OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms*

Emmanuel Owusu† Jorge Guajardo‡ Jonathan McCune† Jim Newsome†
Adrian Perrig† Amit Vasudevan†

†CyLab, Carnegie Mellon University – *{eowusu, jonmccune, jnewsome, perrig, amitvasudevan}@cmu.edu*
‡Bosch Research and Technology Center, Robert Bosch LLC – *jorge.guajardomerchan@us.bosch.com*

ABSTRACT

We present OASIS, a CPU instruction set extension for externally verifiable initiation, execution, and termination of an isolated execution environment with a trusted computing base consisting solely of the CPU. OASIS leverages the hardware components available on commodity CPUs to achieve a low-cost, low-overhead design.

Categories and Subject Descriptors

C.0 [**General**]: Miscellaneous—*Hardware/Software Interfaces, Instruction Set Design*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Physical Security, Unauthorized Access*

General Terms

Design, Security

Keywords

Secure Execution, Remote Attestation,
Instruction Set Extension

1. INTRODUCTION

Despite numerous attacks against a wide spectrum of organizations [6, 29], secure execution environments protected by TCG have not seen widespread application – even in cloud computing, where customers want to verify execution [2, 41]. Perhaps this lack of application is due, in part, to the lack of end-to-end application software that benefit from TCG properties, lack of trust in the TPM vendors, lack of protection against local adversaries, and concerns over poor performance.

Many designs for an isolated execution environment (IEE) have been proposed, but an interesting question remains: What minimal additions do we need to add to a modern CPU to achieve a highly efficient isolated execution environment with remote attestation properties? This work investigates what minimal architectural changes are required to obtain the essential Trusted Computing Base (TCB) – an isolated execution environment completely contained inside a modern CPU – providing resilience against several classes of hardware attacks. In addition, we design this architecture such that minimal changes to a modern commodity CPU are required for deployment. In keeping with minimalist design, we provide a simple programming interface consisting of few instructions.

Contributions

- We present an instruction set for remotely verifiable, efficient code execution requiring a minimal TCB.
- We propose an API where the CPU provides unique cryptographic keys to security-sensitive applications.
- Our deployment model precludes the need for a distributor or manufacturer to protect platform secrets on behalf of the end-user or their customers.
- Our system is designed for deployment on existing commodity CPUs with minimal modifications.
- Contrary to prior approaches, our solution does not require on-chip non-volatile memory to store secrets. Thus, in addition to avoiding the strong assumption of *secure* non-volatile memory, our solution is cheaper to implement in practice as it leverages semiconductor processes already used in modern CPUs.

2. PROBLEM DEFINITION

2.1 Model and Assumptions

Deployment Model. Our use case defines outsourced computation in the sense advocated by public cloud computing. Thus, we identify three key parties; and their different roles and levels of trust as a device moves from production to use:

(i) The processor *hardware manufacturer* (HWM). The HWM is trusted to manufacture the CPU to initialize a cryptographic device key with a Physically Unclonable Function.

(ii) The *service provider* (or device owner) that offers the device as a platform to customers who wish to lease them for a certain amount of time or computation.

*This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office, and by a gift from Robert Bosch LLC. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, Robert Bosch LLC, CMU, CyLab, or the U.S. Government or any of its agencies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS'13, November 4–8, 2013, Berlin, Germany.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2477-9/13/11 ...$15.00..

Finally, (iii) the *user* (or cloud customer) who wishes to lease computing resources. Users are interested in verifying the trustworthiness of devices leased to them, guaranteeing the integrity and confidentiality of their computations and data.

In the remainder of this paper, we refer to the service provider's device simply as the *platform* or P and to the user's device as the *verifier* or V.

Adversary Model. We assume a sophisticated adversary with physical access to the computing platform. In particular, the adversary can introduce malware into the computing platform (e.g., to compromise an application, the OS, or firmware), has access to the external ports of the platform to physically attach malicious peripherals to P. Similarly, the adversary can probe and tamper with low-speed and high-speed buses (e.g., to eavesdrop on a memory or PCI bus), and/or inject code and/or modify data. However, the adversary cannot perform attacks that require complete unscrutinized access to the CPU for extended periods of time. In particular, this implies that the service provider has organizational procedures in place to prevent attacks, but cannot guarantee the absence of a small set of rogue employees.[1] We consider denial-of-service, side-channel, and fault injection attacks beyond the scope of this paper.

Assumptions. With respect to the service provider, we assume that the CPU on the untrusted platform P is not malicious (i.e., we trust the processor). We assume that this CPU contains a Physically Unclonable Function that can only be accessed through the specified APIs. Additionally, we assume that the CPU has a true random number generator. We assume that the CPU is tamper-resistant. Thus, physical security is not a requirement. Finally, we assume that the verifier V has the correct public key of the provider's platform P.

2.2 Desired Properties

The following list contains the desired properties for OASIS.

- $P1$ **Secure.** We would like the following security objectives to be satisfied:
  - $P1.1$ *Externally Verifiable.* Attestable code execution that guarantees platform integrity, code integrity, launch point integrity, and unmodified code execution on the untrusted platform.
  - $P1.2$ *Key Code Binding.* Ensure that a unique cryptographic key is available to each distinct code module that executes in the isolated environment.
  - $P1.3$ *Program State Binding.* The ability to bind data to code.
  - $P1.4$ *Device Transferability.* The ability to transfer ownership of a chip without exposing the secrets of the previous owner.
  - $P1.5$ *Limited Trust.* The HWM should not have access to any device secrets.
- $P2$ **Economical.** We would like the following economic objectives to be satisfied:
  - $P2.1$ *Low-cost.* No substantial increase of manufacturing cost or complexity (e.g., by requiring non-volatile memory within the CPU).
  - $P2.2$ *Self-contained.* No requirement for additional hardware support such as secure co-processors or TPMs.
- $P3$ **Essential.** We aim for a balanced and simple design:
  - $P3.1$ *Minimal TCB.* On-die isolated execution environment with trustworthy computing primitives entirely within the CPU package.
  - $P3.2$ *Minimal Interface.* Minimal interface with minimal controls, which presents a usable programming abstraction.
  - $P3.3$ *Minimal Setup.* Efficient environment setup where expensive operations are bypassed during repeated invocation.

3. HARDWARE BUILDING BLOCKS

3.1 PUFs, Fuzzy Extractors, and TRNGs

Pappu et al. introduce the concept of Physical Unclonable Functions (PUFs), which are functions where the relationship between input (or *challenge*) C and output (or *response*) p_e is defined via a physical system [15, 35]. The physical system has the additional properties of being random and unclonable. The system's unclonability originates from random variations in a device's manufacturing process, which even the manufacturer cannot control. In their most general form, PUFs can accept a large number challenge-response pairs. Examples of PUF constructions include: optical PUFs [35], silicon PUFs [15, 14], coating PUFs [45], SRAM PUFs [16, 17], reconfigurable PUFs [23], and Flash memory-based PUFs [48].

Because of PUF variability across different environmental conditions (voltage, temperature, humidity, etc.), when a PUF is challenged with C_i, a response p'_e (a noisy version of p_e) is obtained. In applications where the PUF response is used as a cryptographic key this noisy response p'_e is not acceptable. To solve this problem, algorithms known as *fuzzy extractors* leverage non-secret helper data to work around the noisy nature of physical measurements typical of PUF applications [21, 27, 10]. We assume that the fuzzy extractor is implemented in a silicon block and is accessible as a function that is used (in combination with the PUF interface) to realize our instructions.

While stability is fundamental for PUFs, variation in unstable bits can be leveraged for random number generation [17, 44, 48]. For the purposes of this paper, we focus on PUFs based on memory arrays, such as SRAM commonly used in CPU caches. SRAM memory can be used as the raw source for a PUF as well as the entropy source for a True Random Number Generator (TRNG).[2]

[1]For example, a cloud service provider may unintentionally grant datacenter access to malicious [20] or negligent [43] employees.

[2]The new Intel random number generator is based on the instability of a couple of cross-coupled inverters, which are the basic building block of an SRAM cell [44].

3.2 Cache-as-RAM (CAR) Mode

Cache memory is ubiquitous across CPU architectures. Traditionally, SRAM is used to implement a cache. Modern CPUs often include several megabytes of memory on-die which can be leveraged to create a *Cache-as-RAM* (CAR) execution environment [28]. Typically, CAR mode is used to perform system boot-up tasks while DRAM (external to the CPU) is initialized. Prior work has demonstrated that the CPU cache subsystem can be repurposed as a general-purpose memory area for isolated code execution and data read/write operations [46]. The CPU CAR environment offers an isolated execution environment using exclusively on-die hardware.

4. OASIS CPU INSTRUCTION SET

We first provide a high-level overview of the design, describing the requirements, execution model, and implementation rationale for the instruction set extension (ISE) proposed in the paper. The notation used in the remainder of the paper is summarized in Table 1.

Requirements. OASIS is a set of new CPU instructions that aim to enable an isolated execution environment contained entirely on chip by leveraging CAR mode execution, and by creating a secret key only available to the CPU (e.g., derived from an SRAM PUF). OASIS is designed for ease of adoption and deployment with respect to existing computing systems.

A central feature of OASIS is the PUF-derived secret key K_p only available within the CPU and which is used as the root of trust of the whole environment. OASIS is based on SRAM-PUFs [16, 17]. This has several advantages: (i) SRAM is already available on modern CPUs in the form of the cache, (ii) SRAM PUFs need to be powered to create the secret key material, thus, they cannot be read offline making them resistant against scanning electron microscope based attacks, (iii) because of their properties, PUFs are tamper-evident (and in some cases tamper-resistant), a property which other technologies do not offer [16], and (iv) SRAM is manufactured using the standard semiconductor process, which leads to decreased costs when compared to non-volatile memory.

OASIS assumes the availability of external non-secure non-volatile memory. This memory is used to store public helper data as well as state and/or programs. External storage is plentiful and does not further complicate the OASIS design since no special security guarantees are assumed. In particular, alterations to the public helper data can be easily detected through the use of robust fuzzy extractors [5, 9].

Root-of-Trust Instantiation. The SRAM-PUF response, p_e, serves as a unique cryptographic secret which is used to bootstrap a unique device identity, per-application encryption and authentication keys, and random number generation. The resulting key material is unique not just per physical device, but per device owner. The SRAM-PUF response is used to derive the secret root key, K_p, which never leaves the processor and is never directly accessible by any party (including any software running on the processor).

The PUF-derived secret root key, K_p, enables the derivation of a key hierarchy as follows. The device owner derives a key (K_{po}) unique to themselves and the device via a key derivation function (KDF), which accepts as inputs an owner supplied seed, S_o, and the PUF-derived secret root key, K_p. This master processor secret, K_{po}, can then be used, in turn, to derive symmetric keys for bulk encryption, authentication, and asymmetric operations. The details are provided in Section 4.1.

Table 1: Notation used in Instruction Set and Protocol

Notation	
`hw_inst[]`	hardware instructions that make up the OASIS programming interface are denoted using a fixed-width font
`f_hw_func[]`	hardware functions are only accessible by OASIS hardware instructions and are denoted using a fixed-width font identifier starting with the letter `f`
$y \leftarrow x$	the value of x is assigned to variable y
$\perp$	this symbol is used to denote a failed platform operation
$x\|\|y$	concatenation of x and y
$x.param$	returns parameter $param$ of variable x
$x.*$	data element formed by concatenating all parameters of variable x
$A \rightarrow B : \langle m \rangle$	A sends message $\langle m \rangle$ to B
$r \xleftarrow{R} \{0,1\}^\ell$	assigns a random integer of ℓ bits to r
K_X	party X's symmetric key
K_X^+, K_X^{-1}	party X's public and private asymmetric key pair
$\{P\}_K$	the resulting ciphertext of plaintext P encrypted using key K
$H(x)$	cryptographic hash function with input x
$\texttt{Enc}_K(P)$	encrypt plaintext P using key K
$\texttt{Dec}_K(C)$	decrypt ciphertext C using key K
$\text{KDF}_K(x)$	key derivation function of key K and non-secret parameter x
$\text{MAC}_K(x)$	message authentication code of x under key K
$\texttt{Sign}_{K_X^{-1}}(m)$	sign message m with party X's private key K_X^-
$\texttt{Verify}_{K_X^+}(m, \sigma)$	verify signature σ on message m using party X's public key K_X^+
$\texttt{Cert}_y(x, K_X^+)$	certificate issued by y that binds the identity x to the public key K_X^+

All keys are stored inside the CPU in a set of special purpose cache registers ($CR.*$) which are only available within the OASIS environment and only accessible by the OASIS instructions. Table 2 lists the keys stored in $CR.*$. Observe that the root key, K_p, is only used for the derivation of the master processor secret. More importantly, the entire key hierarchy is based on an owner seed (S_o), enabling personalization and device transferability.

ISE Overview and Flow. Next, we describe how the PUF-based root-of-trust is used to enable the desired security objectives of Section 2.2 by defining five new instructions: `init[]`, `create[]`, `launch[]`, `unbind[]`, and `bind[]`.

We distinguish between three stages in the life cycle of the CPU. The first stage is performed by the hardware manufacturer (Figure 1(a)). After manufacture, the HWM initializes the master processor key K_p by calling `init[]`. The output of this operation is helper data h_e and a hash $H(p_e, h_e)$,

Table 2: Variables used in Instruction Set and Protocol.

Hidden Variables: values accessible by processor	
p_e	Raw PUF response
K_p	Root key generated from PUF
S_o	Secret seed value set by platform owner
p^*,q^*	Primes corresponding to an RSA private key
$CR.K_{po}$	Master platform secret for a specific owner seed
$CR.K_{po_auth}$	Platform key for authenticating data from untrusted storage
$CR.K_{po_encr}$	Platform key for encrypting data before transfer to untrusted storage
$CR.K_{po_code}$	Platform key used to derive code specific keys
$CR.S_{po_bind}$	Platform binding secret used to derive asymmetric binding keys
$CR.K^{-1}_{po_bind}$	Platform private binding key, derived deterministically from $CR.S_{po_bind}$
$CR.PCR$	Platform configuration registers
$CR.K_C$	Unique cryptographic key for code $\mathbf{C}'$
Visible Variables: values accessible by software	
$K^{+}_{po_bind}$	Platform public binding key, derived deterministically from $CR.S_{po_bind}$
h_e	Helper data used for noise reduction of p_e
h_{PK}	Helper data used for generating asymmetric keys

which is published and available to anyone using the device. We assume that the function `init[]` can be called only once or a limited number of times to prevent attacks that exploit repeated invocations of the generator function `f_init_PUF[]` (described below) to learn p_e [9]. Given that the HWM does not have control of the PUF response p_e (or by extension K_p as it is derived from p_e), the `init[]` instruction enables the limited trust ($P1.5$), low cost ($P2.1$), self contained ($P2.2$), and the minimal TCB ($P3.1$) properties of Section 2.2.

The second stage corresponds to the set-up of the key hierarchy for OASIS, performed by the device owner (Figure 1(b)). This is accomplished by calling the `create[]` instruction, whose main purpose is to derive symmetric and asymmetric keys. These keys will be used to exchange confidential and authenticated messages between the prover (device owner) and the verifier (user) and to guarantee external verifiability ($P1.1$). The main output of the instruction is a public key, which has been derived from the PUF-based root key K_p and a seed S_o known only to the device owner. This allows for transferability of the platform ($P1.4$) as a new device owner can create his/her own public/private keypair ($K^{+}_{po_bind}, K^{-1}_{po_bind}$) by choosing a different seed S'_o. Furthermore, even though the device owner initiates generation of the public/private keypair, only the CPU can access the private key and thus decrypt messages encrypted with the public key.

The third and last stage corresponds to the execution of code on the device by the user (Figure 1(c)). Users will launch the code to be executed by issuing the `launch[]` instruction. This instruction populates the $CR.*$ registers with the symmetric keys derived from the PUF helper data h_e, the device owner's seed S_o, and the public key informa-

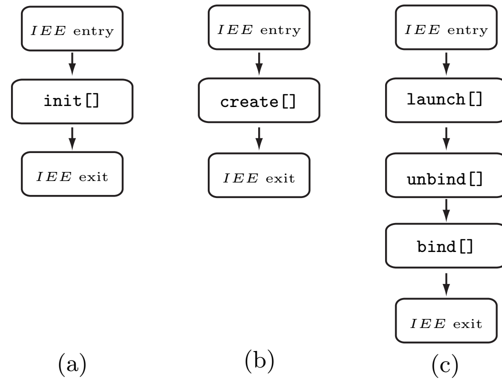

Figure 1: OASIS session during (a) initialization by the manufacturer, (b) setup by the device owner, and (c) code execution by the user.

tion generated using `create[]` in the previous stage. Then, the `unbind[]` instruction can be called to check the input's integrity with respect to a code-specific key and decrypt any input whose confidentiality is preserved by the verifier. The instruction provides two options, one using public-key and one using symmetric-key primitives. The asymmetric option is used the first time the application is called to transmit a secret symmetric key, K_{VP}, only known to the verifier (user) of the platform P. After this initial set-up, the verifier can use fast symmetric-key operations to verify integrity and confidentiality of its data ($P3.3$). At this point the code C can be executed in the isolated execution environment, state is saved (and encrypted if desired), and integrity information is computed on the state using `bind[]` ($P1.3$). Finally, all OASIS memory and internal registers are cleared out, and control is returned to the OS. Observe that any program can in principle be executed in a secure environment using these last three instructions, providing for a minimal and simple programming interface ($P3.2$). Furthermore, the `bind[]` and `unbind[]` instructions, together with the key hierarchy derived with the help of `create[]`, enable external verifiability ($P1.1$) and program state binding ($P1.2$), not only to a particular program but also to a specific CPU, a property unique to OASIS.

4.1 OASIS Functions and Instructions

We describe the functions and instructions used in the design of OASIS. We make a distinction between functions (which are only *internally available* to instructions) and instructions, (which are *externally available* for call by executing software).[3] In practice, functions and instructions might be implemented as digital logic, integrity-checked firmware, microcode, or another process-specific mechanism.

4.1.1 Function Descriptions

We have omitted *explicit* pseudocode definitions for several functions where the specific implementation is left to the hardware manufacturer. Table 3 lists these functions. The functionality of `f_read_PUF[]`, `f_init_PUF[]` and `f_fuzzy_extract_PUF[]` is briefly discussed next.

[3]Instructions and functions are denoted using a fixed-width identifier. Functions begin with '`f_`'. See Table 1.

Table 3: **Hardware Manufacturer Implemented Functions**

p_e	$\leftarrow$ `f_read_PUF[]`
$h_e, H(p_e, h_e)$	$\leftarrow$ `f_init_PUF[`$p_e, rand$`]`
K_p	$\leftarrow$ `f_fuzzy_extract_PUF[`$p_e, h_e, H(p_e, h_e)$`]`
p, q	$\leftarrow$ `f_find_primes[`S_{po_bind}`, RSAParam.size]`
$K^{+}_{po_bind}$, $K^{-1}_{po_bind}$	$\leftarrow$ `f_rsa_key_gen[`p, q, e`]`

The function `f_read_PUF[]` does not accept any inputs; it simply outputs the raw PUF response p_e. We provide two functions to interact with a (robust) fuzzy extractor [5, 9] as is common in the literature: (1) The function `f_init_PUF[`$p_e, rand$`]` accepts a raw PUF response p_e and a random value $rand$ and outputs helper data h_e and a hash $H(p_e, h_e)$. The helper data h_e can be used to reconstruct a uniformly random value K_p from a noisy raw PUF response p'_e. The hash is used to guarantee that only values of K_p constructed with the original helper data h_e are used for further processing in OASIS. (2) The function `f_fuzzy_extract_PUF[`$p'_e, h_e, H(p_e, h_e)$`]` accepts a (noisy) raw PUF response p'_e and helper data h_e and outputs a uniformly random value K_p which can be used as a cryptographic key. The function `f_fuzzy_extract_PUF[]` checks for correctness in the value of $H(p_e, h_e)$ and outputs a special symbol $\perp$ if the input does not correspond to the computed value. If the output is $\perp$, the instruction calling `f_fuzzy_extract_PUF[]` should take appropriate action. In the case of OASIS, we clear all key registers and abort execution.

We assume the use of existing hardware-supported fuzzy extractor implementations [4, 30]. The functions `f_read_PUF[]`, `f_init_PUF[]` and `f_fuzzy_extract_PUF[]` are only available to the OASIS hardware instructions defined in Section 4 and they cannot be accessed by *any* software directly.

Func 1 $S_{po_bind} \leftarrow$ `f_create_sym_keys[`$S_o, h_e, H(p_e, h_e)$`]`

$p'_e \leftarrow$ `f_read_PUF[]`
$K_p \leftarrow$ `f_fuzzy_extract_PUF[`$p'_e, h_e, H(p_e, h_e)$`]`
Clear p'_e
$CR.K_{po} \leftarrow KDF_{K_p}(S_o)$
$S_{po_bind} \leftarrow KDF_{CR.K_{po}}$("bind")
$CR.K_{po_auth} \leftarrow KDF_{CR.K_{po}}$("auth")
$CR.K_{po_encr} \leftarrow KDF_{CR.K_{po}}$("encr")
$CR.K_{po_code} \leftarrow KDF_{CR.K_{po}}$("code")
if $K_p =\perp$ **then**
  Clear $CR.*$
  $S_{po_bind} \leftarrow\perp$
Clear K_p
return S_{po_bind}

Function 1. This function loads the helper parameter h'_e and the hash $H(p_e, h_e)$ into memory. Next, the PUF is read and the fuzzy extractor is invoked to generate the platform symmetric secret key, K_p. Internally, the fuzzy extractor checks whether the inputs $H(p_e, h_e)$ and h_e correspond with the reconstructed value. A special symbol $\perp$ is output should the values be different.

The key K_p and the (device owner) supplied seed S_o are used to derive the master processor secret, $CR.K_{po}$. The seed value S_o allows the device owner to personalize the processor keys. The symmetric key $CR.K_{po}$ is used for the derivation of four symmetric platform keys: (i) $CR.S_{po_bind}$, the platform binding secret, (ii) $CR.K_{po_auth}$, the platform key used for authenticating data residing in untrusted storage from prior invocations (iii) $CR.K_{po_encr}$, the platform key used for encrypting data and (iv) $CR.K_{po_code}$, the platform key used to derive code-specific keys. In all cases, keys are derived via a KDF, which in turn may use pseudorandom functions (e.g., HMAC, CMAC) as building blocks. Constructions of key derivation functions accepting secret and public parameters are well-known [7, 22]. At the end of the process, the function checks if the fuzzy extractor returned the special symbol $\perp$, which would indicate that either the PUF response was too noisy and therefore it was not possible to reconstruct K_p or $H(p_e, h_e) \neq H(p'_e, h'_e)$. In either case, all OASIS registers are cleared and the function returns the special symbol $\perp$ indicating failure. After the check, K_p is cleared and S_{po_bind} is returned.

Func 2*a* $h_{PK} \leftarrow$ `f_create_asym_keys[`S_{po_bind}`]`

$p, q, \leftarrow$ `f_find_primes[`S_{po_bind}`, RSAParam.size]`
$\left\{ \begin{matrix} K^{+}_{po_bind}, \\ CR.K^{-1}_{po_bind} \end{matrix} \right\} \leftarrow$ `f_rsa_key_gen[`p, q, e`]`
$\{K^{-1}_{po_bind}\}_{CR.K_{po_encr}} \leftarrow \mathsf{Enc}_{CR.K_{po_encr}}(CR.K^{-1}_{po_bind})$
$\tau \leftarrow \mathrm{MAC}_{CR.K_{po_auth}}(\{K^{-1}_{po_bind}\}_{CR.K_{po_encr}}, K^{+}_{po_bind})$
$h_{PK} \leftarrow \left\{ \{K^{-1}_{po_bind}\}_{CR.K_{po_encr}}, K^{+}_{po_bind}, \tau \right\}$
if $S_{po_bind} =\perp$ **then**
  Clear $CR.*$
  Clear h_{PK}
return h_{PK}

Function 2*a*. This function generates the processor asymmetric keys. The `f_find_primes[]` function picks a random seed value of size `RSAParam.size` and begins search until the first prime is found. The process is repeated for the second prime using a new seed value. `f_find_primes[]` returns secret primes, p and q. The function `f_rsa_key_gen[]` takes the primes and a public exponent as inputs and generates the keypair $K^{+}_{po_bind}, K^{-1}_{po_bind}$. Notice that the RSA private key $K^{-1}_{po_bind}$ is composed of p, q, and the inverse of the RSA public exponent mod$\phi(N)$, where $N = p \cdot q$. Methodologies to generate primes are well-understood and standardized [18]. The RSA private key $K^{-1}_{po_bind}$ is encrypted using $CR.K_{po_encr}$, and a message authentication code τ is computed over this value and the corresponding public key $K^{+}_{po_bind}$. Finally, a data store h_{PK}, containing the asymmetric keys and τ, is returned.

Func 2*b* $h_{PK} \leftarrow$ `f_create_asym_keys[`S_{po_bind}`]`

$\left\{ \begin{matrix} K^{+}_{po_bind}, \\ CR.K^{-1}_{po_bind} \end{matrix} \right\} \leftarrow$ `f_ecc_key_gen[`S_{po_bind}`, ECCParam]`
$\{K^{-1}_{po_bind}\}_{CR.K_{po_encr}} \leftarrow \mathsf{Enc}_{CR.K_{po_encr}}(CR.K^{-1}_{po_bind})$
$\tau \leftarrow \mathrm{MAC}_{CR.K_{po_auth}}(\{K^{-1}_{po_bind}\}_{CR.K_{po_encr}}, K^{+}_{po_bind})$
$h_{PK} \leftarrow \left\{ \{K^{-1}_{po_bind}\}_{CR.K_{po_encr}}, K^{+}_{po_bind}, \tau \right\}$
if $S_{po_bind} =\perp$ **then**
  Clear $CR.*$
  Clear h_{PK}
return h_{PK}

Function 2*b*. We describe an alternative implementation of the `f_create_asym_keys[]` (Function 2*a*) using elliptic curves in Function 2*b*. The implementation of this function

is analogous but much more efficient than its RSA counterpart, since there is no prime search step. Key generation is a single elliptic curve multiplication, which in general is efficient. In addition, this version has the advantage of small area overhead, if support for asymmetric operations is implemented at the hardware level. These advantages come at the cost of a significant increase in the time required to perform a signature verification operation (when compared to RSA). It is up to the HWM to decide which implementation is more appropriate based on its own requirements and constraints.

Func 3 $K^+_{po_bind} \leftarrow$ `f_read_asym_keys[`h_{PK}`]`

$\tau' \leftarrow \text{MAC}_{CR.K_{po_auth}} \left(h_{PK} \cdot \left\{ \{K^{-1}_{po_bind}\}_{CR.K_{po_encr}}, K^+_{po_bind} \right\} \right)$

if $h_{PK}.\tau \neq \tau'$ **then**
  Clear $CR.*$
  Clear $h_{PK}.K^+_{po_bind}$
else
  $CR.K^{-1}_{po_bind} \leftarrow \mathsf{Dec}_{CR.K_{po_encr}}(h_{PK}.\{K^{-1}_{po_bind}\}_{CR.K_{po_encr}})$
return $h_{PK}.K^+_{po_bind}$

Function 3. This function is very efficient as it only requires symmetric cryptographic operations. In particular, `f_read_asym_keys[]` checks tag $h_{PK}.\tau$ to ensure that input data has not been tampered with. If this verification passes, the function decrypts the private binding key to $CR.K^{-1}_{po_bind}$, using the symmetric key $CR.K_{po_encr}$. Note that the corresponding read functions, for create functions $2a$ and $2b$, are the same except for the sizes of the operands, outputs, and registers required to store private and public keys.

4.1.2 Instruction Descriptions

Inst 1 $\{h_e, H(p_e, h_e)\} \leftarrow$ `init[]`

$p_e \leftarrow$ `f_read_PUF[]`
$rand \xleftarrow{R} \{0,1\}^\ell$
$\{h_e, H(p_e, h_e)\} \leftarrow$ `f_init_PUF[`$p_e, rand$`]`
Clear $p_e, rand$
return $\{h_e, H(p_e, h_e)\}$

Instruction 1. This instruction initializes the helper data h_e used to de-noise the raw SRAM PUF value p_e. The functions `f_read_PUF[]` and `f_init_PUF[]` read the raw PUF value and instantiate the helper data, as described in Section 4.1.1. The hash value $H(p_e, h_e)$ will be used by later instructions to prevent modified helper data from being used in attempts to learn information about the PUF. Observe that a hardware-generated random number, *rand*, is used to introduce entropy in the resulting helper data's value.

The variable *rand* needs to remain secret and exposed only inside the processor. It is also assumed that h_e can only be set once (or a limited number of times) to prevent exposing the output of the fuzzy extractor. This can be achieved during the initialization, which is performed by the HWM. Because of our use of *robust* fuzzy extractors [5, 9], we do not require *any secure* non-volatile memory. All data is stored outside the chip, either locally or externally published on a website. An additional step, not shown and not performed as part of Instruction 1 is the signing of $h_e||H(p_e, h_e)$ by the HWM or a TTP with output $\sigma_{h_e} \leftarrow \mathtt{Sign}_{K^{-1}_{TTP}}(h_e||H(p_e, h_e))$. This guarantees to any third party (users, system integrators, device owners, etc.) that the helper data was created by the HWM and not some other (untrusted) party. Notice this is done only once during the lifetime of the device.

Inst 2 $h_{PK} \leftarrow$ `create[`$S_o, h_e, H(p_e, h_e), \sigma_{h_e}, K^+_{TTP}$`]`

if $\mathtt{Verify}_{K^+_{TTP}}(h_e||H(p_e, h_e), \sigma_{h_e}) = \mathtt{accept}$ **then**
  $S_{po_bind} \leftarrow$ `f_create_sym_keys[`$S_o, h_e, H(p_e, h_e)$`]`
  $h_{PK} \leftarrow$ `f_create_asym_keys[`S_{po_bind}`]`
  Clear S_{po_bind}
  return h_{PK}
else
  ABORT

Instruction 2. This instruction generates a hierarchy of cryptographic keys from the raw PUF response p_e. Symmetric and asymmetric keys are generated by `f_create_sym_keys[]` (Function 1) and `f_create_asym_keys[]` (Function $2a$ or $2b$), respectively.

The h_{PK} variable is assigned the $\{K^+_{po_bind}, K^{-1}_{po_bind}\}$ keypair generated by `f_create_asym_keys[]`. Observe that h_{PK} is encrypted and contains authentication information, which is verified internally by OASIS using a key derived from the internal PUF key and the seed S_o. Lastly, note that verification of the signature σ_{h_e} is most efficient if the signature algorithm is based on RSA using a small exponent (e.g., $3, 17$, or $2^{16}+1$). Regardless of the latency due to signature verification, we expect that this step is performed rarely – e.g., whenever the device changes ownership or if a user desires to setup the environment for future use.

Inst 3 `launch[`$\mathbf{C}, \mathbf{C}.inputs, S_o, h_e, H(p_e, h_e), h_{PK}$`]`

Configure CPU into CAR Mode
Load $\mathbf{C}$ into the CPU cache
$S_{po_bind} \leftarrow$ `f_create_sym_keys[`$S_o, h_e, H(p_e, h_e)$`]`
$K^+_{po_bind} \leftarrow$ `f_read_asym_keys[`h_{PK}`]`
$CR.PCR \leftarrow H(\mathbf{C})$
$CR.K_C \leftarrow KDF_{CR.K_{po_code}}(H(\mathbf{C}))$
if $(S_{po_bind} = \perp)$ **then**
  Clear $CR.*$
  ABORT
Clear S_{po_bind}
Transfer control to $\mathbf{C}$'s entry point

Instruction 3. The `launch[]` instruction is designed to setup the OASIS environment for code $\mathbf{C}$ and populate all necessary registers. It begins by setting up a clean-slate CAR environment, including disabling interrupts and hardware debugging access. It then reads and loads $CR.*$ registers with cryptographic key material for further processing by other instructions.[4]

[4]A possible optimization is to conditionally invoke `f_create_sym_keys[]` and `f_read_asym_keys[]`. For example, `launch[]` can be modified to only invoke `f_create_sym_keys[]` once after the processor reboots and maintain the resulting keys in CR.* during successive OASIS sessions. This optimization must be carefully considered and constructed by the implementer to manage the security trade-off (PUF-derived secrets persisting between invocations). Addition-

To avoid the expensive operations performed in `create[]` for asymmetric key generation (e.g., prime generation), an encrypted data store h_{PK} is returned by `f_create_asym_keys[]` and `f_read_asym_keys[`h_{PK}`]` is used on subsequent invocations. This function's overhead is equivalent to a few efficient *symmetric-key* operations.

Observe that if we want to make the public binding key available outside the environment, Instruction 2 must be called first. Also note that Instruction 2 verifies the signature σ_{h_e} every time it executes, whereas Instruction 3 does not. We expect that signature verification will be performed at most once *per session*, where each session might call the `launch[]` instruction multiple times. Notice that even if the signature verification function is performed every time, the overhead should be minimal, assuming RSA signatures. Refer to Figure 1 for details on when instructions are called.

Next, `launch[]` stores a hash of the target code $\mathbf{C}$ to the platform configuration register $CR.PCR$. Finally, a symmetric key K_C is generated using a key derivation function based on $CR.K_{po_code}$ and a hash of target code $\mathbf{C}$. K_C is used for encrypting and authenticating the executing code's *state* for local storage to untrusted memory.

At the end of `launch[]`, the following registers have been populated: $CR.K_{po}, CR.K_{po_auth}, CR.K_{po_encr}, CR.K_{po_code}$, $CR.K^{-1}_{po_bind}$, $CR.PCR$, and $CR.K_C$.

Inst 4 $X \leftarrow$ `unbind[`$\{X_1, PCR_ver\}_{K^+_{po_bind}}, \{X_2, PCR_ver\}_{K_C}$`]`

```
if {X_1, PCR_ver}_{K^+_po_bind} ≠ NULL then
    X, PCR_ver ← Dec_{CR.K^-1_po_bind}({X_1, PCR_ver}_{K^+_po_bind})
else if {X_2, PCR_ver}_{K_C} ≠ NULL then
    X, PCR_ver ← AuthDec_{CR.K_C}({X_2, PCR_ver}_{K_C})
else
    X ← ⊥

if CR.PCR ≠ PCR_ver then
    X ← ⊥
return X
```

Instruction 4. Inputs X_1 and X_2 contain data values that should only be released to the code that generated the data. The unbind instruction provides assurance to the verifier that the inputs will only be released to the code with measurement PCR_ver. Note that `unbind[]` can decrypt data encrypted under either of the binding key $K^+_{po_bind}$ or the application secret key K_C.

In the protocol described in Section 5, included in X_2 is a symmetric key K_{VP}, which is generated by the verifier V for bulk encryption of data to be transferred between V and the platform P. Notice that we do not suggest using the public binding key, $K^+_{po_bind}$, for bulk encryption. Instead, symmetric keys should be used for bulk encryption operations and the public binding key for storing bulk encryption keys. This is a common practice used to avoid the performance cost of public key cryptography.

In choosing the asymmetric encryption scheme, some care must be taken to prevent an attacker from using the ciphertext $\{X, PCR_ver\}_{K^+_{po_bind}}$, which is intended to be decrypted only by the code with measurement PCR_ver, to generate a related ciphertext $\{X, PCR_ver'\}_{K^+_{po_bind}}$, which the device would be willing to decrypt for different code with measurement PCR_ver'. To prevent this, the encryption scheme must be *non-malleable* – i.e., an attacker cannot use one ciphertext to generate a second ciphertext that decrypts to a plaintext related to the original plaintext. The formal definition of non-malleable is known as Chosen Ciphertex Attack of type 2 or CCA2. Examples of CCA2 (non-malleable) asymmetric encryption schemes include RSA-OAEP and RSA-OAEP+ [39].[5] An alternative strategy to using a non-malleable public-key encryption scheme is to use the secret encrypted with the asymmetric primitive to derive two keys: an encryption key and a MAC key. The MAC key should be used to compute a MAC over the bulk-encrypted ciphertext, and the receiver should reject ciphertexts with an inconsistent MAC. This is the strategy used in the Integrated Encryption Scheme [38]. In this work, we simply assume that we are using a CCA2 public-key encryption scheme regardless of its particular implementation.

Inst 5 $out \leftarrow$ `bind[`$K_{VP}, stateOS, hashInputs, resultV, update$`]`

```
if update ≠ NULL then
    C' ← AuthDec_{K_VP}(update)
    if C' ≠ ⊥ then
        CR.PCR ← H(C')
        CR.K_C ← KDF_{CR.K_po_code}(CR.PCR)
out.OS ← AuthEnc_{CR.K_C}(stateOS, CR.PCR)
V.hosstate ← H(stateOS)
V.hinp ← hashInputs
V.encK ← AuthEnc_{CR.K_C}(K_VP, CR.PCR)
V.res ← resultV
out.V ← AuthEnc_{K_VP}(V)
Clear CR.*
Clear all state
return out
```

Instruction 5. The `bind[]` instruction prepares data for transfer to the untrusted code. This instruction should be called by the executing code right before returning. Inputs to this instruction include a shared secret K_{VP}, the application state *stateOS*, a hash of application input *hashInputs*, and the application results *results*. The variables *out.OS* and *out.V* are ciphertext to be stored in local memory and sent to the verifier, respectively. Please note that *out.OS* and *V.encK* bind *stateOS* and K_{VP} to the launch point measurement of executing code C. Finally, observe that `bind[]` enables program code $\mathbf{C}$ updates. This is enabled by checking whether the update has been encrypted and authenticated with the shared secret K_{VP} and upon successful verification, updating $CR.PCR$ and $CR.K_C$, accordingly.

ally, the call to `f_read_asym_keys[]` may be skipped for sessions that only require symmetric keys.

[5] Note that it is possible for an encryption scheme to be semantically secure while still being malleable [11]. For example, in a hybrid scheme where RSA is used to encrypt a symmetric key, which is in turn used in a block cipher to encrypt the bulk data, then clearly the last block of the bulk-encrypted data can be modified without changing the decryption of the preceding plaintext blocks. This could allow the attacker to change the specified PCR if it appears at the end of bulk encrypted data. Even if the authorized PCR is at the beginning, the attacker would still be able to modify the end of the bulk data without changing the value of the preceding ciphertext.

5. SECURE REMOTE EXECUTION

Figure 2 shows the protocol for the initial and subsequent executions of security sensitive application $foo()$. We assume that the remote verifier V has a copy of the public platform binding key, $K^+_{po_bind}$. Similarly, the verifier can keep a certificate that is used to confirm the authenticity of the public key it receives from the platform. We also assume that the verifier has access to the plaintext code.

In Step 1, the verifier V initiates an isolated execution session with the platform. V generates an encryption key K_{VP}, and binds the hash of the code $foo()$ with K_{VP}. Bind allows the verifier to encrypt data using the public part of the platform key while ensuring that only the correct code running in a correctly setup execution environment can access the data. The inputs along with the code are sent to the platform.

In Step 2, the OS calls the hardware instruction `launch[]` using the plaintext code $foo()$, the verifier inputs $V.inp$, and the previously stored state $OS.inp$ as inputs. If it is the first code execution $OS.inp$ is empty.

In Step 3, the isolated execution environment IEE first checks inputs received from the verifier. If a "setup" command was received from the verifier the IEE attempts to unbind the encrypted inputs from V as follows. The IEE releases shared encryption key K_{VP}, using the `unbind[]` instruction, and decrypts any private inputs, aborting execution if either operation fails. These checks prevent unauthorized code from proceeding. After the checks, the application logic is executed. For example, if the application is a secure counter, during the first iteration the counter is set to zero. In the case of an encrypted database, the first records could be stored in the database or all records could be initialized to zero.

Steps 4 and 5 show the parameters returned to the OS and the verifier, respectively. Step 5 is critical as it provides evidence to the verifier that the computation has been performed on the correct inputs and, in particular, that the inputs have not been manipulated prior to entering OASIS.

6. DISCUSSION

6.1 Linkable Code Blocks

So far, we have presented how an application C that is fully contained within the CPU cache is executed in OASIS. Recall that the `unbind[]` instruction guarantees that only C can access its protected state during future invocations by verifying that the loaded application has measurement $H(C)$ before decrypting.

Now we consider the case of an application that has size greater than the cache (e.g., application $C = C_0|C_1|\dots|C_n$ where C_i refers to the ith application code block). Execution of more complex applications is achieved by computing a Merkle hash tree over the entire program, and binding the resulting tree's root value to the application state. The loaded code block C_i is accepted if and only if the hash tree validation succeeds.

The hash tree construction provides several nice properties. First, it extends state protection and load-time integrity checking to applications of arbitrary size. Second, it maintains a small TCB. Third, it enables efficient execution because code block C_i may be safely executed before the entire application C has been hashed.

Setup Session

1. V : $V.inp.cmd \leftarrow$ `command`
: $V.inp.pubdata \leftarrow pubInputs$

: **if** ($V.inp.cmd =$ "setup")
: $K_{VP} \xleftarrow{R} \{0,1\}^\ell$
: $V.inp.privdata \leftarrow \mathsf{AuthEnc}_{K_{VP}}(privInputs)$
: $V.inp.encsym \leftarrow \mathsf{Enc}_{K^+_{po_bind}}(\{K_{VP}, H(\mathbf{foo}())\})$

: **else if** ($V.inp.cmd =$ "compute")
: $V.inp.privdata \leftarrow$
: $\mathsf{AuthEnc}_{K_{VP}}(privInputs, outV.hosstate)$
: $V.inp.encsym \leftarrow outV.encK$
$V \rightarrow OS$: $\langle \mathbf{foo}(), V.inp\rangle$

: **else** /* other functionality */...

Launch Code[a]

2. OS : $OS.inp \leftarrow out.OS$
: $\texttt{launch}[\mathbf{foo}(), \{V.inp, OS.inp\}]$

Execute Code

3. IEE : **if** ($V.inp.cmd =$ "setup") **then**
: $ksym \leftarrow \texttt{unbind}[V.inp.encsym, NULL]$
: **if** ($ksym = \perp$) **then** ABORT
: $data_1 \leftarrow V.inp.pubdata$
: **if** ($V.inp.privdata \neq NULL$) **then**
: $data_2 \leftarrow \mathsf{AuthDec}_{ksym}(V.inp.privdata)$
: **if** ($data_2 = \perp$) **then** ABORT
: **else**
: $data_2 \leftarrow NULL$
: $state \leftarrow \mathbf{doWork1}(data_1, data_2)$
: $out \leftarrow \texttt{bind}[ksym, state, H(V.inp), NULL]$

: **else if** ($V.inp.cmd =$ "compute")**then**
: $ksym \leftarrow \texttt{unbind}[NULL, V.inp.encsym]$
: **if** ($ksym = \perp$) **then** ABORT
: $data_1 \leftarrow V.inp.pubdata$
: **if** ($V.inp.privdata \neq NULL$) **then**
: $data_2 \leftarrow \mathsf{AuthDec}_{ksym}(V.inp.privdata)$
: **if** ($data_2 = \perp$) **then** ABORT
: $state_{old} \leftarrow \texttt{unbind}[NULL, OS.inp]$
: **if** ($data_2.Vhosstate \neq H(state_{old})$) **then** ABORT
: $\{state_{new}, res\} \leftarrow \mathbf{doWork2}\,(state_{old},$
: $data_1, data_2)$
: $out \leftarrow \texttt{bind}[ksym, state_{new}, H(V.inp), res]$

: **else** /* other functionality */...

Save State

4. $IEE \rightarrow OS$: $\langle out.OS, out.V\rangle$
OS : store $\langle \mathbf{foo}(), out.OS\rangle$

Verify Execution

5. $OS \rightarrow V$: $\langle out.V\rangle$
V : $outV \leftarrow \mathsf{AuthDec}_{K_{VP}}(out.V)$
: **if** ($outV = \perp$) **then**
: reject: invalid computation
: **if** ($outV.hinp \neq H(V.inp)$) **then**
: reject: invalid inputs

[a] Please note that $OS.inp$ is assigned $NULL$ during the first launch.

Figure 2: *OASIS Execution Protocol:* This protocol shows the interaction between the verifier V and untrusted system OS during the initial invocation (*setup*) and repeated invocations (*compute*) of code foo() within isolated execution environment IEE. During the initial invocation the verifier V uses the public platform key $K^+_{po_bind}$ to establish shared secret K_{VP} which is used for repeat invocations.

6.2 Rollback Prevention

A *rollback attack* occurs when old state is presented to the isolated execution environment. Since the stale state is cryptographically consistent, an isolated execution environment implemented without rollback prevention will incorrectly accept it – potentially bypassing stateful protection mechanisms to, for example, undo the append-only property of an audit log. Thus, rollback resistance is needed to guarantee state continuity of the executing application.

One technique for ensuring state continuity is to include a protected monotonic counter as part of the state [33]. Another technique for rollback prevention is to keep a trusted summary (e.g., a hash) of the expected state. Parno et al. include a summary of the state *history* to permit reverting to a safe state in the case of an unexpected crash [36]. These methods can be achieved by using protected non-volatile memory for persistent storage of data describing the expected state. However, we seek a rollback prevention mechanism that enables OASIS to remain stateless between invocations. Additionally, we rule out using a trusted third party for state management.

What follows is a description of how the verifier can confirm state continuity using the OASIS instruction set. During the execution protocol, the `unbind[]` instruction is invoked to decrypt any state belonging to code C (Figure 2 step 3). After executing code C, the `bind[]` instruction is invoked to protect state destined for the OS as well as output destined for the verifier. Included in the output for the verifier is a summary of the current state, $H(stateOS)$. The verifier output is encrypted under key K_{VP} before transferring control to untrusted OS code for delivery to the verifier. The verifier includes this state summary as an input during the next invocation. If the state presented by the untrusted OS matches the expected state, the code executes and the new state summary is communicated to the verifier as acknowledgment. Otherwise, the protocol aborts. In this fashion, we achieve rollback prevention without requiring persistent application state in the OASIS TCB.

6.3 Distributed Deployment

We have presented cryptographic techniques for data secrecy, authenticity, and freshness. Still, the rollback prevention mechanism described thus far is insufficient if we consider the distributed deployment model where multiple verifiers collaborate through a remote service provider. In this asynchronous context, even if cryptographic techniques prevent forged responses and data snooping, a compromised OS can launch *forking attacks* by concealing the operations of one verifier from another. For example, a compromised server may simply omit the current state and replay an old state to the other verifiers.

Fork consistency ensures that all verifiers see the same operations log before an omission but no verifier can see any other verifier's operations after an omission fault (fork). Furthermore, the fork consistency condition enables the verifiers to detect a misbehaving service provider after a single omission.

Li et al. present a protocol for achieving fork consistency where each verifier maintains a signed version structure list [25]. Each verifier signs increasing version numbers and appends these to their respective lists, allowing them to compare lists and detect a fork attack.

6.4 Version Updating

To support version updating (i.e., updating code $\mathbf{C}$ to legitimate new code $\mathbf{C}'$), the application must implement an update command which calls `bind[]` with parameter *update* set (where the *update* parameter contains the new code version $\mathbf{C}'$ encrypted under key K_{VP}). The `bind[]` instruction first checks parameter *update* for authenticity and then updates $CR.PCR$ and $CR.K_C$ using the new code version $\mathbf{C}'$ (refer to Table 2 for definitions of variables and Instruction 5 for details on `bind[]`). In this way the application state of the current software version $\mathbf{C}$ is bound to the new software version $\mathbf{C}'$. Accordingly, the next invocation of `unbind[]` will release the application state to $\mathbf{C}'$.

The decryption and authentication operations prove to OASIS that the software originated from the verifier V as she is the only one in possession of the key K_{VP}. It is possible to design an alternative update mechanism based on asymmetric operations which has the advantage that an entity different from V can provide an update $\mathbf{C}'$, thus granting it access to the current OASIS state. However, this comes at the cost of requiring certification which would add complexity and computational overhead.

6.5 Device Transferability

Recall that the device owner selects seed value S_o during key generation (refer to Function 1 for details). The seed value S_o enables derivation of owner-specific processor keys. Customization, via the owner-generated seed S_o, precludes previous device owners, including the manufacturer, from generating the same platform secret as the current owner. Thus, the device can be safely transferred. This protects the owners of new devices by limiting the ability of malicious parties (e.g., along the supply chain) to learn the platform secrets of the end user. This allows, for example, a device to be repurposed at a new business unit or sold to a new owner.

Please note that the owner-generated seed S_o effectively disassociates any resulting key material from the device manufacturer. Nevertheless, the owner needs a mechanism to prove the authenticity of their processor to a third party. A default seed value that is fixed for the life of the device may be included to support secure device transfer while still providing a mechanism for proving the authenticity of the executing platform. We refer to this default seed value as the *identity* seed value or ${S_o}^*$. Next, a master signing key is derived from root secret K_p and identity seed ${S_o}^*$. Certification can be handled by a third party for further unlinkability. In this way, secrets linked to the hardware are derived from the fixed identity seed ${S_o}^*$ whereas secrets exclusive to the owner are derived from the custom owner seed S_o.

Allowing the owner to choose any S_o as often as they like may allow an attacker to leak the root platform key K_p through cryptanalysis. This can be mitigated by rate-limiting requests for a fresh S_o. Upon request, the device generates a fresh seed value S_o and computes a MAC over it using a key derived from the root secret K_p and the identity seed value ${S_o}^*$. This ensures that chosen values of S_o cannot be correlated with a response, during device initialization, to learn the root platform key K_p.

7. PERFORMANCE EVALUATION

7.1 System Configuration

We model our proposed processor instruction set using Simics, a full-system simulator [31]. We build a prototype system by adding our new instructions to the x86-hammer model.[6] We model a 2 GHz processor with non-unified L1 cache (64 KB data and instruction caches). We use a modified Linux 2.6.32 kernel as our target operating system.

7.2 Microbenchmark Results

To evaluate micro- and macro-level benchmarks, we measure the performance of our implementation against TCG-style implementations of common security-sensitive code operations. We use a pessimistic benchmark for the OASIS isolated execution environment and compare it to an optimistic benchmark for TCG 1.2. See Table 4 for a list of the platform primitives and their associated costs. See Table 5 for a comparison of performance overheads for OASIS and DRTM-based implementations.[7]

We base the median performance costs associated with the cryptographic primitives by leveraging open source libraries LibTomCrypt and OpenSSL.[8] It is likely that these functions further increase in performance with a hardware implementation.

7.3 Performance Advantages

We now present the performance advantages of our architecture as compared to a TPM implementation.

In terms of processor speed, cryptographic applications benefit from running on a processor core instead of a TPM. For example, the Infineon TPM co-processor operates at 33 MHz, which pales in comparison to even mid and low-end commodity processor speeds.

In terms of communication overhead, we avoid costly communication overheads by implementing cryptographic functions on chip instead of on a co-processor. For example, the TPM interfaces using the Low Pin Count (LPC) bus. The LPC is used to connect low-bandwidth devices to the CPU (4-bit-bus on a 33 MHz clock).

8. RELATED WORK

Architecture Extensions. Hardware-based security mechanisms have been proposed and implemented by both commercial and academic groups. In terms of commercial hardware-based IEE technologies, the main components are the Trusted Execution Environment (TEE) which provides capabilities for isolated execution and ensuring software is in a known good state before launch, and the Trusted Platform Module (TPM) which provides remote attestation, binding, and sealing capabilities. Popular TEE implementations include ARM Trust Zone [1], and Intel TXT [3]. More recently, Intel has improved on the TXT architecture with the development of Intel SGX [19]. These techniques can be combined with the OASIS API. For example, Enclaves from SGX would replace CAR mode based memory isolation to support applications of much larger size.

Table 4: Performance overheads for platform operations used to instrument the OASIS isolated execution environment hardware simulation. Times are based on a 2 GHz processor clock.

	avg (of 2^{10} executions)	
	cycles	*time(ms)*
Platform Support		
$rand \xleftarrow{R} \{0,1\}^{\ell}$	1.6 K	$7.91 \cdot 10^{-4}$
`f_read_PUF`	~	$2.55 \cdot 10^{-5}$
`f_init_PUF`	~	$2.40 \cdot 10^{-5}$
`f_fuzzy_extract_PUF`	~	$3.30 \cdot 10^{-5}$
Crypto		
$H(p_e)$	4.9 K	$2.49 \cdot 10^{-3}$
$KDF_{CR.K_{po}}$	20.9 K	$1.04 \cdot 10^{-2}$
`f_sym_encrypt`	1.2 K	$6.02 \cdot 10^{-4}$
`f_sym_decrypt`	1.2 K	$6.12 \cdot 10^{-4}$
`f_rsa_key_gen`	3.2 B	$1.61 \cdot 10^{+3}$
`f_rsa_encrypt`	3.08 M	$1.54 \cdot 10^{+0}$
`f_rsa_decrypt`	65.7 M	$3.29 \cdot 10^{+1}$
$\mathtt{Sign}_{K_X^{-1}}(m)$	65.9 M	$3.30 \cdot 10^{+1}$
$\mathtt{Verify}_{K_X^{+}}(m,\sigma)$	3.1 M	$1.53 \cdot 10^{+0}$
OASIS Functions		
`f_create_sym_keys`	104 K	$5.21 \cdot 10^{-2}$
`f_create_asym_keys`	3.7 B	$1.84 \cdot 10^{+3}$
`f_read_asym_keys`	18.5 K	$9.26 \cdot 10^{-3}$
OASIS Instructions		
`init`	7.2 K	$3.58 \cdot 10^{-3}$
`create`	4.3 B	$2.16 \cdot 10^{+3}$
`launch`	137 K	$6.84 \cdot 10^{-2}$
`unbind` with asym	68.1 M	$3.40 \cdot 10^{+1}$
`unbind` with sym	17.9 M	$8.95 \cdot 10^{+0}$
`bind`	3.12 M	$1.56 \cdot 10^{+0}$

Similar to our work, Defrawy et al. propose SMART, an architecture for establishing a dynamic root of trust in remote devices [13]. SMART focuses on remote embedded devices (in particular, low-end microcontroller units (MCUs)) whereas we are applicable to high-end processors. Additionally, SMART investigates the usage of secret key material to establish a root of trust, assuming the existence of *secure* non-volatile memory to store the secret. In contrast, OASIS is based on the use of SRAM memory–based PUFs [16, 17].

Previous work has explored hardware extensions designed for an adversary model where software and physical attacks are possible. Lie et al. present XOM, a hardware implementation of eXecute-only-memory [26]. Similar to our adversary model, XOM assumes a completely untrusted OS. Unlike OASIS, XOM assumes a secure manufacturing process, allows secure XOM applications to access the platform secret, and requires secure non-volatile memory. Lee et al. present SP, a processor architecture for isolated execution [24, 12]. Similar to OASIS, SP does not require a secure manufacturing process; however, SP includes no immutable device secret which makes it a challenge to prove the authenticity of the executing platform to a third party.

Memory cloaking provides secrecy and integrity of application data while allowing the OS to carry on most of its memory management tasks by limiting the OS's data access to ciphertext. More recently, Williams et al. (Secure Exe-

[6] *x86-hammer* is a hardware model representing a generic 64-bit AMD Operteron processor sans on-chip devices [47].

[7] We have based performance overheads in Table 5 on TPM benchmarks from [37] where the reference DRTM implementation does not provide performance numbers for 2048-bit RSA operations.

[8] LibTom: `www.libtom.org`. OpenSSL: `www.openssl.org`.

Table 5: **Comparison of performance overheads by invocation scenario.**

Scenario	OASIS		DRTM		
	operation(s)	*time*	*operation(s)*	*time*	*ref.*
One Time	`init[]`	3.6 μsec	NV Write, TPM 2048 Root Key Generation	> 25 sec	[37]
One Time per Owner	`create[]`	2.6 sec	TrustVisor-modeled AIK Generation: TPM and μTPM 2048 AIK Generation	> 25 sec	[37]
Per Module Launch (First Time)	`launch[]` and `unbind[]` with $K^{+}_{po_bind}$ encrypted input	34.1 msec	TrustVisor-modeled DRTM: Transfer SLB over LPC, Unseal μTPM keys, quote SLB, μTPM HV_Quote of PAL	> 1.8 sec	[32]
Per Module Launch (Repeated Invocation)	`launch[]` and `unbind[]` with K_C encrypted input	9.0 msec	TrustVisor-modeled DRTM: Set-up and HV_Quote of PAL	22 msec	[32]

cutables [49]) and Chhabra et al. (SecureMe [8]) propose an isolated execution environment using hardware-based memory cloaking. Secure Executables uses CPU-protected memory regions to store the register set (e.g., while a Secure Executable is suspended during a system call). This solution has the advantage of avoiding cryptographic operations; however, direct memory attacks may be possible (e.g., by a DMA-enabled hardware component). The root of trust in Secure Executables is based on a public/private keypair that is installed in the CPU during manufacturing. In our design, the manufacturer and the device owner (or system integrator) both contribute to initializing a root of trust. This reduces the possibility of any large-scale data breaches and also facilities repurposing the device for new owners. SecureMe improves upon previous cloaking methods by ensuring that the entire address space of the application remains protected at all times. OASIS differs from SecureMe in its usage model. Unlike SecureMe, OASIS enforces isolation in the strictest sense by suspending the OS for the duration of its sessions.

PUF-Based Secrets. Similar to our work, Suh et al. propose a secure processing architecture, AEGIS, that makes use of Physical Unclonable Functions for creating and protecting secrets [42]. AEGIS consigns security-sensitive OS functionality (e.g., context switching and virtual memory management) to a security kernel. However, this approach faces the same problem as the trusted OS model – the resulting TCB can be quite large.

Alternate Deployment Models. Our ISE is inspired by the recommendations of McCune et al. [34] but in contrast to previous approaches that use a TPM as the root of trust, we use a PUF-derived key, integrated within the processor. This integration increases performance and diminishes the possibility of attacks on the buses connecting the platform to the TPM.

We use hardware instructions to ensure strong isolation properties during the execution of self-contained security-sensitive code. Another alternative is to use a special-purpose hypervisor instead of additional hardware instructions. The hypervisor provides a less expensive alternative to hardware instruction set extensions and is significantly smaller than a full OS. Nonetheless, a disadvantage of this approach is that the hypervisor is trusted to enforce memory isolation and DMA protection for executing code and, accordingly, must be included in the TCB.

An alternative to extending functionality to the CPU is to use a secure co-processor [40]. A dedicated TPM is the approach endorsed by the TCG. In terms of manufacturing, this approach has the advantage of decoupling system security from the production of traditional processors. A drawback of using co-processors, however, is a reduction of physical security due to the exposed bus. Additionally, the performance hit due to communicating over the bus is not suitable for minimal TCB execution where sessions are repeatedly setup and torn down.

Alternatively, a co-processor could be included as an IP on a SoC which would provide speed, tighter control, and enhanced security. The motivation for extending the processor ISA rather than a SoC TPM implementation is cost savings.

9. CONCLUSION

Currently, TPM-based solutions have not reached widespread application in security-sensitive contexts, perhaps because TCG solutions lack protection against a more resourceful adversary, lack sufficient properties for end-to-end application protection, lack architectural safeguards against supply-chain compromises, or concerns over poor performance. OASIS offers a stronger degree of protection through highly efficient isolated execution with no hardware dependencies outside the CPU.

We have explored the extent to which minimal modifications to commodity CPUs can support isolated code execution. The ISA extensions explored in this research enable compute service providers and application developers to provide high-security assurance at low cost in terms of platform and software complexity.

10. ACKNOWLEDGMENTS

We are thankful to Olatunji Ruwase, Chen Chen, Yanlin Li, and Siddhartha Chhabra for their insightful discussions and for making valuable suggestions for completing this work, and to the anonymous reviewers for their detailed comments and valuable feedback.

11. REFERENCES

[1] ARM Security Technology - Building a Secure System using TrustZone Technology, 2009. Available at http://infocenter.arm.com/.

[2] The CDW 2011 Cloud Computing Tracking Poll, 2011. Available at www.cdw.com.

[3] Intel Trusted Execution Technology (Intel TXT) - Software Development Guide, 2013. Document Number: 315168-009 Available at www.intel.com.

[4] Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., and Tuyls, P. Efficient Helper Data Key Extractor on FPGAs. In *Cryptographic Hardware and Embedded Systems (CHES)* (2008).

[5] Boyen, X., Dodis, Y., Katz, J., Ostrovsky, R., and Smith, A. Secure Remote Authentication Using Biometric Data. In *Advances in Cryptology (EUROCRYPT)* (2005).

[6] Brian Krebs. Coordinated ATM Heist Nets Thieves $13M, 2011. Available at http://krebsonsecurity.com.

[7] Chen, L. Recommendation for Key Derivation Using Pseudorandom Functions (Revised). NIST Special Publication 800-108, 2009.

[8] Chhabra, S., Rogers, B., Solihin, Y., and Prvulovic, M. SecureME: A Hardware-Software Approach to Full System Security. In *ACM International conference on Supercomputing (ICS)* (2011).

[9] Dodis, Y., Katz, J., Reyzin, L., and Smith, A. Robust Fuzzy Extractors and Authenticated Key Agreement from Close Secrets. In *Advances in Cryptology (CRYPTO)* (2006).

[10] Dodis, Y., Reyzin, M., and Smith, A. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In *Advances in Cryptology (EUROCRYPT)* (2004).

[11] Dolev, D., Dwork, C., and Naor, M. Non-Malleable Cryptography. In *SIAM Journal on Computing* (2000).

[12] Dwoskin, J. S., and Lee, R. B. Hardware-rooted trust for secure key management and transient trust. In *ACM conference on Computer and communications security (CCS)* (2007).

[13] El Defrawy, K., Francillon, A., Perito, D., and Tsudik, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[14] Gassend, B., Clarke, D., van Dijk, M., and Devadas, S. Controlled Physical Random Functions. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)* (2002).

[15] Gassend, B., Clarke, D., van Dijk, M., and Devadas, S. Silicon Physical Random Functions. In *ACM conference on Computer and Communications Security (CCS)* (2002).

[16] Guajardo, J., Kumar, S. S., Schrijen, G.-J., and Tuyls, P. FPGA Intrinsic PUFs and Their Use for IP Protection. In *Cryptographic Hardware and Embedded Systems (CHES)* (2007).

[17] Holcomb, D. E., Burleson, W. P., and Fu, K. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Computers* (2009).

[18] IEEE. IEEE Standard Specifications for Public-Key Cryptography — IEEE Std 1363^{TM}-2000, 2000. Available at www.ieee.org.

[19] Ittai Anati, Shay Gueron, S. P. J. Innovative Technology for CPU Attestation and Sealing. In *Workshop on Hardware Architecture for Security and Privacy* (2013).

[20] Jason Kincaid. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. Available at http://techcrunch.com.

[21] Juels, A., and Wattenberg, M. A Fuzzy Commitment Scheme. In *ACM conference on Computer and Communications Security (CCS)* (1999).

[22] Krawczyk, H. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology* (2010), CRYPTO.

[23] Kursawe, K., Sadeghi, A.-R., Schellekens, D., Skoric, B., and Tuyls, P. Reconfigurable Physical Unclonable Functions – Enabling Technology for Tamper-Resistant Storage. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)* (2009).

[24] Lee, R., Kwan, P., McGregor, J., Dwoskin, J., and Wang, Z. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2005).

[25] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure Untrusted Data Depository (SUNDR). In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2004).

[26] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices* (2000).

[27] Linnartz, J.-P., and Tuyls, P. New Shielding Functions to Enhance Privacy and Prevent Misuse of Biometric Templates. In *International conference on Audio and Video Based Biometric Person Authentication (AVBPA)* (2003).

[28] Lu, Y., Lo, L.-T., Watson, G., and Minnich, R. CAR: Using Cache as RAM in LinuxBIOS, 2012. Available at http://rere.qmqm.pl/ mirq.

[29] Lucian Constantin. One year after DigiNotar breach, Fox-IT details extent of compromise, 2012. Available at www.wired.com.

[30] Maes, R., Tuyls, P., and Verbauwhede, I. Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *Cryptographic Hardware and Embedded Systems (CHES)* (2009).

[31] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer* (2002).

[32] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy (S&P)* (2010).

[33] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for TCB Minimization. In *ACM European Conference in Computer Systems (EuroSys)* (2008).

[34] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008).

[35] Pappu, R. S., Recht, B., Taylor, J., and Gershenfeld, N. Physical One-way Functions. *Science* (2002). Available at web.media.mit.edu.

[36] Parno, B., Lorch, J. R., Douceur, J. R., Mickens, J. W., and McCune, J. M. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy (S&P)* (2011).

[37] Schmitz, J., Loew, J., Elwell, J., Ponomarev, D., and Abu-Ghazaleh, N. B. TPM-SIM: A Framework for Performance Evaluation of Trusted Platform Modules. In *ACM Design Automation Conference (DAC)* (2011).

[38] Shoup, V. A Proposal for an ISO Standard for Public Key Encryption. Version 2.1, 2001. Available at www.shoup.net.

[39] Shoup, V. OAEP Reconsidered. In *Advances in Cryptology (CRYPTO)* (2001). Available at www.shoup.net.

[40] Smith, S. W., and Weingart, S. "building a high-performance, programmable secure coprocessor". *Computer Networks* (1999).

[41] Song, D., Shi, E., Fischer, I., and Shankar, U. Cloud data protection for the masses. *IEEE Computer* (2012).

[42] Suh, G. E., O'Donnell, C. W., and Devadas, S. AEGIS: A Single-Chip Secure Processor. *Information Security Technical Report* (2005).

[43] Symantec. Symantec-Sponsored Ponemon Report Finds Negligent Employees Top Cause of Data Breaches in the U.S. While Malicious Attacks Most Costly, 2012. Available at www.symantec.com.

[44] Taylor, G., and Cox, G. Behind Intel's New Random-Number Generator. *IEEE Spectrum* (2011). Available at http://spectrum.ieee.org.

[45] Tuyls, P., Schrijen, G.-J., Skoric, B., van Geloven, J., Verhaegh, N., and Wolters, R. Read-Proof Hardware from Protective Coatings. In *Cryptographic Hardware and Embedded Systems (CHES)* (2006).

[46] Vasudevan, A., McCune, J., Newsome, J., Perrig, A., and van Doorn, L. CARMA: A Hardware Tamper-Resistant Isolated Execution Environment on Commodity x86 Platforms. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012).

[47] Virtutech. Simics x86-440BX Target Guide, 2010.

[48] Wang, Y., kei Yu, W., Wu, S., Malysa, G., Suh, G. E., and Kan, E. C. Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints. In *IEEE Symposium on Security and Privacy (S&P)* (2012).

[49] Williams, P., and Boivie, R. CPU Support for Secure Executables. In *Trust and Trustworthy Computing* (2011).