SecureCell: An Architecture for Computing with Private Data on the Cloud *

Eric J. Love
UC Berkeley

Soham U. Mehta
UC Berkeley

John D. Kubiatowicz
UC Berkeley

ABSTRACT

We present an architecture called SecureCell that enables secure computation with users' private data in a cloud setting. Specifically, it allows the users to run untrusted third-party applications on top of untrusted operating systems on machines they do not control, while remaining confident that no private data will be leaked by either the applications or the operating system. The SecureCell architecture accomplishes these goals by introducing a "sealed container" primitive, a hardware feature that ensures only code running inside a single address space may see an unencrypted view of that address space's data, and that automatically encrypts and decrypts data as it moves across the container's boundary. We present an initial implementation of the automatic encryption feature and evaluate its performance in an architectural simulator.

1. INTRODUCTION

Two technology trends have contributed to the growing importance of cloud-based computing infrastructures: the increasing availability of inexpensive yet abundant storage offered by services such as Amazon's S3 (CITE), and consumers' desire to adopt mobile devices as their primary means of access to computing resources. The latter are smaller than traditional workstations and laptops, possess less storage space, and cannot sustain intensive or long-running computations due to their underpowered CPUs and limited battery life. Their mobility also makes them more ephemeral in nature; they are easily lost and frequently replaced as new models appear, with the result that their flash-based memories contain at best a transient subset of a user's total data.

These properties of mobile clients make the persistence and availability afforded by cloud storage services very attractive for users who would rather not concern themselves with the details ensuring data durability. At the same time, cheap compute cycles on remote cluster machines provide an opportunity to offload computationally intensive tasks, such as building indices over large sets of files or manipulating high-resolution video, that are impractical on phones or tablets.

However, one major obstacle threatens the viability of these services to meet the needs described above: how can users keep their information private when it is in the hands of others? The gravity of this issue is exacerbated by both the devices and the services: the fact that mobile clients are more tightly integrated into users' lives means that the data they carry is correspondingly more personal, and the rising prominence of data-mining as part of advertising strategies provides a strong economic incentive for cloud service providers to violate their customers' expectations of privacy.

1.1 Achieving Privacy in the Cloud

Previous research has focused on either providing secure storage for data that will be used primarily for computations conducted outside of the cloud setting [16], or on making that storage durable and available in a highly distributed and volatile environment [7, 14]. Much less emphasis has been placed on securing computation with private data, and most of the few existing proposals have either relied on trusted OS kernels (such as SELinux) [11] or moved private computation into sealed [4] virtual machines [12]. OS-based solutions rely on remote attestation techniques to ensure that a cloud server is running a known, trusted kernel before permitting computation with private data. This has the disadvantage of forbidding any flexibility for updates to the system-level software deployed in the cloud infrastructure, and also exposes the entire kernel as an attack surface, increasing the overall vulnerability of the service. Solutions built on top of virtual machines (VMs) are very heavyweight; they require a separate VM instance for each user, which introduces enormous start-up overhead to any compute task, as well as wasting memory, and degrading performance through increased cache-conflicts, TLB misses, and syscall service time. They also require that the hypervisor [1] be completely trusted [3].

But even resolving all of these concerns would not eliminate an issue more fundamental to the nature of cloud computing: what software will users execute remotely, and why should users trust it, alongside the cloud provider, not to

*This was a submission for the CS252 course at Berkeley, and is a snapshot of a project that is still in an extremely primitive state. If you are reading this and are not UCB faculty, you are very likely to be disappointed.

reveal their personal information? It is extremely unlikely that users will continue to confide their increasingly personal data to third-party applications, particularly once those are to be executed on machines outside of their control. We therefore believe that any proposal for secure cloud computation must exclude the user-level applications themselves from the trusted computing base (TCB), but should still allow for the possibility of using these applications to perform rich computations on sensitive information [13]. This belief motivates our threat model, which we describe in the next section.

1.2 Threat Model

A cloud computing setting encompasses the following components

- A user's client device used to create data, send it to the cloud, and initiate remote computation on that data.
- Remote (to the user) servers and storage owned by the cloud provider
- The operating systems software running on the provider's servers
- Third-party applications software run on top of the provider's chosen OS

The first of these–the user's computing device–is nominally trusted since it is physically owned and controlled by the user, who presumably will take care that it does not fall into the hands of those who would pry into his confidential data for malicious reasons. Methods such as full-disk encryption could be used to guard against data leakage in case the device is stolen. Although this paper concentrates mainly on securing computation when it is in the cloud, the methodology we describe could just as easily be deployed on mobile devices for the same purposes. In fact, since, as we shall argue, its memory and energy overhead is less than that of VM-based solutions, it may even represent a powerful alternative to existing methods of securing data against malicious applications and faulty OSs on client computing devices.

The last two items–the OS of the provider's servers, and the applications the user wishes to run on them–are both considered to be untrustworthy. We assume that the developers of any applications the user wishes to run may have maliciously written their code so as to harvest information from the data it has been assigned to work on and send the to an arbitrary location of the developers' choosing. Additionally, the provider of cloud computing resources may have modified the OS kernels of his servers to examine the memory contents of running applications in order to extract information from users' ongoing computations. We explicitly reject proposals that force the provider to use a verifiably-trusted OS on the grounds that the provider should have the freedom to choose whichever kernel (including one with in-house modifications) best serves the needs of his datacenters and customers. Moreover, even a supposedly "trustowrthy" OS may have bugs which accidentally allow information release. This leaves only the hardware itself–the physical CPUs and RAM–owned and operated by the cloud provider. We argue in favor of trusting this (and only this) component to ensure the protection of users' data. It is extremely unlikely that any purveyors of cloud computing services would also have the means to alter their physical compute devices. We also consider it improbable that servers' owners would expend the effort required to eavesdrop on the memory address bus, or initiate arbitrary transactions with the DRAM modules to read or modify their contents, since the cost of doing so will, in most cases, far exceed the expected benefits from using such information for, say, targeted advertising, data-mining, or the fulfillment of their employee's voyeuristic desires. Our proposed scheme therefore would not be appropriate for use with top-secret data concerning national security, where a governmental agency may have both the resources and motivation to conduct a more sophisticated and expensive attack.

More important, however, is the fact that hardware designs are by necessity of more limited complexity and subject to more rigorous verification than software, and therefore present to a potential adversary the most minimal possible attack surface against which to attempt the circumvention of any data protection scheme. Moreover, the extended time required to produce and validate a CPU compared to that required for issuing a software update to an existing kernel is exceedingly vast. A data protection mechanism which relies only on a few minor additions to a CPU whose functionality remains otherwise relatively static will therefore be much more desirable and flexible than any software-based alternative. The purpose of this paper is to propose just such a mechanism, which we call SecureCell.

1.3 The SecureCell Model

2. RELATED WORK

SecureCell is not the first proposal for hardware extensions that enable some form of trusted execution on remote systems, and, indeed, its design draws on ideas from various areas of recent computer systems research ranging from architecture to operating systems and Big Data.

2.1 AEGIS

The AEGIS processor [15] proposed a mechanism to provide "Tamper-evident Computing" via which the application running on a remote system can cryptographically certify the integrity of the results it produces, and can guarantee that none of its intermediate state (memory, registers) has been modified by anything other than its own code. An AEGIS system treats only the CPU itself as trusted, and therefore must encrypt and decrypt all data as it leaves and enters the CPU. It must also maintain a Merkle hash tree over all data stored in DRAM to guard against tampering. The combination of these measures adds a significant overhead to most applications, since every cache miss or eviction must be accompanied by multiple encryption, decryption, and hashing operations. We feel that this level of security is excessive for most users of cloud services, since carrying out the physical attacks necessary to snoop on DRAM contents is too expensive to be offset by the limited financial benefit to providers from doing so. More importantly, AEGIS's central concern is protecting applications themselves, which are

given complete freedom to behave as they wish. We are concerned, however, not with the applications themselves, but with the *data* on which they operate; whereas an application in AEGIS may dispose of its data as it sees fit, applications running in a SecureCell are prevented from distributing information in a form that is accessible to anyone other than the information's owner.

2.2 Overshadow

Similarly to AEGIS, the Overshadow hypervisor [3] represents an attempt to provide applications a guarantee that their contents will remain private even in the face of a potentially malicious OS. Overshadow introduces a hypercall that individual applications can make to enter a special "cloaking" mode. This mode presents an encrypted view of the applications' memory to the OS, while leaving the applications themselves to operate on plaintext data. The Overshadow hypervisor accomplishes this by allowing each machine physical page (MPP) to be mapped into at most one shadow page table (a cache between guest virtual page numbers (GVPN) to MPPNs) entry at any time, even if that MPP appears in multiple guest physical page entries. Whenever an access would cause MPP to be (re)mapped into the shadow page table, the hypervisor checks whether it is currently owned by some other GVPN→MPPN entry, and, if it is and that guest address space is in cloaking mode, the physical page contents are encrypted before access may be granted.

We borrow from Overshadow the very important idea of presenting two different views of the same physical page depending on what entity is accessing it. However, we believe that the design of SecureCell, which does not include a hypervisor or involve hypercalls, affords it the potential to be much more efficient than Overshadow, since encryption, decryption, and permissions checks are performed by the hardware itself rather than by software after a context switch. Our approach forces the memory access control logic into a more minimal and therefore less vulnerable mechanism. Overshadow also has the same deficiency as AEGIS when used to protect users' data, which is that it assume the applications operating on that data are themselves trustworthy; indeed, cloaking mode is entered in Overshadow only when an application issues a hypercall to do so. These applications are free to make arbitrary syscalls, for which Overshadow's per-process syscall "shims" will dutifully marshall all arguments and deliver them in plaintext to the OS. This process is an additional source of performance degradation in Overshadow, since each syscall is now transformed into a sequence consisting of a hypercall to the VMM, a call back into the "cloaked shim" to marshall arguments, another call into the VMM, which transfers control to the kernel, and then a return through the uncloaked shim to the VMM, a pass through the cloaked shim to unmarshall arguments, and another call through the VMM back to the app. We believe syscalls can be handled differently since, by necessity, we cannot allow applications to marshall plaintext data to the OS.

2.3 Critical Secrets in Microprocessors

From Lee et al. [8] we take inspiration for our key management philosophy. They presented a methodology for allowing users to store an unlimited number of keys in publicly-accessible storage by maintaining even the keys themselves in an encrypted form. Their proposed processor would allow such keys to be used only by a specially trusted piece of software called the Trusted Software Module (TSM). Any software can make requests to the TSM to encrypt or decrypt data using specific keys, but only the TSM itself every sees these keys. The TSM operates in a "concealed execution mode" (CEM) that resembles AEGIS in its threat model and treatment of all resources outside the CPU boundary as untrusted and public. In order to make a request to the TSM, an application must have (plaintext) access to the key used to encrypt the root of a tree of keys belonging to a particular user (or other entity) of which only the leaves may be used for encrypting and decrypting actual data.

The idea of allowing encryption keys to be stored in public, but restricting their use to very controlled circumstances is also fundamental to the SecureCell model. However, we impose different conditions than [8] on the use of such keys. Rather than permitting the use of keys subject to the availability of a password for the root of a key tree, we restrict use of keys to the inside of an environment that will remain linked to any keys it uses, and this linkage implies that any data subsequently produced in that environment must be re-encrypted before it exits.

3. SECURECELLS AND SECUREOBJECTS

This section introduces the two main components of SecureCell's mechanism for enabling rich computations with private data on remote systems, the SecureObject and SecureCell.

3.1 SecureObject

A main purpose of any cloud computing system should be to maintain users' data persistently in highly available remote storage. If it is also desirable to permit arbitrarily rich computations on this data while maintaining privacy, then careful attention must be paid to how that data is stored. Pervious proposals for secure data capsules [10] have outlined key characteristics that such a mechanism should ideally have. Maniatis et al. argued that data should be stored as cryptographically secured "objects" that contain, in addition to a main payload of data, some sort of policy describing how and by whom that payload may be read or modified. Furthermore, any modifications to an existing object should be recorded in a log that preserves the history of that object since its inception and reveals which entities have accessed it. Although they do not implement a system capable of realizing these objectives, our proposed architecture (described in Section 4 allows users to work with data formatted as SecureObjects–a format which we developed and which achieves many of the goals outlined in [10].

A *SecureObject* is simply an arbitrarily long extent of data encrypted with some key belonging to its owner. The data is stored in this encrypted form and remains encrypted when it is at rest as well as in transit. It never exists as plaintext except during the middle of some computation that requires it, and even then it only exists as such inside of the SecureCell environment described in Section 3.2. SecureObjects do not contain any explicit code inside of them to check uses against an ACL or enforce a specific kind of log update during writes, but we believe that this is unnecessary if the use of SecureObjects is restricted to an environment that will

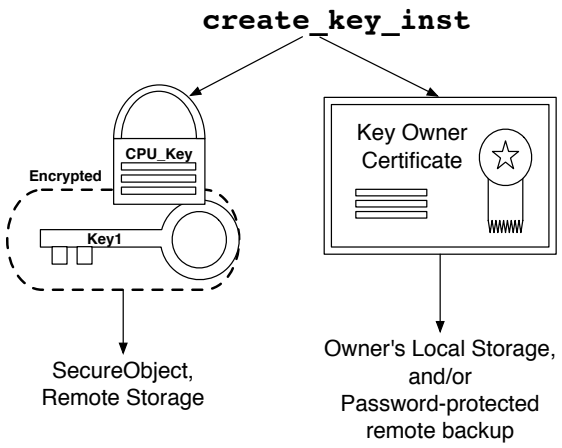

Figure 1: The process of creating a key generates 1.) an encrypted (with the CPU manufacturer's private key) version of the new key and 2.) a certificate representing ownership of that key. The certificate is, again, signed by the CPU manufacturer's key and contains a copy of the generated key to record this correspondence.

not allow release of any derived information unless it is encrypted with the same key as the object's data itself, as is discussed later.

Of crucial importance to the SecureObject data representation scheme is the distinction between having *access* to an object's key, and being its *owner*. Access to a key merely entails possession of a physical copy of the (encrypted!) bits from which the key can be derived in order to use it, temporarily, for manipulating the contents of an object. However, this does not imply that anyone with access to they key can actually gain access to those contents; they can at most request that a particular program be run using the object's plaintext, but the output of that program will not be visible to them.

Instead, only the *owner* of the object's key may see this output. We say that a user *owns* a key if she requested its creation, and, as a result, has in her private possession a cryptographic certificate attesting to her having done so, as is shown in Figure 1. This certificate allows her and only her to extract the plaintext contents of an actively running computation (perhaps as simple as `cat` [object]) and export it to the display or other output channel of her computing device. We call this process *declassification*.

This certificate-based mechanism, by which a trusted SecureCell processor may be asked to permit an ordinarily disallowed (for security reasons) operation when presented with the owner's certificate, can be generalized beyond declassification in order to implement more sophisticated policies that define how a SecoreObject may be used. We encapsulate this idea in the form of a *signed codelet*–a piece of software that is (potentially) trusted by the user to manage her data for her and which is "signed" by the SecureObejct key of which she is the sole owner. This code's responsibilities could include, for instance, remotely determining whether a proposed new version of a SecureObject is properly formatted and obeys a predefined log-structure, or signing that some update represents a write operation selected by a Byzantine quorum of servers as the next step in a total serial order, as was done in OceanStore [7]. We further discuss how to use secure codelets in the next section.

3.2 SecureCell

In order to use SecureObjects with untrusted but behaviorally unrestricted applications, it is necessary to run those applications inside some kind of "sealed container" that prevents them from leaking sensitive information to the outside. We now describe just such a container, which we call a *SecureCell*. The name refers to the cell concept introduced by the Tessellation OS [9], a research operating system developed at the University of California, Berkeley. Tessellation provides the cell as an abstraction for a bundle of resources coupled with strict quality-of-service guarantees, which ultimately isolates the performance of applications running inside one cell from the behavior of those in other cells. This goal is very closely linked with that of SecureCell, since isolation applications from each other's behavior philosophically implies also isolating their data from each other. As such, we expect that future version of Tessellation will provide SecureCell as a fundamental resource container type, though we are careful note that the security guarantees offered by a SecureCell environment are provided ultimately by the hardware itself (as described in Section 4), and are independent of the actions taken by a possibly faulty or compromised kernel.

A SecureCell is fundamentally an execution context with an associated set of keys that determine, together, the set of SecureObjects whose data it can access. This association is, like the cell itself, a dynamic entity; it is not an object that is maintained in durable storage, but more closely resembles a process than a program. Indeed, just as a kernel maintains for each process a table of open file descriptors, so too does a SecureCell contain a list of the keys of objects whose data it has seen. This must be maintained by the hardware, however, rather than the OS, in order to ensure security even when the kernel is potentially malicious.

The SecureCell represents the only location where the data stored as SecureObjects ever exists in plaintext form. Any computation performed on a SecureObject's contents must be conducted inside of a SecureCell, and a SecureCell ensures that no results of this computation may ever leave the cell until they have been re-encrypted with the appropriate key. To use a SecureObject in a SecureCell, an application must:

- Cause a SecureCell to be instantiated (if not already running inside one)
- Make the appropriate requests to the OS to load the object's (encrypted) data into the cell's memory.
- Also ask the OS to load the encrypted key for that object if it is not yet loaded.
- Issue an instruction to the CPU to associate the loaded key with the current cell.

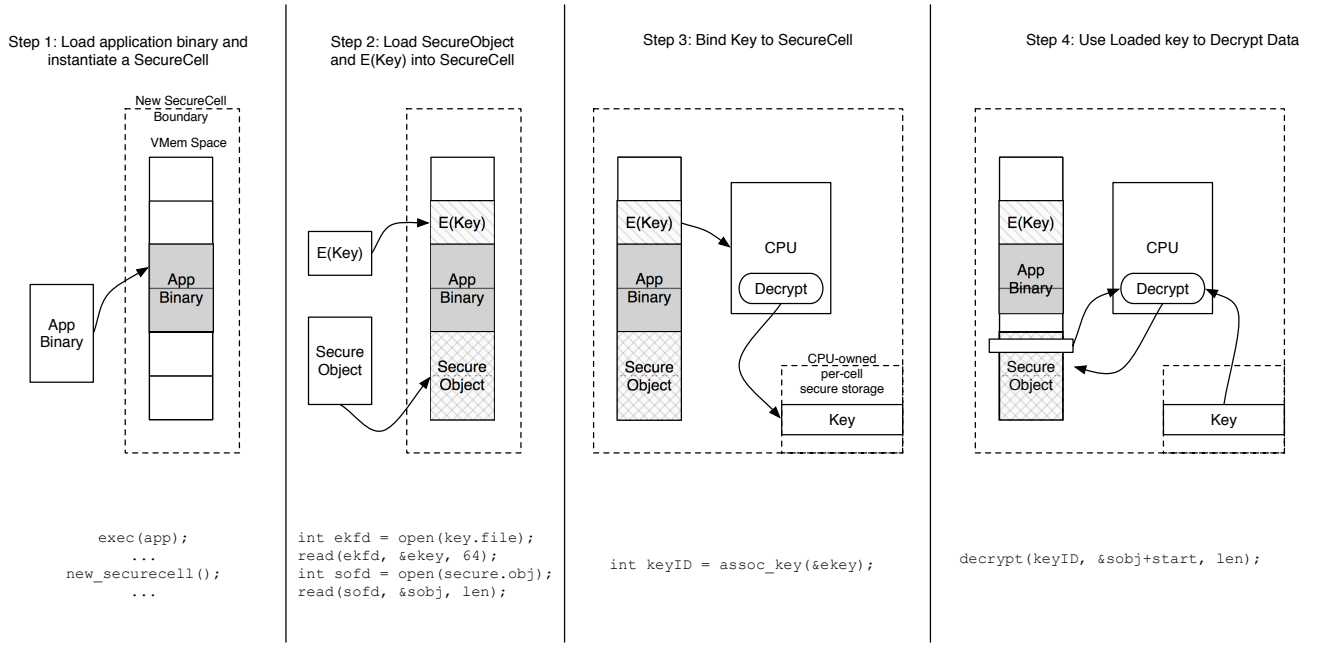

Figure 2: How an application (or several) can use a SecureObject inside of a SecureCell. In the first step, an application is loaded and issues an instruction to the CPU to make a new SecureCell environment, which the CPU must thereafter track and continue to identify as such. In Step 2, the app asks the OS to read files containing 1.) the encrypted capability for the key to the desired SecureObject, and 2.) the encrypted SecureObject itself. In Step 3, the app asks the CPU to bind that key to its SecureCell as a capability. Finally, the app can ask the cpu to decrypt parts of the loaded SecureObject using the associated key.

- Issue further CPU instructions to use the loaded key to decrypt whichever part of the SecureObject's data are needed

The above process is depicted in Figure 2. Once a key has been associated with a cell, it is permanently bound to it for the remainder of the cell's lifetime. If a SecureCell has read data from multiple SecureObjects, then any results it produces must be encrypted with a key derived from the totality of all keys with which it has become associated. Declassification of such results requires access to the certificates of all the relevant key owners–a policy which is essentially equivalent to the labeling schemes employed by information flow tracking operating systems and architectures [6, 17].

4. THE SECURECELL ARCHITECTURE

This section describes the hardware additions and modifications to a traditional CPU that are required to implement the SecureCell execution environment.

4.1 What the Hardware Vendor Must Provide

Much as in the AEGIS system, we require the the CPU vendor embed her private key into the SecureCell CPU so that it may remotely attest to its authenticity as the exact CPU she designed and which SecureObject creators entrust with the safekeeping of their data. This attestation is not an explicit step involving communication between the SecureObject creator's client device and the cloud service provider's servers, but is implicitly performed whenever a key is bound to a SecureCell, since this is only possible when the hardware contains the key for decrypting the object key. Thus, the vendor must provide two things: her private key in sealed storage on the CPU, and her brand name to the user, who must trust it.

4.2 Cells and ASIDs

The SecureCell CPU's mechanism for identifying a particular SecureCell instance closely resembles the address space identifier (ASID) field commonly used in translation lookaside buffers (TLBs) to associate entries with a particular address space in order to avoid flushing all entries on a context switch. In fact, if SecureCells are restricted to contain no more than a single address space, these two items are exactly the same. For this reason, we refer to a SecureCell's ID as an ASID from here onwards.

4.3 Encryption and Decryption

Computation with SecureObjects requires frequent encryption and decryption of various kinds of data. Although number of cryptographic operations performed in a SecureCell is often smaller than in other systems reviewed in related work (especially AEGIS), this still has the potential to degrade performance of applications running in SecureCells. This suggests that hardware acceleration of cryptographic primitives is at least a good idea for reasons of efficiency, but it is actually necessary for security reasons as well. Since we only allow the hardware itself to access (plaintext versions of) they keys used to encrypt SecureObjects' contents, we force all encryption operations to be carried out in hardware

so as to avoid the need for any part of the SecureCell runtime or OS kernel to see such keys at any time. Instead, we simply provide instructions to encrypt and decrypt ranges of virtual memory using keys that have been associated with a cell. The size of these ranges will depend on the particular hardware model used for cryptography, as described below in Section 4.3.2. The performance of this functionality is in our view so important to the viability of the SecureCell CPU that we devote most of our still extremely limited implementation efforts (Section 5) to analyzing the design tradeoffs it affords.

4.3.1 AES Acceleration

In order to achieve a tolerably small amount of overhead for SecureCell-based computations compared to execution on a regular CPU, we assume that cryptographic acceleration hardware is available. This assumption is justified by the presence of such functionality in Intel's SandyBridge and later microarchitectures [5], which offer dedicated instructions for various AES operations, including key expansion and each of the individual AES rounds. This requires software to explicitly fill the AES registers (actually the xmm registers in SandyBridge) with values corresponding to each block, decrypt it, and write decrypted block to memory. We describe below possible variation of this that makes fewer demands on the software (and hence the i-cache and memory system, as well as CPU cycles).

However, even if the SecureCell CPU's hardware were built with the same instruction set extensions as offered by Intel, it would still require some slight differences in available registers, since any registers containing decrypted keys must not be visible to software running on the CPU. The left hand side of Figure 3 depicts such an implementation. The requesting application, rather than loading a key into special key registers and then issuing an expand key instruction, stores a *capability* into a `keyPtr` register, which the SecureCell CPU can use to identify a specific key associated with the currently-running SecureCell instance.

The most straightforward implementation is to have this identifier simply point to the memory address containing the (from the perspective of the calling program) encrypted SecureObject key. Storing this pointer into the `keyPtr` register can then have as a consequence that its target key is loaded fetched from memory, loaded into special internal registers visible only to the CPU itself, decrypted, and subjected to the AES key expansion operation. This is, of course, a long side-effect for an instruction to have, but this is no different from, for example, updating the `CR3` register on an x86 machine and thus causing a page table walk. An optimization could cache decrypted and expanded keys in an associative SRAM indexed by the `keyPtr` register contents. If the CPU has a region of DRAM reserved for its own private use, then sets of expanded keys can be stored to this location during context switches between SecureCells. An even better solution can be found in the coprocessor solution which we will now describe.

4.3.2 Parallelizing Cryptographic Operations

Since SecureObjects are of variable length and possibly extremely large (i.e. many gigabytes), it is sensible to consider them as remaining mostly on disk while only relevant sections of their contents are demand-paged into the SecureCell environment. Nevertheless, just as operating systems attempt to provide minimize the time required to service a page fault for memory-mapped files by engaging in predictive pre-fetching, we can imagine that an intelligent SecureCell runtime library–which is responsible for requesting that data be decrypted prior to its use, as described in Section 5.1–might preemptively fetch and decrypt segments of a SecureObject's data in a similar fashion. This is perhaps only a reasonable behavior when there are extra hardware resources available for parallel execution with the main application, such as unused SMT threads. An ideal SecureCell CPU would allow such resources to be exploited advantageously. Even in cases where such hardware is not unoccupied, an hardware cryptographic unit which does not require numerous explicit instructions from an active thread for each processed AES block will be expected to improve performance and power consumption.

Figure 3 depicts the difference between an Intel-style accelerator (left) and our proposal for an encryption unit that is explicitly parallel with the regular instruction stream(s). The design on the right side includes a dedicated cryptographic coprocessor to which the CPU (i.e. main execution engine, since both this and the coprocessor could be either one "core" or units on a heterogeneous chip) sends requests to decrypt ranges of memory addresses. Each request packet sent on the FIFO contains 1.) the ASID of the calling context 2.) a key capability pointer (`keyPtr` register value) 3.) a starting virtual address, and 4.) a length of the region to decrypt (must be a number of AES blocks).

When the coprocessor receives a request, it pops it from the head of the queue and initiates a DMA operation to gather the request regions of DRAM into its local cache, or, similar to a vector processor, directly into functional units. In this case, the functional units are small cores optimized for AES operations and little else. Because this is a dedicated coprocessor that is designed specifically for cryptographic tasks, we assume that there is enough die area to contain many such cores, as well as an interconnect network to funnel DMA response data directly into them.

There is obviously a tradeoff between the extent of parallel, preemptive decryption and overall energy consumption, and we intend to investigate in our future work how effective this coprocessor might be and how the optimum amount of speculative decryption or encryption effort might vary between different workloads. For now, we present some evidence for the potential efficacy of overlapping decryption with program execution in Section 6.1.

5. IMPLEMENTATION

This section describes the initial steps we have taken towards the evaluation of a prototype SecureCell CPU and accompanying runtime libraries. We must unfortunately admit that we do not at this time have even an "alpha" quality simulation of SecureCell completed, so we cannot hope to present an evaluation of the various effects that our proposed CPU modifications will bear on full-system workloads. Instead, we concentrate solely on the mechanism for bringing data out of SecureObject files and into applications that compute with them. To that end, we describe below our experience

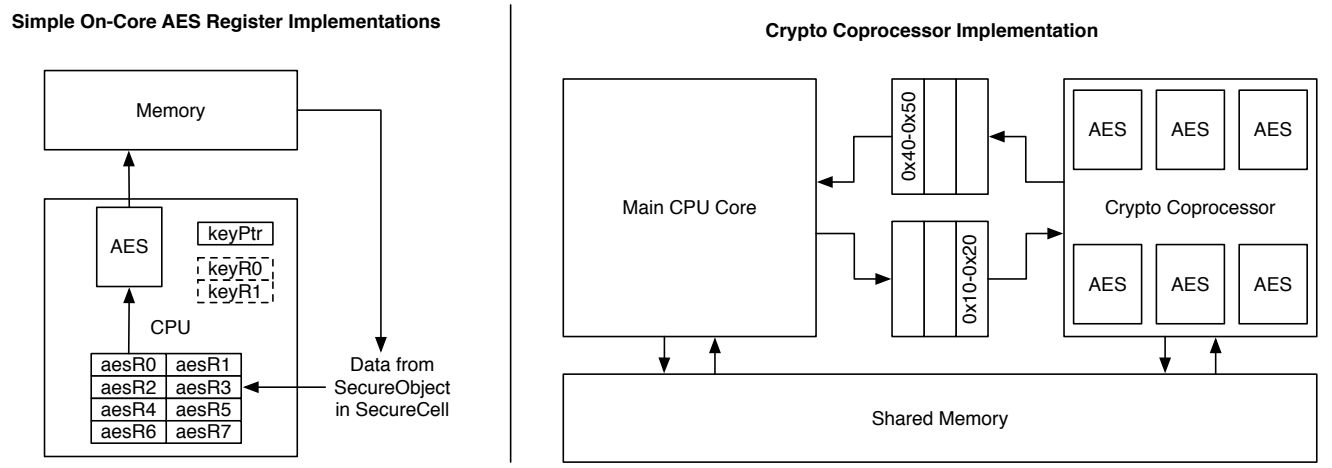

Figure 3: Two implementations of hardware cryptographic primitives. The left hand side depicts an implementation built entirely on the main CPU core using registers for words of an AES block and for an expanded key, and so resembles Intel's AES extensions in SandyBridge. The right hand side depicts a cryptographic coprocessor connected to the main core by two FIFOs: the CPU pushes pairs of (start address, +length) ranges onto the request queue, which the coprocessor then fetches from shared memory, encrypts or decrypts and writes back. As it finishes each range, it pushes that ranges address tuple onto the response queue to alert the CPU that it is finished.

building a CPU model in the GEM5 architecture simulator that allows applications compiled for the ARM ISA to work with rudimentary SecureObjects. We preface that discussion by describing the format of these SecureObjects and the library that manipulates them.

5.1 SecureCell Library (SCLIB)

A key feature of the SecureCell design is that the privacy of a SecureObject's contents, once inside of a SecureCell, is enforced by a minimal set of hardware mechanisms whose only function is to establish a boundary between the inside and outside of a SecureCell. The security of data is therefore independent of the software that requests access to it, or that is responsible for storing and retrieving its encrypted manifestation outside of any SecureCell. This allows for tremendous flexibility in choosing how to implement that software, which could reside in an (untrusted) OS kernel, or be part of an application's runtime environment, much like the C standard library.

We adopt the latter strategy in our demonstration and introduce SCLIB: the SecureCell library. This is a library file against which existing applications may be linked in order to use SecureObject files in our prototype SecureCell CPU. Currently, it implements replacement functions for the standard `open()`, `read()`, `write()`, and `close()` system calls as shown in Figure 4.

5.1.1 Preliminary SecureObject File Format

(mention scutil) Our prototype SCLIB implementation organizes SecureObject files as ordered sequences of *chunks* with a preamble at the beginning to state important parameters for the AES algorithm, as described below. A chunk is a unit of data accompanied by some metadata for cryptographic and organizational purposes. As a result of this

---

Externally-visible functions defined in `sclib.h`:

`int sc_open(const char *path, int oflag, int mode);`
Allocates a new file-descriptor in the SCLIB descriptor table and fills it with a descriptor structure. It then calls the systems actual `open()` function to get a file descriptor to the file at path, which it saves for internal use. Before returning it, reads a preamble from the opened SecureObject file to set parameters such as the AES key size, chunk size and SCLIB version.

`int sc_read(int fd, void *bufPtr, int nbytes);`
Compares the `host_pos` offset of bytes read with the host OS's `read()` call into a buffer with the current `guest_pos` offset at which the calling app thinks it is reading. If the host behind, it asks the OS to read the appropriate data from disk; if it is ahead, we look at the queue of chunks read from the SecureObject. If the head of the queue is already decrypted, start copying bytes into the calling app's buffer. Otherwise, decrypt chunks as they are reached in the queue. Can trigger prefetch operations in future versions when certain behavior is noticed.

`int sc_write(int fd, void *bufPtr, int nbytes);`
`int sc_close(int fd);`

Functions from the SCCRYPT cryptographic function wrapper:

`void *sc_encrypt(SC_CHUNK_DESC *chunk, SC_FD *file);`
`void *sc_decrypt(SC_CHUNK_DESC *chunk, SC_FD *file);`
Both used internally by the SCLIB library functions to encrypt and decrypt individual chunks. Serves as an abstraction to hide which of the of the previously described encryption hardware schemes is in use (or software).

Figure 4: SCLIB Library Functions: these functions serve as replacements for the standard POSIX low-level file I/O functions and give existing applications transparent access to data in SecureObject files.

additional information's appearance in the raw disk data, the calling application may think it is asking to `read()` at a particular offset in the SecureObject's "file" representation, but the underlying OS will actually read this byte at a different location in the raw bits stored on disk. This translation is handled transparently by SCLIB.

Future versions of SCLIB may employ a more sophisticated strategy for arranging chunks in the external (on-disk) representation of a SecureObject. This could be necessary, for example, to maintain a history of all past versions of all chunks in a SecureObject, which may be best represented as an append-only log with hash-verified pointers from new to old versions of a chunk, as was done with extents in Antiquity [16]. Such a layout will make this translation process non-trivial, thus underscoring the importance of allowing arbitrarily complex code to manage the formatting of SecureObject data both on disk and inside a SecureCell.

5.1.2 Buffer Cache and Prefetching

The chunk is the basic unit of I/O in SCLIB and represents the granularity at which it reads and writes data to disk. Since the size of a chunk (8kb presently) may possibly exceed that of a request to `read()`, SCLIB currently stores a pointer to the current chunk in a "read queue" which may buffer buffer excess data that was read in from disk and decrypted during a call to `sc_read()`. A subsequent call to `sc_read()` that uses this data will this be services much faster. In effect, SCLIB transforms `read()` into its buffered standard library cousin, `fread()`.

Future versions of SCLIB could, however, have `sc_read()` prefetch additional data from disk, and begin decrypting it while the calling application continues to use the previously read data. We later describe a hardware mechanism that could allow `sc_read()` return before decrypting data written to the calling applications buffer if that memory can then be marked as not yet ready.

5.1.3 AES Parameters

We choose the Advanced Encryption Standard (AES) as the main cryptographic cipher for the current SecureCell incarnation. AES is a symmetric cipher that operates fixed-length (128-bit) units of data called *blocks* using keys of 128, 192, or 256 bits. Larger key sizes provide stronger levels of security but run somewhat more slowly. AES can additionally be employed in one of several *cipher modes*, the choice of which determines the relation between different blocks in segment of data. The simplest mode, electronic codebook (ECB), encrypts and decrypts each block separately without any information linking any two blocks. Because this reveals, through the encrypted results, if two blocks are the same, other modes, such as chaining block (CBC), can be used instead. CBC maintains a shared state between processing consecutive blocks, with the result that data in one block will affect the encrypted versions of all blocks that follow it. SCLIB currently encrypts each chunk using CBC mode, with an initialization vector (IVEC) for a chunk stored in its surrounding metadata. An alternative to CBC would be to add a random nonce to each AES block, but this would be be an inefficient use of space potentially.

5.2 Simulating a SecureCell CPU

We have used GEM5, a cycle-accurate architectural simulator [2], as our experimental platform for prototyping the SecureCell CPU. It offers several models of CPU for simulation, ranging from a simple model that does not include any timing information and executes as fast as an ISA simulator, to a complex model of an out-of-order core that provides timing measurement at sub-cycle granularity. We have used the "simple timing" model that treats each instruction as having one-cycle latency, but measures delay from memory references that cause caches misses, as a base from which to derive a custom CPU model for SecureCell, which we call SimpleSCCPU.

Of the various target ISAs supported by GEM5, we choose ARM is our working instruction set. However, rather than actually modifying the ARM ISA to introduce new instructions (which would have required also modifying the ARM decoder in GEM5's backend them and an assembler to emit them) we have simulated our ISA extensions by reserving a region the virtual address space for "special" locations to which a load or store becomes equivalent to some target instruction. Such loads or stores execute with a CPI of one just like other non-memory instructions.

We exploit this means of communicating with the CPU simulator in order to allow existing applications to use the SCLIB library without any modifications to the sys call stubs provided by GLIBC for `open()`/etc. Instead, we simply link against the SCLIB binary and insert a call to a special function, `init_sc()` at the beginning of the application's `main()`. This function stores a function pointer, for instance, to the GLIBC `open()` stub into address `0x1000000`:

```
mov r3, #268435456 ; 0x10000000
ldr r2, [pc, #100] ; [pc, #100] contains &open
str r2, [r3] ; store &open at 0x10000000
```

The SimpleSCCPU model as an additional location which, when loaded with 1 by the wrapper `MM_SET_JUMP(1)`, causes an interception of any control instructions that jump to `&open` whereby the SimpleSCCPU changes the branch target PC to `&sc_open`.

Tracking of these special memory words is performed by a `MemTracker` C++ object that interfaces with the simulator and is instantiated on a per-thread-context basis. This object is also invoked by GEM5's sys call emulation mode (which handles system calls on the host system) before `read()` returns to the simulated application. `MemTracker` marks the `read()` buffer contents as "encrypted" since they contain data from an encrypted SecureObject file. The next section describes how this is used to evaluate a potential opportunity for hiding decryption overhead by parallelization.

6. EVALUATION

Our SecureCell CPU simulation is unfortunately not yet complete enough to evaluate the various implementations of an instruction for encrypting or decrypting ranges of memory as described in Section 4.3.2. Nevertheless, we present here a preliminary result from our SimpeSCCPU model that provides an insight into how low overhead from cryptographic operations can be made for oblivious applications.

6.1 TOrTTOU: Identifying a New Kind of Parallelism

When an application's call to `read()` returns, the contents of the application's designated buffer will be filled with call's results, and the application can begin to process those bytes of data. However, if the application performs any reasonably sophisticated manipulation of that data, then there will be a delay of some number of cycles between when the application uses some word of that buffer for the first time and when it accesses others. It is this difference in first use of some part of the buffer that we refer to with the phrase "Time Of `read()` To Time Of Use" (TOrTTOU). TOrTTOU represents a window of opportunity for overlapping decryption with useful computation. If this window is sufficiently large, then it provides space in which to hide the overhead of decryption.

We used the SimpleSCCPU model to investigate the size of TOrTTOU (measured in cycles between the return of `read()` to the time when an individual word of the buffer is first accessed) for a simple benchmark to motivate our ongoing investigation of cryptographic architectures. We cross-compiled the `bzip2` compression tool for ARM, inserted a call to `init_sc()` into its `main()` for use in SimpleSCCPU, and used it to decompress a 32kb test file. The results are shown by the histogram in Figure 5 under the series title "Immediate." The distribution is quite clearly bimodal, with some words being referenced shortly after `read()` and others much, much later. We discovered that much of the hump on the left was the result of copying operations, since execution traces of bzip revealed that each call to `read()` requests that 8kb of data be read into a stack-allocated buffer, and then copied somewhere into the heap.

Since this type of reference does not actually represent a any computationally meaningful usage of data, we decided to investigate how the results this experiment would change if we did not count a load from the buffer as a use, but instead simply marked all the destination registers of the load instruction as belonging to that particular `read()` event. We modified the SimpleSCCPU to propagate this marking through subsequent stores that used loaded buffer data in their source registers, marking destination of the store in memory with this metadata and clearing the markings of any registers that are overwritten by non-`read()`-buffer data.

The results of repeating the experiment with the new tracking procedure are shown in the series called "Follow" in Figure 5. Allowing loads and stores to copy data without counting it as used shifts the distribution strongly to the right, eliminating the left-sided cluster noted previously. A trace examination revealed that the particular load causing the greatest number of "marking propagations" was one in the middle of the C library's `memcpy()` routine, which confirmed our earlier suspicions. It seems that a window of opportunity for conducting decryption in parallel with other useful work is provided by standard library functions that engage in unnecessary double-buffering and copying of data, though exploiting such a larger window may require special hardware to keep track of which words of memory have been decrypted already and which have not.

6.2 Issues with Implementation and GEM5

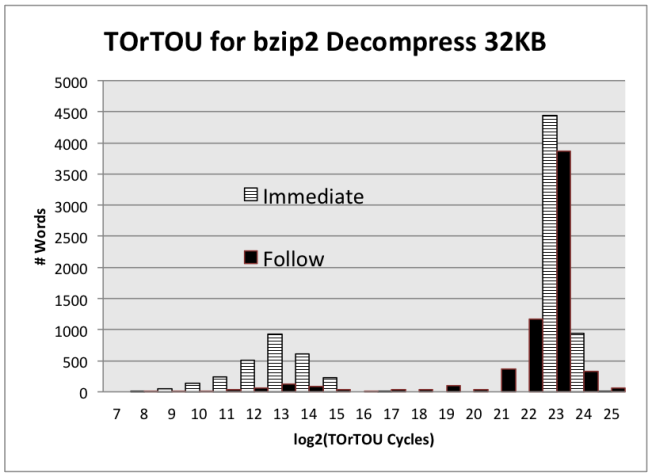

Figure 5: Measuring Time-Of-read()-to-Time-of-Use for the bzip2 program on a test file. "Immediate" presents the results when any access to a word of the buffer after read() counts as a first use, while "Follow" allows load and store operations on a particular word that is not counted as used until an arithmetic operation is performed on it.

We wanted to conclude this section with some statistics about how much runtime overhead our combination of SCLIB and SimpleSCCPU has for bzip2 and other benchmarks, but we have been thwarted in this attempt by two factors: our baseline version of SCLIB with a software implementation of AES (i.e. call to OpenSSL's libcrypto) could not run in GEM5's syscall emulation mode because it requires access to `socket()` which that mode does not yet emulate. Although this should theoretically work in the full system mode, our SimpleSCCPU model does not yet run correctly in this mode. Secondly, our implementation of simulated hardware cryptographic primitives, while underway, is not yet complete and cannot be used for measurements.

7. CONCLUSIONS & FUTURE WORK

This paper has presented a high-level overview of the SecureCell architecture, a minimal set of hardware mechanisms that allow for arbitrarily rich computations with private data using untrusted applications that are guaranteed to be unable to leak any sensitive information even in the presence of a colluding or compromised OS. We have described an format called SecureObject for storing private data in the cloud and examined how it can be used in SecureCells. Although our implementation is, by any reasonable metric, still in its infancy, we hope that the future measurements will show that the performance impact of SecureCell is minimal compared to that of other schemes for secure computation. In any case, we offer the novel ability to guarantee the privacy of data even when applications, in addition to system operators and OS kernels, are possibly malicious.

Much evaluation work remains to be performed, but even our high-level discussion of SecureCell's design is incomplete. For example, we do not yet have an answer to the question of how to handle system calls from inside of a SecureCell. If an appellation has the choice between some number of such

possible calls, does this not represent a leak of arbitrary bits of data? Moreover, how can the application pass useful data as arguments if all application state must be encrypted before it is seen by the kernel?

Moving to a multicore setting will introduce additional difficulties for a SecureCell design. Care must be taken to ensure that, for instance, two cores belonging to different SecureCells cannot send inter-processor interrupts to each other, as this would result in a sharing of data. There will also be a question of how to effectively multiplex new hardware resources, such as the coprocessors described earlier, among different cores, or whether it is best simply to duplicate such functionality for every standard core. We hope to address these and other issues in our future work.

8. ACKNOWLEDGMENTS

We would like to acknowledge Krste Asanović for his guidance during the course of this (still ongoing!) work. We also thank Mohit Tiwari and Emil Stefanov for sharing with us some of their as-of-yet unpublished work, from which we take some of the inspiration for our threat model and motivation.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 42(2):2–13, Mar. 2008.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, Oct. 2003.

[5] S. Gueron. *Intel Advanced Encryption Standard (AES) Instructions Set*. Intel Corporation, 2010.

[6] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, Oct. 2007.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, Nov. 2000.

[8] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. *Computer Architecture, International Symposium on*, 0:2–13, 2005.

[9] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: space-time partitioning in a manycore client os. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[10] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are?: secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[11] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for mapreduce. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[12] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[13] D. Song, E. Shi, I. Fischer, and U. Shankar. Cloud data protection for the masses. *Computer*, 45(1):39 –45, jan. 2012.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[15] 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 17th annual international conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[16] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.*, 41(3):371–384, Mar. 2007.

[17] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.