3.7%, respectively. Even when unrealistically increasing the temperature by 100C 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 an ideally symmetric layout of the circuit in Figure 2 would 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 $R = PUF(C)$. 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. 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 with 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. As the PUF circuit has only a few hundred gates, the duplication incurs a modest increase in gate count.

3.2 Preventing Model Building

Because our PUF circuit is rather simple, attackers can try to construct a precise timing model and learn the parameters 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.

3.3 Secret Sharing

PUF responses can be considered as secrets because they are randomly determined by manufacturing variations, and difficult to predict without access to the PUF. If users know secret PUF responses of a secure processor, they can authenticate the processor.

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. 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

$$R = PUF(C) = PUF(H(SKHash \circ PreC))$$

where $H()$ is an ideal cryptographic one-way hash function, $\circ$ represents the concatenation, and $PUF()$ is the physical random function. As a result, a malicious kernel cannot obtain the response for a specific challenge C using `l.puf.response`. To do this the kernel would have to find the input $PreC'$ that produces the challenge $C = H(SKHash' \circ PreC')$ for its program hash $SKHash'$. (equivalent to finding a collision in the one-way hash function $H()$.)

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 $PreC = H(AHash \circ$ *UserPreC*) so that the challenge depends on both the security kernel and the user application ($AHash$ is the program hash of the application).

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 $C = H(SKHash \circ H(AHash \circ UserPreC))$. In this case, $UserPreC$ should be kept secret so that others cannot use the same application and generate the same CRP.

Bootstrapping can 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, $UserPreC$ 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.

A second instruction, `l.puf.secret`, allows the security kernel to share a secret with a user who already knows a CRP. This instruction takes a challenge C as an input and returns an m-bit secret K,

$$K = H(SKHash \circ PUF(C)).$$

The secret is the cryptographic hash of the program hash $SKHash$ concatenated with the PUF response $PUF(C)$. 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 $R = PUF(C)$ from a secret K' obtained from $SKHash'$ since it requires inverting the one-way hash function.

A security kernel provides a system call `sys_puf_secret(`C`)` to user applications so that each application can generate a unique secret. The system call takes a challenge C as an input, and returns $H(AHash \circ$ `l.puf.secret`$(C))$ so that the secret depends on both the security kernel and the application.

3.4 Reliable Secret Generation

The PUF instructions as described are inappropriate to generate secrets that can be used as cryptographic keys. Because of the measurement noise, PUF responses are likely to be slightly different on each evaluation, even for the exact same challenge C. However, 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 our PUF so that the same secret can be generated on every execution.

Figure 3 summarizes extended PUF instructions which include error correction. For `l.puf.response`, 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 instruction

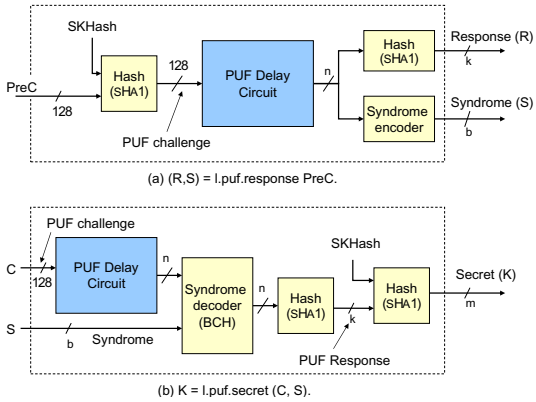

Figure 3: The reliable secret generation using PUFs.

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

The new `l.puf.secret` instruction 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 `l.puf.response` instruction. Finally, the response is concatenated with the program hash, and hashed to generate a secret K.

Obviously, 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 has only a one-out-of-2^k chance to guess the response correctly.

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 instructions 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 the 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 instruction's 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 miscorrect is negligible. Therefore, the processor can always retry in case of the error correction failure. Alternately, hash(K) can be passed along and used to check that the correct key K has been generated to handle miscorrects and failures uniformly.

3.5 PUF-Based Random Number Generation

Since many cryptographic security applications require a source of pure randomness, any secure processor should implement some type of random number generation algorithm on-chip. Hardware algorithms have been proposed before [9, 18], however it is also possible to use the existing PUF circuitry to generate a random number which is acceptable for cryptographic applications [16].

3.6 Security Analysis

In this subsection, 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 and learn the parameters from many CRPs. However, model building is not possible because the PUF instructions never directly return the PUF delay circuit output (see Section 3.2).
- **Duplication**: To bypass the PUF instructions, 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. 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 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.)

4. PROCESSOR ARCHITECTURE

This section describes our processor architecture that provides secure execution environments under software and physical attacks. The processor can be used in a multitasking environment, running mutually mistrusting processes.

4.1 Secure Execution Modes

To allow for varying levels of security, the processor not only has user and supervisor modes, but also has four secure execution modes a program operates in: Standard mode (STD) which has no additional security measures, TE mode which ensures the integrity of program state, PTR mode which additionally ensures privacy, and Suspended Secure Processing mode (SSP).

SSP mode implements the suspended secure environment discussed previously, giving a smaller trusted code base and improving performance. SSP mode can only be transitioned to when running in TE or PTR mode.

To keep track of which security mode a supervisory process is currently in, the processor maintains the SM mode 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 UM bits keep track of the user process security mode, and are managed by the security kernel.

Mode	VM Control	Memory Verifi ed/Private	Mode Transition	PUF Insts
STD	Y	-/-	TE (`enter`)	N
TE	Y	RW/-	PTR (`csm`) SSP (`suspend`) STD (`exit`)	N
PTR	Y	RW/RW	TE (`csm`) SSP (`suspend`) STD (`exit`)	Y
SSP	N	R/-	TE/PTR (`resume`)	N

Table 1: The permissions in each supervisor execution mode. The prefix `l.secure` is omitted for instructions.

Table 1 summarizes the four 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 PUF instructions. The user mode permissions are very similar except that no mode has control of the VM mechanism. Each protection mechanism is discussed in more detail in the following subsections.

4.2 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. Our architecture defends against software attacks by adding an access permission check within the memory management unit (MMU), and protects against physical attacks with integrity verification (IV) and memory encryption (ME).

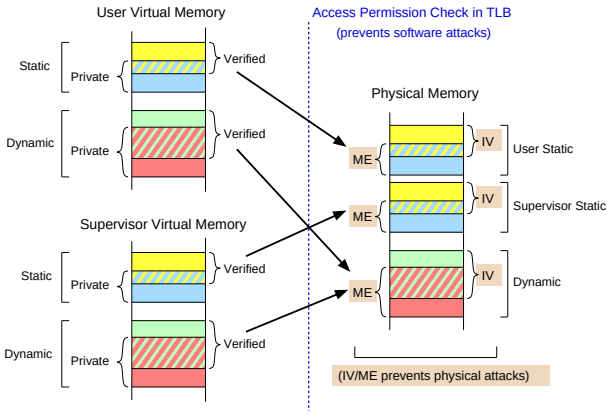

Figure 4: Protected regions in virtual and physical Memory

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

As shown in Figure 4, the processor separates physical memory space into regions designated "IV protected" or "ME protected" (allowing for overlap), to provide protection against physical attacks on off-chip memory. 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 "static" or "dynamic", differentiating between 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 [21]. The difference between the static and dynamic ME regions is that the dynamic ME region requires time stamps to be stored in memory along with encrypted data whereas the static ME region does not need the time stamp. The location of the time stamps is specified by the security kernel.

The processor protects the dynamic IV region by creating a hash tree for the region, and saving the root hash node on-chip [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. Again, the security kernel should reserve the memory for hashes and MACs, and properly set the special registers of the integrity verification unit.

Once the security kernel is in the secure mode and protected from physical attacks, it can configure and enable the mechanisms to defend 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.

1. Read-only (static) Verified memory
2. Read-write (dynamic) Verified memory
3. Read-only (static) Private memory
4. Read-write (dynamic) 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 and PTR modes, but only allow read access in STD or SSP mode. The private regions are accessible only in PTR mode.

To properly protect user's verified and private regions from physical attacks, it is clear that the virtual memory manager (VMM) in the security kernel needs to map the private and verified regions in the virtual space to the IV and ME regions in the physical memory space. Figure 4 illustrates how this is done, with verified regions mapping to IV regions and private regions mapping to ME regions.

The processor only needs to have a single dynamic IV/ME region (one each), because the dynamic regions can be shared between the user's and supervisor's dynamic Verified/Private regions; this can be handled easily by a security kernel's VMM. On the other hand, the processor separately provides user-level static IV/ME regions and supervisor-level static 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 decryption 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. Modifying these special purpose registers defining these regions is done through a supervisor-level instruction, `l.secure.cpmr`, which can be used only in the TE/PTR mode. It should be noted, however, that modification of the boundary of an existing IV region can be quite expensive, as it requires the rebuilding of an entire hash tree.

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 in the hash tree mechanism, 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.secure.*` or `l.puf.*`) or an exception takes place, the processor waits until all instructions and data have been verified. In PTR mode, the same conditions are true, except that the processor must also wait for all previous off-chip accesses to be verified before stores which write data to non-private memory. This is to confirm a store was indeed intended since otherwise private data could leak into non-private memory.

4.3 Execution Mode Transition

Our processor architecture controls the transition of the supervisor's security mode while relying on the security kernel to control multitasking user applications. Therefore, we first describe the processor support for the security kernel's mode transition, and briefly discuss how the security kernel can provide the same functions to the user applications.

4.3.1 Security Kernel

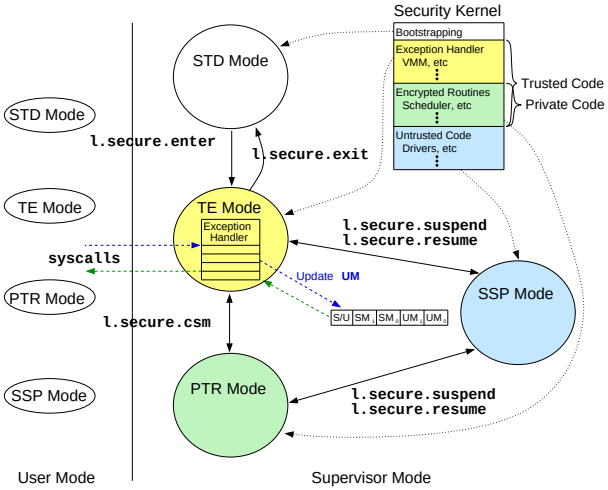

Figure 5: Security modes and transitions

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

The processor begins operation within the supervisor STD mode. Regular operating systems that do not use our 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 defend against software memory attacks. To enter TE mode and begin executing in a secure environment, the security kernel executes `l.secure.enter`. This instruction, as with all of the security instructions introduced here (`l.secure.*`), requires supervisor permissions to run.

To ensure a secure start-up, the `l.secure.enter` instruction must be called with parameters defining boundaries for integrity verification (IV), memory encryption (ME), 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 dynamic ME region, and enable the ME mechanism.
4. Compute and save the security kernel's program hash.
5. Set the execution mode to be TE.

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.

A $SKHash$ is computed over the binary defined by PC-relative offset parameters as well as by the IV/ME boundaries. 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

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

where $d1$ and $d2$ are the reverse and forward offset distances specifying the region of memory containing the kernel code, respectively, and B contains all IV/ME boundary addresses and protection settings such as whether debugging is enabled or not. 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.

Once running in the protected TE mode, the "Change Secure Mode" instruction (`l.secure.csm`) can be executed to transition back and forth between TE and PTR modes. The `l.secure.csm` instruction is rather straightforward; the processor simply changes the security mode by setting the SM bits. The major difference between TE and PTR modes is that the private regions of memory can only be accessed under PTR mode. Additionally, PTR mode may have degraded performance because it must ensure the authenticity of stores which can potentially write private data into public regions of memory, (see Section 4.2). The `l.secure.csm` instruction can only be used in either TE or PTR mode.

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

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 regions. They cannot write into the Verified regions or access the Private regions. The programs also cannot change the VM or use the PUF instructions in SSP mode (cf. Section 4.1). Second, programs can only return to the suspended secure execution mode at the exact location specified by the `l.secure.suspend` instruction.

The `l.secure.suspend` instruction requires the address at which the secure mode 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.secure. resume` instruction. The processor will confirm that the PC value of the resume instruction is the same as the address given by `l.secure.suspend`, and will change the security mode back to the mode which initiated the transfer to SSP mode.

Finally the `l.secure.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 encryption keys and private data in the cache.

As seen in Figure 5, there is one additional way for the processor to enter a supervisor's secure execution mode; system calls

and exceptions 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.secure.enter` has already been called without a `l.secure.exit`), system calls and exceptions trigger the processor to enter the supervisory TE mode at the hardwired handler location. The security kernel should ensure that proper handler code is at the location.

4.3.2 User Application

Thus far we have only considered mode transitions in supervisor mode. The security kernel is in charge of multitasking, and provides the same security functionality to user applications. To assist in this, the processor only allows the UM bits (which determine the user-level security mode) to be updated by the security kernel while in the supervisory TE/PTR modes.

For full application support, the security kernel must duplicate the processor's security instructions as system calls. For instance, the instruction `l.secure.enter` can be emulated in the security kernel by a system call, `sys_secure_enter()`, which performs similar operations. The only difference is that the user-level program specifies the protected regions in virtual memory space (Verified and Private) rather than in physical memory space. The security kernel alters the VMM mapping to direct the user program's Verified/Private regions to the IV/ME regions in physical memory. Other calls such as `sys_secure_csm()` or the SSP entrance call `sys_secure_suspend()` can be handled in supervisory mode by modifying the UM bits, updating any internal state held by the kernel, and returning control to the user-level process.

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

4.4 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.secure.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 [15].

4.5 Protection Summary

Attacks	Protection
Initial state corruption	Program hashes
Physical attacks on off-chip memory	Integrity verification, memory encryption
Inter-process software attacks	Virtual Memory (VM)
Software attacks in SSP mode	Permission checks in the MMU

Table 2: The protection mechanisms in our architecture.

Table 2 summarizes the protection mechanisms used in our processor architecture, and illustrates that our architecture indeed prevents all the plausible attacks. Any attacks before the program starts an execution, such as executing an untrusted security kernel, are detected by different program hashes and different secret keys from the PUF. 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.

5. PROGRAMMING MODEL

Having described the architectural features of our processor, we now focus on how user applications can be written and compiled to allow for secure computation. Specifically, we discuss how the program's code and data structures can be placed in the four protected memory regions, how the program hash can be computed for both encrypted and unencrypted applications, how the programs can securely switch between trusted code and insecure library code, and how applications that contain multiple libraries encrypted with different keys can be supported while the processor only supports one decryption key.

Using high level languages, we propose a programming methodology which would create secure applications by separating the application's procedures and data structures into three different types. From a programmer's perspective, functions and variables will either be unprotected, verified, or private. The functions and variables are kept within the corresponding regions of memory stated in Section 4.2.

Unprotected functions execute in STD or SSP mode, and are restricted to access only unprotected variables. Verified functions and private functions are trusted and allowed to access both verified and private variables. The difference is that private functions are encrypted and always execute within PTR mode while verified functions can execute in either TE or PTR mode depending on whether they operate on private variables or not. Verified functions use the verified stack in TE mode, and the private stack in PTR mode. Finally, the application begins with a `sys_secure_enter()` call, and switches the mode at the function call boundary.

This technique would require three separate stacks and three separate heaps to hold the differently tiered variables, as well as three different regions of memory to hold the instruction code. Programmers must then be aware of the dissimilar stack variables, which depend on their functional scoping, as well as three complementary heap allocation functions (e.g., `malloc`).

It is clear that a compiler will be necessary to manage the placement of these memory regions, and to insert the necessary code to securely transfer between modes on procedural boundaries. In order to trust the secure application generated, this compiler needs to be trusted. However, note that malicious compilers can only compromise the applications compiled with them. Other applications executing with the compromised applications are protected by our processor architecture.

5.1 Program Memory Layout

Given our programming methodology, we depict in Figure 6 one way in which a compiler could arrange the various segments of an application in the virtual memory space. At the bottom of memory we have the unprotected code which will be executed under either STD or SSP mode. Above that unprotected code we have our static verified region which is comprised of the public but verified functions as well as the private functions. After the static verified region, we have the dynamic verified region which holds all of the verified and private variables, the two protected stacks, and the two protected heaps. Naturally, the private segment is a subsection of the verified region. At the top of the memory space we keep the unprotected variables and the unprotected stack and heap.

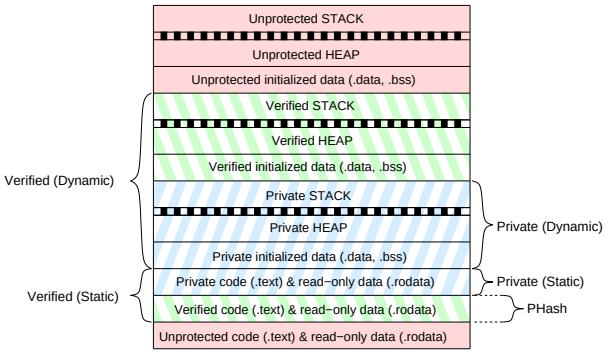

Figure 6: Typical program layout, protection regions, and program hash with private code.

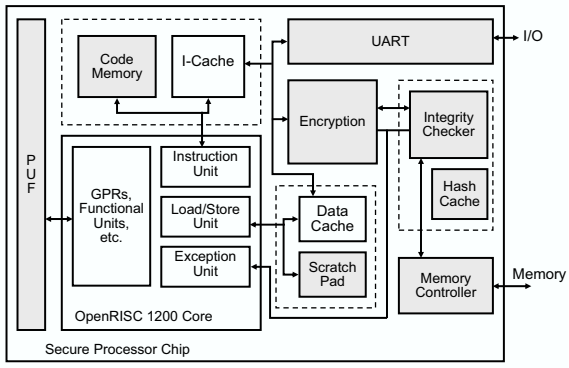

Figure 7: The overview of our processor implementation.

One point to notice is that the entire application can be uniquely identified with the $AHash$ computed only over the public part of the static verified region containing verified code and read-only data. To check the rest of the code and data, a compiler can add authentication code in the static verified region, which will run in PTR mode immediately after the `sys_secure_enter()`. This code will compute the hash of all other initial code and data, and compare it against a hash value saved in verified read-only data (.rodata). Since the hash computation will be done on the decrypted values of the private regions this allows the $AHash$ to remain consistent despite the possibility of different cipher texts for different encryption keys.

5.2 Mode Transition

Transitioning between execution modes when making function calls requires only a slight instruction overhead which a compiler could easily add as part of its calling convention.

If execution is changing from TE mode to PTR mode, there must be code inserted which changes the security mode, changes the stack pointer to the private stack, and then handles the rest of a standard calling convention routine (such as placing arguments). Naturally, changing from PTR to TE mode would require a similar stack redirection from the private one to the verified one.

When transitioning from TE or PTR mode to SSP mode more caution must be taken. The compiler should similarly change the stack pointer, but also determine a location in the caller function where the callee (unprotected) function will return to when it has completed. This address must be passed as an argument to the `sys_secure_suspend()` call. The location which was determined will contain the `sys_secure_resume()` call, and will be located within the caller function. By doing this, a callee function does not need to know about any security instructions, and can be precompiled without any special compiler handling. This allows the usage of library code and other third party binaries.

It is even easy to transition between procedural calls from two private libraries which are encrypted with different keys. In this situation a compiler can add instructions which cease executing from the first library, swap the two static encryption 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.

6. IMPLEMENTATION

We implemented our 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 25MHz on a Xilinx Virtex2 FPGA with 256-MB off-chip SDRAM (PC100 DIMM). Note that 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.

Figure 7 illustrates our secure processor implementation. The processor is based on the OR1200 core from the OpenRISC project [17]. 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. Therefore, our implementation can be considered as a secure embedded processor.

Most security instructions such as `l.secure.enter` perform rather complex tasks and are only used infrequently. As a result, the instructions to change the execution modes and the PUF instructions are implemented in firmware software. The processor enters a special trap mode on those software-implemented instructions, and executes the appropriate code in the on-chip code ROM. The processor also has a on-chip scratch pad that can only be accessed within the special trap 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 two PUF instructions. The PUF is only accessible in the special trap 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 dynamic IV region, and a simple MAC scheme protects the static region. For encryption, a 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 simple extensions independent to the processing core. Therefore, these protection modules can be combined with more complex processing cores in the same way. Moreover, the embedded computers are likely to be the main applications for secure processors because they often operate in a potentially hostile environments that require physical security. Thus, the evaluation of embedded secure processors is in itself interesting.

6.1 Implementing Cached Hash Trees

As noted in Section 4.2, we use a cached hash tree to protect the integrity of off-chip memory. However, because the original algorithm [6] cannot be implemented efficiently, our processor uses a slight variation.

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, verifying a read value or updating the hash tree with a write-back value can incur another cache miss and a write-back before the value can be dequeued from

the buffer. As a result, in the worst case when many cache misses and write-backs occur, the original algorithm requires an extremely large buffer (potentially as large as the cache itself).

Instead of having large enough buffers for the worst case, our implementation has small buffers (eight entries each for reads and write-backs) and only use the hash tree algorithm with caching if the buffer space is available. Otherwise, a simple hash tree without caching is used.

7. EVALUATION

This section evaluates our secure processor design based on the RTL implementation. We first study the overheads of new security instructions and the off-chip protection schemes such as integrity verification and encryption. Then, we analyze sensor network software for environmental monitoring to demonstrate the importance of being able to partially trust the code.

Parameters	Specifi cation
Processor	OR1200 core, 25 MHz
Off-chip SDRAM	64MB, 64bits/cycle, 12.5MHz
I/D cache	32KB, direct-mapped, 64B line
IV unit	8KB, direct-mapped, 64B line cache 3 AES blocks (3 * 128bits/12 cycles)
ME unit	4KB, direct-mapped, 64B line cache 5 SHA-1 blocks (5 * 512bits/80 cycles)

Table 3: The default processor parameters.

In the evaluation, we use the parameters in Table 3 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 the same memory latency in clock cycles as we would see with actual ASIC processors that have a much higher clock frequency. The parameters of IV and ME units are selected to match the off-chip bandwidth.

7.1 Security Instructions

The new security instructions incur both space and performance overheads. Because they are implemented as firmware executing in the special 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.

Table 4 summarizes the overheads of our security instructions. For each instruction, the table shows the size of the code in the code ROM, the size of the on-chip scratch pad (SRAM) required to execute the code, and the number of cycles that each instruction takes to execute. The execution cycle does not include the overhead of flushing the pipeline for a trap, which is only a few cycles per instruction.

The space overhead of our instructions are 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 make use of 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. We note, however, that the memory requirement of BCH decoding depends on the length of the codeword, and the execution time can vary significantly by the number of errors being corrected. While our current implementation uses a 255-bit codeword, it is always possible to trade spaces 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.

The performance of our instructions is not a concern, either. All instructions that would be used frequently take a small number of cycles. While some instructions such as `l.secure.enter` and `l.puf.secret` can take a high number of cycles, they will

Instruction	Code size (B)	Memory req. (B)	Execution cycles
`l.secure.enter` (1)	576	256	22,937+2m+n(1)
`l.secure.exit`	24	0	25
`l.secure.csm`	28	0	18
`l.secure.cpmr`	212	4	196+2m
`l.secure.suspend`	36	0	43
`l.secure.resume`	72	0	48
`l.puf.response` (1,2,4)	596	1,236	48,299+2(1)+(2)+(4)
`l.puf.secret` (1,3,4)	600	1,240	57,903+2(1)+(3)+(4)
(1) $SHA1\ hash$	1,960	120	4,715
(2) bch_encode	468	20	161,240
(3) bch_decode	5,088	1,100	2,294,484
(4) get_puf_resp	880	88	932,159
$common\ handler$	540	0	46 − 92
All	11,080	1,240	–

Table 4: The overheads of security instructions.

be used infrequently in applications. The `l.secure.enter` instruction should only appear 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.

In Table 4, 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 execution time of $SHA1\_hash$ could be reduced greatly 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.

7.2 Hardware Resource 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 our secure processor. Using TSMC $0.18\mu m$ libraries, we compare the size of major components in Table 5. The gate count is an approximation based on an average NAND2 size of $9.97\mu m^2$, and ROMs are implanted as combinational logic.

As seen in the table, our processor with all the protection mechanisms is roughly 1.9 times the size of the baseline. However, our logic gate count is $311,083$ versus the baseline $58,671$. However, much of this overhead is due to the extreme simplicity of the OpenRISC core, which only uses $28,164$ gates. (The SDRAM controller UART alone makes up almost 40% of this.) Further, our protection mechanisms are independent of the processor core, remaining nearly constant in size when applied to other cores, and can be implemented within a smaller area for reduced performance.

For example, more than 75% of the logic overhead comes from the IV and ME modules. These modules were designed for relatively high performance cores, and can be considered to have nearly maximal area usage. It is possible to reduce the number of AES and SHA-1 units used (giving lower performance), or even replace them with "weaker" algorithms such as MD5 or RC5 which consume significantly less area. This said, the IV and ME modules are still fairly small compared to more complex cores. The logic size of the IV and ME modules is $1.9mm^2$ compared to the $4mm^2$ core size of the embedded PowerPC 440 ($0.18\mu m$). Considering high performance CPUs, the total area of these modules is $3.5mm^2$

Module	Resource Usage (AEGIS / Base)		
	Logic μm^2	Gate Count	Mem. μm^2 (Kbit)
CPU Core (tot)	576,641 / 281,064	57,784 / 28,164	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	3,104,404	311,083	6,096,442 (730)
Base Totals	585,521	58,671	4,280,922 (546)
Full Chip Area	9,200,846 μm^2 vs. 4,866,513 μm^2 (1.9x larger)		

Table 5: The hardware resource usage of our secure processor.
*** Denotes that values are identical on both AEGIS / Base**

compared to the $42.7mm^2$ die size of the PowerPC 750CXr (G3) ($0.18\mu m$) and the $217mm^2$ die size of the Pentium 4 ($0.18\mu m$).

7.3 Performance Overheads

The main performance overheads of our secure processor comes from the two off-chip memory protection mechanisms: integrity verification and encryption. While we have modified the processor core to support the special trap for security instructions and the MMU for access checks, those changes are carefully designed not to effect the clock frequency.

The 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 processor to off-chip memory bandwidth. Second, the encrypted data cannot be used by the processor until it is decrypted. Therefore, they effectively increase the memory latency.

Table 6 shows the performance of our processor under TE and PTR modes when run on our FPGA. In TE mode only integrity verification is enabled. In PTR mode both IV and ME are enabled. PTR mode experiments encrypt both the dynamic data and static program instructions, however we found that nearly all of the slowdown comes from the encryption of dynamic 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 our processor suffers much of its performance hit during cache evictions, this benchmark attempts to demonstrate worst-case slowdown. Table 6 shows that the performance degradations are not prohibitive for programs with reasonable cache miss rates.

One other major factor which affects performance overhead is the size of the protected dynamic integrity verification region. Table 6 uses 4MB as a typical protected region size, but also breaks down the effects of smaller and larger protected regions using the `vsum` 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 noticably 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. Each EEMBC kernel

	STD cycles	TE slowdown	PTR slowdown
Synthetic "vsum" (4MB Dynamic 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 Dynamic 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 Dynamic 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 6: Performance overhead of TE and PTR execution.

was run using its largest possible data set, and was tested on 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 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.

Due to space constraints, a much more thorough analysis of the performance overheads, including a larger representation of applications and the use of SSP mode, can be found in the extended version of this paper [22].

7.4 Case Study: Sensor Networks

One primary difference between our processor architecture and previously proposed secure processors is that our processor allows only a part of the process to be trusted and protected. 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 [10] 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 the `Surge` application in the `TinyOS` package [13]. This application consists of only three main functions: sensor input processing, ad-hoc networking, and MAC computation. At present, 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 the sensor in simulation using the TinyOS TOSSIM 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 verifi-

cation. 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.

8. RELATED WORK

Recently, there have been significant efforts to build a secure computing platform that is able to authenticate applications with a smaller trusted computing base. Trusted Platform Module (TPM) from Trusted Computing Group (TCG) [8], Next Generation Secure Computing Base (NGSCB) from Microsoft [15], and TrustZone from ARM [2] incorporate some of the mechanisms similar to the ones in our processor architecture. For example, TPM and NGSCB use the program hashes to authenticate software, and NGCSG and Trust Zone provide a higher security level within an application. However, these systems can only handle software attacks, not physical attacks. Our processor is designed to prevent both software and hardware attacks.

XOM [14] is designed to handle a security model similar to ours where both software and physical attacks are possible. However, XOM assumes completely untrusted operating systems whereas we have used a security kernel to handle multitasking and described the detailed architecture to support the security kernel. As a result, our architecture is very different from XOM.

Our processor implementation is based on a previous design [20]. However, there are some key differences in the two designs. First, our new architecture enables programs to be partitioned into trusted and untrusted parts whereas the entire application has to be trusted and protected in the previous design [20] as well as XOM. Second, we use PUFs to to securely store secrets in the processor. Previous work simply assumes that non-volatile memory is secure.

Our architecture uses the off-chip protection mechanisms of integrity verification [6] and encryption [21, 23]. While the mechanisms are not new, we have described a complete processor architecture showing how the mechanisms can be applied for secure processors with a security kernel. Also, we have evaluated a functional RTL implementation of these mechanisms, as opposed to doing a simulation study.

There is one item of note regarding the use of the One-Time-Pad (OTP) encryption in our processor. 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 [19]. Our architecture speculatively uses unverified instructions and data until there are security critical operations (see Section 4.2). However, as discussed in Section 2.1, we assume that the address bus is protected with appropriate schemes such as oblivious RAMs [7] or HIDE [24]. It is important to note that without these protections for side channels, there are other security breaks that can compromise the privacy of applications.

Physical Random Functions and corresponding secret sharing protocols for PUFs have been studied previously [4, 5]. In this work, we detailed how an error correction method can be applied to a PUF for reliable secret sharing, which is crucial to enable cryptographic operations. We also described how the basic PUF can be modified so as to be applicable to secure processors with a security kernel.

9. CONCLUSION

We have described the AEGIS processor architecture that can be used to build a secure computing system. Physical random functions are used to reliably create, protect, and share secrets within processing architecture. The processor also provides four modes of operation which enable new applications for secure computing. The suspended secure mode of execution is a new contribution which allows the trust base of an application to be reduced, and improves performance. We have also shown a programming model which allows programmers to use our secure functionality with high level languages. Our processor has been implemented on an FPGA, and we have shown that the overhead for secure computing is reasonable.

10. REFERENCES

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

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

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

[4] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled Physical Random Functions . In *Proceedings of* 18^{th} *Annual Computer Security Applications Conference*, December 2002.

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

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

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

[8] T. C. Group. TCG Specifi cation Architecture Overview Revision 1.2. `http://www.trustedcomputinggroup.com/home`, 2004.

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

[10] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *Second ACM Conference on Embedded Networked Sensor Systems (SensSys 2004)*, November 2004.

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

[12] 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 Identifi cation and Authentication Applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, June 2004.

[13] P. Levis, S. Maddena, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. 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.

[14] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

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

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

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

[18] 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.

[19] W. Shi, H.-H. Lee, C. Lu, and M. 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.

[20] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSG-Memo-474 is an updated version)*, June 2003.

[21] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Effi cient Memory Integrity Verifi cation and Encryption for Secure Processors. In *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, pages 339–350, Dec 2003.

[22] 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. Technical report, MIT CSAIL CSG Technical Memo 483, November 2004.

[23] 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.

[24] X. Zhuang, T. Zhang, and S. Pande. Hide: An infrastructure for effi ciently protecting information leakage on the address bus. In *Proceedings of the ASPLOS-XI*, October 2004.