RC25369 (WAT1304-026) April 12, 2013
Computer Science

IBM Research Report

SecureBlue++: CPU Support for Secure Executables

Rick Boivie
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Peter Williams*
Stony Brook University

*Currently at MIT Lincoln Laboratory

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

SecureBlue++: CPU Support for Secure Executables

Rick Boivie Peter Williams

April 11, 2013

Abstract

To protect software and data against vulnerabilities and malware, we describe simple extensions to the Power Architecture for running Secure Executables. By using a combination of cryptographic techniques and context labeling in the CPU, these Secure Executables are protected on disk, in memory, and through all stages of execution against malicious or compromised software, and other hardware. We show that this can be done without significant performance penalty; additionally, our transparency-focused approach maintains ease of software deployment. Secure Executables can run simultaneously with unprotected executables; existing binaries can be transformed directly into Secure Executables by re-linking. Moreover, Secure Executables can safely make use of system calls provided by an untrusted operating system. In sum, we show that a simple set of processor modifications suffices to provide secure execution in an untrusted environment, without significant changes to the executable.

1 Introduction

Almost every week, we hear of incidents in which systems are compromised and sensitive information is stolen in an Internet-based attack, with victims suffering significant financial harm. A selection of recent startling examples includes [2, 3, 8, 12, 14]. While we do not believe a single solution can solve the entire problem, part of the comprehensive approach required is to focus on protecting the sensitive information itself. This protection should come in the form of a simple, robust barrier against potential attackers. Moreover, we believe that to be broadly applicable to today's software industry, such a solution needs to work transparently with existing software, given the prevalence of legacy software and the expense of developing new software.

Given the evident difficulty in creating bug-free operating systems that reliably protect software, hardware support should be employed to minimize the trusted code base. In particular, to achieve a meaningful protection boundary around the sensitive information, the operating system must be outside the trusted base. On the other hand, we believe a small set of architecture

extensions is acceptable if they realize guarantees of software protection while avoiding significant performance penalty. Existing work (described below) has looked into this approach; we will identify and attempt to solve the set of issues still preventing these approaches from revolutionizing the security industry.

In summary, a dire need exists for systems and methods to prevent the theft of information from a computer system in an Internet-based attack. In particular, this includes the need to protect sensitive data and software on a computer system from other software, including software that an attacker may be able to introduce into a targeted computer system. We take a fresh look at hardware-based isolation techniques in this paper, providing a developer-transparent solution for achieving strong protection of sensitive software.

We describe an architecture that protects the confidentiality and integrity of information in an application so that other software or hardware cannot access that information or undetectably tamper with it. In addition to protecting a secured application from attacks from outside the application, the architecture also protects against attempts to introduce malware "inside" the application via attacks such as buffer overflow or stack overflow attacks. Finally, a strong root of trust is established to guarantee confidentiality and attest to the integrity of results. In sum, this architecture provides a foundation for strong end-to-end security in a network or cloud environment.

The architecture, called SecureBlue++, builds upon the IBM SecureBlue[9] secure processor technology. SecureBlue has already been used in tens of millions of CPU chips to protect sensitive information from physical attacks. In a SecureBlue system, information is "in the clear" when it is inside the CPU chip but encrypted whenever it is outside the chip. This encryption protects the confidentiality and integrity of code and data from physical probing or physical tampering.

SecureBlue++ builds upon SecureBlue. Like SecureBlue, we continue to protect against physical attacks. SecureBlue++, though, uses "fine-grained" SecureBlue-like cryptographic protection that also protects the confidentiality and integrity of information in an application from the *other software* on a system. Moreover, it does this in a way that is transparent to applications.

1.1 Overview

We provide a trusted execution environment for individual applications, protecting an application from other software, *including privileged software* such as the operating system and malware that manages to obtain root privileges. The set of hardware changes described below provides this strong isolation in a way that is nearly transparent. In particular, we can run Secure Executables side-by-side with unprotected software, with only a small performance impact to the Secure Executable, and no performance impact while running unprotected software. The software source code requires no changes to build a Secure Executable; we simply link with new initialization code and system call handlers. The idea is that the CPU uses encryption and integrity checks to prevent the OS from reading or tampering with application memory. At the same time, it

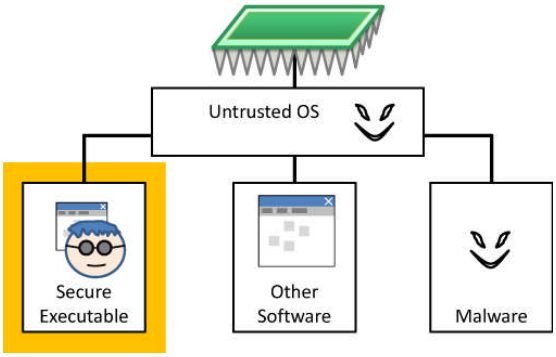

Figure 1: Overview: the Secure CPU provides a boundary of protection around the Secure Executable, protecting its confidentiality and integrity from other software, malware, or a compromised operating system.

enforces some basic semantics of system calls that allow applications to safely make use of them, even when they are implemented by an untrusted OS.

We show the full set of required hardware changes is limited in scope, and convenient for integration into mainstream CPUs. We begin now with a brief summary to provide the intuition behind how sensitive applications are protected at all stages—on disk, in memory, in cache lines, and in registers. A "Secure Executable" is an executable file, encrypted (aside from the loader) so that only a specific target Secure-Executable-enabled CPU can access it. The included cleartext loader sets up the memory space and enters secure mode, enabling the CPU to decrypt the rest of the executable. Confidentiality and integrity of the encrypted portion of the Secure Executable in memory are protected by keys not available to software. Cache lines are stored in cleartext, but associated with a Secure Executable ID that restricts access of the cache line. Reads and writes to locations in the CPU cache incur no new overhead aside from verifying the cache label. Finally, on interrupts, the contents of the registers are protected from the operating system by moving them to protected memory. Figure 1 provides an overview of our approach.

1.2 Related Work

There is a long history of using hardware features to isolate software from a compromised operating system, starting with XOM [11]. Figure 2 compares Secure Executables to related work.

XOM. XOM [11] is a set of architecture changes that extends the notion of program isolation and separate memory spaces to provide separation from the OS as well. They introduce "secure load" and "secure store" instructions, which tell the CPU to perform integrity verification on the values loaded and stored. For this reason, the *application transparency* is limited: developers must tailor a particular application to this architecture.

	SCPU (IBM 4764)	Flicker	SP Secret-Protecting Architecture	XOM	Aegis	Overshadow	Secure Executables
Requirements							
Works without Hardware Changes	N	✓	N	N	N	✓	N
No OS in Trusted Code Base	Card OS in TCB	✓	✓	✓	✓	Host OS in TCB	✓
Works without OS Support	N	✓	N	N	N	✓	N
Transparent to Developers	N	N	N	N	N	✓	✓
Protection							
Code privacy (transparently)	✓	N	N	N	✓	N	✓
Resilient to Memory Replay Attacks	✓	✓	N	N	✓	✓	✓
Protection from Physical Attacks	✓	N	✓	✓	✓	N	✓
Features							
Multiple Simultaneous Instances	✓	N	N	✓	✓	✓	✓
Multi-threading / Multi-core support	✓	N	N	N	N	✓	✓
Support Shared Memory Regions	✓	N	N	N	N	N	✓
Virtualization Support	N	N	N	N	N	✓	✓

Figure 2: Comparison with existing work

Aegis. Aegis [15] fixes an attack in XOM by providing protection against replay attacks. This requires using an integrity tree to protect memory. The authors also offer optimizations specific to hash trees in this environment that greatly reduce the resulting performance overhead. These optimizations include batching multiple operations together for verification, and delaying the verification until the time where the values affect external state. Again, the approach is not transparent to developers: the set of sensitive code areas must be explicitly specified by the developer. Additionally, the developer must identify the specific calculations requiring expensive integrity checkpoints, which are the only places tamper checking is performed. In contrast, SecureBlue++ maintains integrity guarantees across all code sections and computations within the Secure Executable.

Secret-Protecting Architecture. Secret-Protecting Architecture [10] provides a mechanism to run code in a tamper-protected environment. The authors design in [5] a mechanism extending a root of trust to supporting devices. However, this technique does not have the transparency of other techniques. For example, there can be only one Secret-Protected application installed and running at a given point in time. The ability of protected applications to make use of system calls is likewise limited.

TPMs. The Trusted Platform Module, widely deployed on existing consumer machines, can be used to provide guarantees that the operating system has not been tampered with. It does not exclude the operating system from the trusted code base, however. Instead, the TPM signs the current memory and register state, and it is up to other parties to decide if this signed value constitutes a valid operating system state.

A separate class of approaches, including Flicker, uses the TPM late-launch feature to protect software from a compromised operating system.

Flicker. Flicker [13] provides a hardware-based protection mechanism that functions on existing, commonly deployed hardware (using mechanisms available in the TPM). As with other solutions, this incurs only minimal performance overhead. It provides a protected execution environment, but without requiring new hardware changes. The trade-off is that software has to be specifically developed to run within this environment. The protected environment is created by locking down the CPU using TPM late-load capabilities, so that the protected software is guaranteed to be the only software running at this point. System calls and multi-threading in particular do not work for this reason. Moreover, hardware interrupts are disabled, so the OS is suspended while the protected software is running. Thus, software targeted for Flicker must be written to spend only short durations inside the protected environment. The advantage of Flicker is the reduced hardware requirements: it is supported by existing TPM-capable processors.

Overshadow. Overshadow [4] provides guarantees of integrity and privacy protection similar to AEGIS, but implemented in a virtual machine monitor instead of in hardware. This approach has several other advantages as well; making the implementation transparent to software developers—for example, software shims are added to protected processes to handle system calls seamlessly. This means, however, that there is no protection provided against a malicious host OS or against physical attacks. Nonetheless, the authors provide an updated approach that incorporates techniques we can adapt to our hardware-based-protection model. One example is the use of system call wrappers, which transparently manage system calls from a secured application, otherwise unmodified.

1.3 New contributions

We detail several novel contributions in this paper, addressing components missing from much of the related work. Moreover, we believe these features need to be provided by any secure architecture that will achieve widespread adoption. See Figure 2 for a comparison with existing work.

- minimal size of trusted computing base (TCB). In SecureBlue++, the TCB consists of just the secured application and the hardware. Relying on the security of any other components makes it much more difficult to obtain reasonable security guarantees.

- minimal changes to hardware and software, including minimal changes to OS. Instead of defining a new hardware architecture altogether, we provide a small set of changes that can be applied to existing architectures, without affecting other software running on the system.

- transparency to applications. Developers do not need to design programs any differently in order to make Secure Executables. An existing program

can quickly and easily be rebuilt into a Secure Executable (not even requiring recompilation from source code). We believe this is key to enabling widespread adoption.

- transparency with respect to build tools. There are no language changes required to take advantage of Secure Executables, and we are compatible with existing compilers and link editors. We do introduce a new linking step that links system call wrapper code in with an existing application, and a final step to encrypt the application and enable Secure Mode.

- protecting confidentiality and integrity of sensitive information. We encrypt the Secure Executable binary, so sensitive information is protected even in the file system before execution. The architecture establishes a root of trust guaranteeing the integrity of a Secure Executable, ensuring its sensitive information is readable only by the target CPU, and only during execution of that Secure Executable.

- shared memory. We provide a mechanism allowing seamless sharing of memory regions between Secure Executables. Furthermore, as long as values stay in the cache, this sharing avoids the need for any cryptographic operations.

- multi-threading. We support multiple threads (again, seamlessly) for a single Secure Executable. This is a critical feature if the architecture is to be used by modern day applications.

- virtual machine support. Our architecture is compatible with both type-1 and type-2 virtual machines, and can be used to establish a root of trust all the way up to the physical CPU.

We now describe the precise security guarantees we wish to achieve, and the architecture changes necessary to achieve them.

2 Model

A user wishes to run a particular application (the **Secure Executable**) with integrity and confidentiality guarantees. The target system is simultaneously executing untrusted software, including a potentially untrusted or compromised operating system. This paper develops a set of extensions to the CPU running on this untrusted system that make these integrity and confidentiality guarantees achievable. The specific guarantees are revisited again at the end of this section.

We employ standard cryptographic constructions, using hashes, symmetric encryption, and public-key encryption to maintain the integrity and privacy of information. We do not specify the particular constructions to use at this point, however; potential candidates include AES, and RSA with 1024 bit keys. Since the security of our mechanisms rely on these cryptographic mechanisms,

we assume our adversaries are unable to defeat the employed cryptographic encryption primitives.

We consider an attacker who has full control of the software running on the system, aside from the Secure Executable. The protection of memory regions established at launch leads naturally to enforcement of no-write and no-execute regions, which is useful for discouraging buffer overflow and stack writing. However, we do not otherwise protect the Secure Executable from itself: vulnerabilities in the Secure Executable will remain exploitable. We significantly reduce the size of the trusted code base—instead of trusting the entire OS, developers merely have to trust the code that they have control over. We believe a Secure Executable can be much smaller, and consequently more easily verifiable, than an entire operating system.

The attacker may also have physical access to the system, and we protect against attacks to this memory, disk, buses, and devices. However, we assume that it is beyond the abilities of the attacker to successfully penetrate the CPU boundary. That is, the gates and registers inside the CPU are not considered subject to tampering or leak. We believe this is a reasonable assumption for most scenarios and adversaries. However, in situations where it is not, there are existing physical protection techniques, such as those employed on the IBM 4764 [1] that can be used to increase the cost of physical attacks against the CPU. These techniques typically result in increased CPU cost and/or decreased performance due to limited heat transfer capacity.

It is beyond the scope of this paper to consider side channel attacks such as power analysis and timing attacks. We note, for example, that the addresses of the accessed memory locations are revealed to the operating system, which is necessary to allow it to service page faults. There are existing techniques which can hide the access patterns, such as Oblivious RAM [7]. However, existing access-pattern-hiding techniques are slow, and our goal here is to provide an inexpensive, broadly applicable solution.

The mechanism we provide consists of a set of changes to a CPU architecture (in this paper we address the POWER architecture specifically, but the techniques can be adapted to most modern CPU architectures). Four new instructions enable execution of a Secure Executable, under which decryption and execution of protected code is possible, while outside tampering or snooping is simultaneously prevented.

2.1 Integrity and confidentiality guarantees

We consider here informal definitions of the desired security guarantees.

Application Integrity. A polynomially-bounded adversary has control over all values read from external memory and all software on the system. An architecture provides *application integrity* if the adversary's ability to tamper with the code or data of a secure executable (that is, to get the Secure Executable to operate out of the range of behaviors allowed by its source code), taken over the possible choices of Secure Executable secret keys, is negligible.

Defining confidentiality is more difficult. It might be desirable to provide a confidentiality guarantee such as that a fully malicious adversary gains no advantage at guessing the code or data of a secure executable through physical access to the system, and through specifying all software running on the system besides the secure executable. This could be captured with a game in which the adversary specifies two Secure Executables, observes the system, and guesses which of the two is run.

This is definition is not helpful, however, since any secure executable performing useful computation trivially reveals itself to the adversary if allowed to even output a result or interact with its environment. Moreover, the operating system in any practical construction still has the ability to observe system calls (the inputs and outputs to the Secure Executables), and the pattern of memory locations accessed. A narrower definition is needed, closer to the model we are working within, to capture the notion that an adversary gains no advantage from compromising the operating system.

Application Data Confidentiality. An architecture provides *application data confidentiality* if the advantage of any polynomially-bounded adversary, with control over all values read from external memory and all software on the system, to answer any question about the observed Secure Executable, is negligibly better than the advantage of an adversary that observes only the sequence of system calls, memory access pattern (but not *contents*), file size, inputs, and outputs.

3 Approach

In our design, sensitive information is cryptographically protected whenever it is outside the CPU chip. It is available in cleartext inside the chip but only by the software that "owns" the sensitive information. The sensitive information is decrypted as it moves from external memory into the chip, and encrypted as it moves from the chip into external memory. Integrity values are also checked when information moves into the chip and updated when information moves back out. This approach is illustrated in Figure 3.

A public/private key pair is installed in the CPU at manufacture time, with the public portion signed by the factory. This is used to establish the root of trust, allowing Secure Executable binaries to be encrypted so that only a target CPU can access the embedded code and data.

3.1 New Hardware Logic

Memory encryption. Encryption/decryption is performed between cache and main memory. Cache lines corresponding to the protected region are decrypted as they are brought into the cache, and encrypted as they are evicted.

Integrity tree verification/update. A cache line is verified using the integrity tree when data is brought into the cache. This may require performing additional fetches to make sure enough of the tree is present to perform this

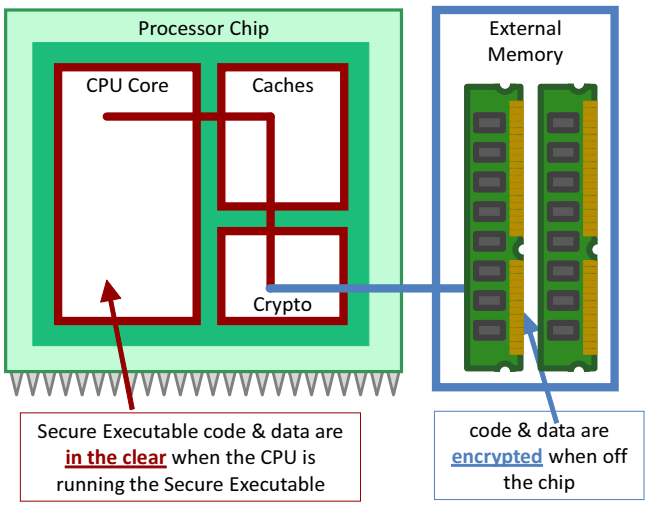

Figure 3: The main idea: sensitive information is cryptographically protected whenever it is outside the CPU chip, and visible as cleartext inside the chip, but only by the software that owns the sensitive information.

verification. When lines are evicted from the cache, the corresponding parts of the integrity tree are updated.

Cache line label. On loads from a protected region, the ID of the Secure Executable associated with that cache line (linked indirectly, though the Memory Region ID described later) is compared to the current Secure Executable ID. If there is a mismatch, the access is treated as a cache miss. On stores, the cache line Secure Executable ID is updated to match the currently running Secure Executable.

Register protection. The CPU ensures that application registers are not visible, or undetectably modified while the Secure Executable is suspended. The two obvious approaches are to encrypt the registers before giving control to the OS interrupt handler, or to store the registers on chip while the Secure Executable is suspended. Both approaches have drawbacks. Encryption takes time, and is potentially vulnerable to replay attacks unless a hash is stored on chip; storing a copy of the registers on chip uses valuable space and severely limits the number of supported simultaneous suspended Secure Executables.

We choose instead to store the register set in protected memory corresponding to each Secure Executable (allocated by the loader code). The registers end up protected in the cache (and ultimately by the Secure Executable root hash value, if these values get evicted from the cache). Only the Secure Executable root hash value, plus the metadata required to use this hash value, are stored on-chip. This eliminates the performance penalty of the cryptographic operations, while greatly reducing the amount of required on-chip storage.

3.2 Software changes

Because software transparency is a main goal, we allow existing applications to be transformed into Secure Executables by relinking. This relink process attaches initialization code (Section 3.3), reserves portions of the address space for the integrity tree (Section 4) and thread restore locations (Section 6), attaches system call wrapper code (Section 7), and reserves a portion of the application memory address space for use during launch as follows.

A metadata region is stored in the Secure Executable application memory space. It is used to register initial parameters with the CPU, as well as establish memory regions. It is encrypted with the Executable Key, and protected by a keyed hash, the initial value of which is loaded on chip with the Enter Secure Mode instruction (described below). Updates to this region require the CPU to re-compute the metadata hash before the next interrupt can be dispatched.

- Metadata region size
- Code entry point
- Signal handler entry point
- Core dump data key
- Location of the protected memory region table (within metadata region)
- Location of the thread restore list (outside metadata region)

The `metadata region size` specifies the total size of this region. The `code entry point` is a 64-bit memory location identifying the next instruction the CPU should execute after running `ESM`. The `signal handler entry point` is a 64-bit memory location identifying the first instruction of the Secure Executable signal handler. This is the only location the OS can jump to in this program; see the discussion of signal handling, under system calls, for more information on the use of this value. The `core dump data key` is an AES encryption key, that if non-zero, will be used by the CPU to encrypt the dynamically generated data keys and export to software when requested by the OS (in case of a core dump). This allows software that has access to the core dump data key to decrypt the protected memory regions in the core dump.

Finally, two more values specify the location of data structures maintained by the helper library linked with the Secure Executable. The protected memory region table keeps track of the ranges of protected memory, and the memory region ID corresponding to each. This fixed-size list is stored on chip every time the Secure Executable is entered (e.g., with the RestoreContext instruction). This is necessary to allow quick translation from memory location to Region ID on memory accesses. The thread restore list keeps track of the re-entry points into the Secure Executable, used with the RestoreContext instruction.

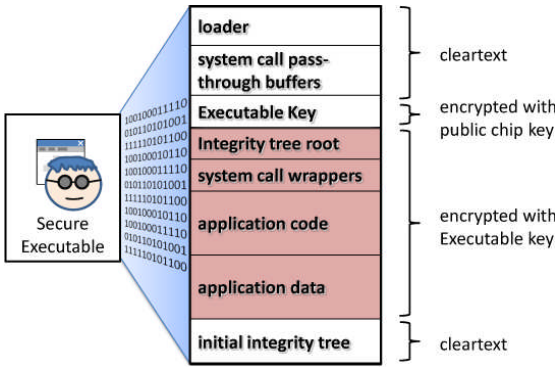

Figure 4: The Secure Executable as a binary file on disk: regions and parameters

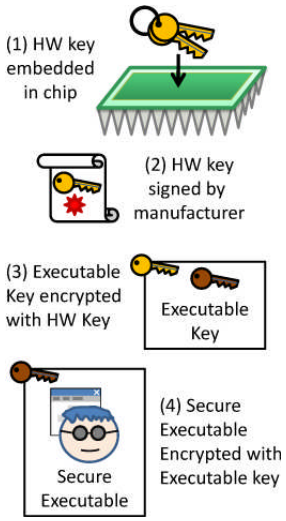

Figure 5: Establishing the root of trust

3.3 Launching

At run time, when the Secure Executable is to be launched, the OS executes the Secure Executable binary, as a normal ELF file. The ELF file consists of a protected region and a cleartext region. The confidentiality of the protected region is protected using a symmetric encryption algorithm; the integrity of the protected region is protected by an integrity tree in the cleartext region. The cleartext region also includes loader code and an ESM instruction. The loader code copies the initial integrity tree into memory. It then issues the `ESM` instruction, which enables decryption of protected regions, and jumps to the entry point (identified in the ESM operand). Integrity and confidentiality protection of memory accesses is also enabled at this point. The ELF application binary format is illustrated in Figure 4.

3.4 Root of trust

The root of trust is established as illustrated in Figure 5. In the factory, a hardware private key (1) is embedded in the CPU. The public portion of this key is signed by the manufacturer, producing a public key certificate (2) asserting that this private key corresponds to a CPU supporting Secure Executables. To build a Secure Executable for this chip, the Secure Executable decryption is encrypted using the signed public key (3). Finally, the Secure Executable code itself is encrypted with this executable key. This ensures that knowledge of the hardware-installed private key is necessary to access the sensitive code and data distributed through the Secure Executable.

4 Integrity Tree

An integrity tree consists of a set of encrypted hash values of memory locations, with these hashes themselves protected by parent hashes. This tree is used to protect memory at runtime. The branching factor of this tree is taken to be the number of hashes that fit in a cache line. For 128-byte cache lines, and 128-bit hash values, for example, each node of the tree protects 8 nodes directly below.

The initial value of the integrity root is signed and loaded with the Enter Secure Mode (`ESM`) instruction, used to start of the Secure Executable. This root ensures that the CPU will only use an intact version of the code; the CPU provides the guarantee that only integrity-verified words are visible to the application in regions of the protected address space.

Ancestors of a dirty cache line are pinned in the cache to avoid updating the tree all the way to the top on each write. Evicting a node from the cache requires updating its parent, which might require evicting another node. By locking all ancestors in the cache, however, it suffices to simply update the parent of a node (assuming it can be brought into the cache). Thus, every dirty line in the cache has all the ancestors pinned in with a reference counter.

This requires considering the case in which a parent cannot be brought in, because all the lines where it could go are already pinned. In this case, the tree computation must be performed immediately, as far up the tree as necessary (until a line whose ancestors all fit is found). The procedure to perform a store from a Secure Executable context is as follows. First, bring ancestors in, starting at top (if they're not already there). Lock each parent when one of its children comes in, by incrementing its reference counter. If a child cannot be brought in, compute the path up to this point, reading and writing all the children directly to/from memory. Then the parent is updated to reflect the new value, and marked as dirty in the cache.

This approach has the property that the CPU can evict a cache line with a zero reference count at any time, with one hash operation but requiring no additional loads or stores. Lines representing instructions or data can always be evicted, as can nodes of the integrity tree whose protected lines are no longer in the cache. Flushing the cache requires starting with all the lines with a zero reference count, evicting these, and repeating as many times as necessary (which can be up to the tree height). On each iteration one level of the integrity tree is evicted.

Due to cache locking conflicts, we do not guarantee that a write requires only updating the parent; it could require up to $\lg(n)$ loads/stores. But this stalls only the current running Secure Executable, and it can be paused and resumed later. This case should be rare, with high enough cache associativity.

This integrity tree needs to be correct over the whole protected address space, even those parts of the address space that have not yet been allocated. Moreover, the parts of the tree covering these unallocated regions should not require initialization, since the protected memory region is potentially much larger than the portion ever accessed. We introduce the notion of a "sparse" integrity tree: one that correctly protects the whole memory region, but is

zero everywhere that it protects unallocated memory. In this sparse integrity tree, these zeros protecting unallocated memory are implicit. Memory is not allocated for these integrity values until they are accessed, at which point they are initialized (by the OS) to zero. Implementing this sparse integrity tree requires a property of the integrity hash function: the hash of an empty region must be zero.

A second requirement for this property of integrity tree correctness over uninitialized data is that the *cleartext* corresponding to uninitialized memory is also zero. To achieve this, we consider all memory locations protected by a hash value of zero to be zero themselves.

4.1 Overcommit

Since we are targeting a 64-bit architecture, the size of protected memory regions can be large. For this reason, we need a simple way to dynamically grow the integrity tree. The Linux overcommit mechanism allows us to reserve room in the address space corresponding to a tree protecting large memory regions, while delaying the physical memory allocation until each page is accessed. In overcommit mode, available in recent Linux kernels, application memory allocations always succeed, even if there is not enough physical memory or swap space to support the entire allocation. That is, pages are not mapped to physical memory until they are accessed. This OS mechanism is particularly useful in 64-bit mode, since the size of the address space we are protecting is potentially much larger than the available physical memory. Without overcommit, our applications would need to specify a limit on the total memory use beforehand, since we must use a much smaller integrity tree (that can fit entirely in RAM or swap space).

Since the hash tree is maintained by the CPU, instead of the software, any solution must involve the CPU as little as possible; the CPU in particular should not be involved in memory management, and should not issue memory allocation requests. Overcommit provides a convenient solution: the Secure Executable loader can allocate a portion of the virtual address space large enough to contain a hash tree over the rest of the address space. With overcommit enabled, this can be much larger than the available physical memory plus swap space. When the CPU attempts to access an address in this range, a page fault is generated, since this address has not yet been assigned to physical memory. Since the virtual address has been reserved by the Secure Executable process (with the mmap syscall), the OS handles this fault transparently by assigning a physical page to this location. This provides exactly the behavior we desire, with no changes to the OS (aside from enabling overcommit).

Downsides to using overcommit.

When overcommit is enabled, processes run the risk of running out of physical memory at any time—and being targeted by the kernel's out-of-memory killer. A potential workaround described here prevents this from occurring. That is, to guarantee that when the machine is out of memory, only new allocations will fail (matching traditional behavior when physical memory is fully

consumed), rather than leaving the potential for any process at all to be killed. The OS provides a new type of memory allocation, specifically for integrity trees. Processes are still allowed to allocate the large integrity tree, but the OS is now aware that the total space used by this integrity tree will stay within a fraction of the rest of the process's used memory. Thus, instead of over-committing to the entire allocation, the OS is only committing to the normally allocated memory plus a constant fraction.

4.2 Integrity tree extensions

The integrity tree provides complete integrity protection for a reasonable performance overhead. However, in certain scenarios the entire range of integrity protection may not be necessary. It may be desirable to have a more limited form of integrity protection at a significantly reduced cost. In particular, by storing keyed checksums of the data, but no integrity tree, we can prevent all integrity attacks except for rollback attacks. This means the only modification an adversary can get away with undetected is restoring a given block of data to an earlier state. For read-only segments of memory this is not a problem. For example, the adversary cannot modify undetectably the read-only code segments since there is no other version of these to successfully roll back to. This is the level of protection that XOM [11] provides; see [15] for an explanation of why XOM protection is vulnerable to such replay attacks.

4.3 Integrity tree optimizations

A more sophisticated layout of the integrity tree can minimize the number of memory pages needed to cover a particular region. The idea is that each page of memory is organized as a local breadth-first tree, containing only nodes under the first entry in the page. The next page begins with the first node (according to global breadth-first order) not yet included. The calculation to determine where in memory a given node is more complicated in this case; we favor simplicity of design by simply using the global breadth first layout.

5 Memory Regions and Shared Memory

The ability to share protected memory regions between Secure Executables is critical. This could potentially be implemented in software using cryptography. However, to achieve transparency and avoid the need to intercept all memory writes, it is still desirable to have hardware support (even if the cryptographic performance were not a concern). There are two issues to consider with respect to sharing memory: how to provide this mechanism transparently to the application, and how to implement it efficiently.

For common cases of inter-process communication, we automatically create the protected shared memory region. One case is anonymous pipes opened before a `fork`: it is apparent these two Secure Executables need to communicate.

The custom system call wrapper sets up the shared memory region over which to implement the pipe. Additionally, on `fork`, the software system call wrapper assigns an MRID (Memory region ID) to all anonymous memory regions. Memory-mapped files cannot be shared securely and transparently without new semantics, however, since it is not known ahead of time what processes and Secure Executables will be accessing this memory-mapped file.

To allow multiple protected regions per application, with various share options, these protected regions are registered at runtime by the applications, by adding an entry to the Protected Memory Region table.

First we consider the hardware implementation of shared memory. Tagging the cache line with the Secure Executable ID of all sharing processes is insufficient, since any number of Secure Executables will potentially share this location. Instead, the cache line is tagged with the Memory Region ID. This `MRID` indexes into an on-chip table containing information allowing efficient verification of the current Secure Executable's permission to access this shared memory region.

We keep only a minimal amount of mapping information stored on-chip in a shared memory region table. This mapping contains two Secure Executable IDs (SEIDs): one for the creator, and one for the first sharer. This allows handling the most common memory region cases (private, or shared among two processes) without penalty. To handle the rest of the cases, a secret is associated with the memory region. Secure Executables are allowed access to this region if the secret is installed at a registered location in the SE memory.

The MRID of the cache line simply points to an entry in the Protected Memory table that indicates both the integrity root and Memory Region secret. Access is verified by checking the `SEID1` and `SEID2` of the Memory Region. Failing this, the secret corresponding to this region is compared at the registered virtual memory location of the current process with the secret in the on-chip memory region table.

The CPU needs to determine whether an access is to a protected region, and to which region. Since this determination needs to be made on every access resulting in a cache miss, the information needs to be stored on chip. We reserve a location in the Secure Executable metadata region to serve this purpose. On every context switch into the Secure Executable, the mapping table linking memory locations to the appropriate `MRID` is loaded into this hardware table.

To protect the integrity of these shared memory regions, Secure Executables share the integrity tree. This is supported by the OS as a standard shared memory region; the virtual pages for each Secure Executable participating in the sharing are mapped to the same physical page.

5.1 Memory Policies: Protecting the Secure Executable from itself

The mechanisms used to register memory regions and support shared memory lend themselves readily to enforcement of no-write and no-execute policies

over specific regions. While modern processes already support page-based access flags restricting writing and execution, they cannot prevent a compromised operating system from tampering with these flags. This, of course, is not an issue in standard processors, because without a Secure Executables approach, a compromised operating system already leads to compromised applications.

Secure Executable support for such policies (e.g. write XOR execute), to protect against attacks such as stack overflow and buffer overflow, includes new parameters to the ESM instruction. These set up the address ranges for an application. One of these will tell the CPU hardware that the address range corresponding to the application's code is, from the application's perspective, "read-only". The other will tell the hardware that the address range corresponding to the application's data, stack and heap is "no-execute". Thus if an attacker attempts to exploit a bug in an application so that the application will attempt to write into the code region or execute instructions from the data region, the attempt will fail.

6 Multi-threading

We must distinguish here between user threads and kernel threads. User threads are implemented in user space; that is, a user space dispatcher maintains and schedules the threads. This means user thread libraries typically do not support true concurrency: only one user thread runs at a time. Kernel threads are instead maintained and scheduled by the operating system. At the price of slightly increased thread switching latency (due to kernel-user context switches), kernel threads allow true concurrency on multi-threaded, multi-core, or multi-chip systems.

User threads succeed in our model without any additional changes, since the previously described protection mechanisms protect the user-space dispatcher compiled into the Secure Executable. Kernel threads require CPU awareness, since this means a given Secure Executable must support multiple simultaneous saved contexts; the OS scheduler is allowed to choose between several available contexts for a given Secure Executable.

Kernel thread creation via the `clone` system call creates a new context restore point equivalent to the current running one, except for a different stack pointer. Our system call wrapper takes care of the bulk of the work in creating a new thread, then registers a new stack pointer with the OS and the CPU. Registration with the OS works as in existing POSIX systems, by using the `clone` system call.

Using part of the application protected memory space to store register restore sections provides a convenient method to support threads. The thread is registered with the CPU by creating a new entry in the protected memory region of this Secure Executable corresponding to the restorable register sets. Then, the OS kernel can switch to this context by providing the memory location of the thread state.

The Secure Executable meta data region contains a pointer to the thread

restore table. This thread restore table contains the following rows:

- Restorable flag, to prevent the OS from reusing the restore point
- Register set (including program counter)

Since threads share a common memory image, all context restore points associated with a given SE share the same integrity root. There is only one entry in the hardware Secure Executable table, covering all threads belonging to the Secure Executable. Thus, the thread count is not subject to hardware limits, while the number of simultaneous Secure Executables is limited to the size of the hardware Secure Executable table. The CPU is not even directly aware of the number of running threads for a given Secure Executable; the OS merely provides it with the restore point, which it validates by checking that it is a valid offset into the thread restore region.

This model works for multi-core CPUs as well; of course, the cores need to maintain coherency. This can be achieved, for example, by enforcing exclusive access to a shared Secure Executable table. Running multiple threads within a SE across multiple CPUs requires a more sophisticated coherency protocol.

7 System calls

Since we prevent any other process from reading values in our Secure Executable's registers or address space, we naturally need a new mechanism to support passing data in system calls. Nevertheless, we support system calls transparently to both the Secure Executable and the operating system. We describe two mechanisms here to support system calls transparently. To make both approaches transparent to the operating system, the CPU leaves the general purpose registers alone on a system call interrupt, so that these registers can still be used to pass parameters to the OS system call handler. Recall that on all other interrupts, the registers are hidden before control is giving to the interrupt handler. For this reason, both approaches move the application registers to protected memory before invoking the system call.

7.1 Approach: System call wrappers

We use this approach in our proof of concept implementation. The application is linked with a customized version of libc, that replaces system calls with wrappers. These wrappers serve the purpose of copying the system call parameters from the protected memory region to the unprotected, OS-readable region.

On invocation, each wrapper performs the appropriate copy, depending on the parameter types. Note that in order to correctly copy the system call parameters, the wrappers have to be aware of each system-call. The wrapper then copies the general purpose registers to the stack, and zeros them, aside from the ones being used as parameters in the system call. Each wrapper then invokes the appropriate system call, and copies the result back into the protected region. Finally, it restores the registers from the stack.

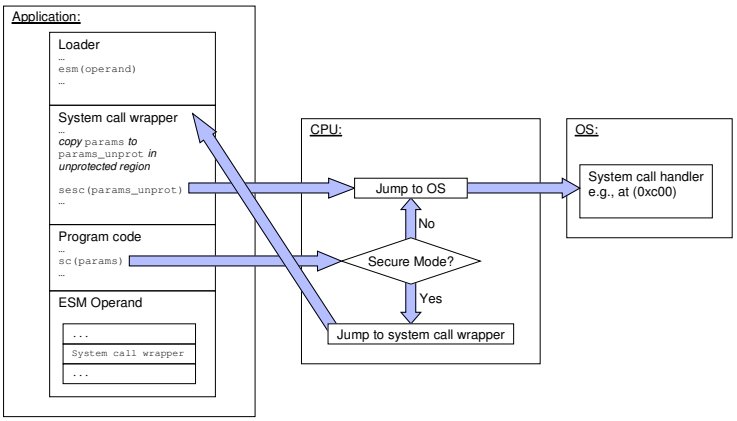

Figure 6: `sesc` (Secure Executable System call) handling. The idea is that a standard system call instruction (`sc`) gets intercepted by the CPU and redirected to the registered Secure Executable software system call wrapper. This wrapper, when ready, issues `sesc`. At this point, the CPU passes control on to the OS system call handler.

Naturally we must consider the privacy and integrity of the parameters while they are in unprotected memory. We necessarily give the OS the ability to read the parameters and choose arbitrary return values. There is nothing we can do to ensure the OS is correctly handling these system calls; thus we leave it to the Secure Executable application code to make the appropriate decisions about the information it passes to the system call. For example, we do not prevent the Secure Executable from printing out sensitive information if that is what it is programmed to do.

7.2 Approach: New system call instruction

This approach allows for better application transparency, by avoiding the requirement for a modified version of libc (and thus, now supporting executables that are not even linked with libc). We still add a layer to the system call invocation, but rather than adding this layer as a set of wrappers sitting between the Secure Executable and its system call invocation, we add the layer after the invocation.

To do this we employ two system call instructions, illustrated in Figure 6. The first, existing PowerPC `sc` (System Call) instruction behaves as today (passing control to the OS system call handler), unless we are in Secure Mode. If we are in Secure Mode, this instruction instead returns control to a previously registered location within the application. A new PowerPC instruction, `sesc` (Secure Executable System Call) gives control to the OS system call handler, *regardless of whether we are in Secure Mode.*

Now, the Secure Executable registers a system call wrapper to the CPU,

so that when `sc` is issued, the system call wrapper gets control. The system call wrapper performs the equivalent memory copies / register cleanup as the previous approach, then invokes the new system call instruction `sesc` to give control to the operating system.

The advantage of the second approach is that we do not have to modify libc (thus eliminating part of the complexity of turning a conventional binary into a Secure Executable). The system call wrapper still needs to be located somewhere in the application address space, and registered by the Secure Executable loader as a parameter to the `ESM` operand. We believe, however, that this address space modification is more application-agnostic than modifying the linked libc. For example, this transformation can be performed directly on a statically compiled binary. Moreover, this modification to the address space and entry point are also performed in other steps as part of the Secure Executable build process.

In both approaches, the CPU needs to keep track of the valid return points; the OS is not allowed to jump to arbitrary positions within the Secure Executable. At any time, the OS can redirect control to one of a fixed set of locations: the program counter associated with a thread restore point, or the registered signal handler (see handling the `signal` call below). In the case a system call is being run (the `InSystemCall` flag is set in the restore point), the OS is also allowed to redirect control to the restore point *plus 4*. These semantics are used in PowerPC to indicate a system call failure. The CPU sets the `InSystemCall` flag when creating the restore point as `sesc` is executed, and clears it after a `RestoreContext` to that restore point.

7.3 Handling specific system calls

The majority of system calls are handled in the straightforward manner described above—the only change required is moving the parameters to and return values from the unprotected region to enable communication with the OS. However, several system calls require special support, e.g., those with side-effect semantics that modify the application memory or registers. We now consider specific cases of system calls.

`read`, `write`, `socketcall`, `mmap`, `sbrk`, etc. No special hardware support is needed for those system calls which the OS can handle without looking inside the application. The calls listed above fall into this class. The wrappers merely move parameters and results from and to the protected region as described above. The wrapper must be aware of the size and type of each argument so that it can perform this transformation.

`signal`. The `signal` system call requires specific hardware support in our model, since the OS is no longer allowed to set the program counter to run arbitrary code in the Secure Executable. We handle the signal system call similarly to Overshadow [4]. This entails registering with the CPU a top-level signal handler as part of the `ESM` instruction operand. When a signal occurs, this application-level signal handler within the Secure Executable then dispatches the signal to the appropriate handler within the Secure Executable. The CPU

then enforces that the OS can only to reset the program counter of a Secure Executable to its top-level signal handler. This gives the Secure Executable protection of the entry points into its code, so that the OS can not invoke code at arbitrary locations.

The `signal` system call wrapper stores the re-entry point in a table, and replaces the entry point in the system call to point to the top-level signal handler. This way, when the OS dispatches the signal, it gives control to the top-level signal handler. The CPU allows this since this is the registered signal handler of the Secure Executable. This top-level signal handler then dispatches the signal to the appropriate code, as indicated in its local table. This approach is transparent to the OS—it simply appears to the OS that the application is using a single signal handler for all signals.

`fork`. Supporting the `fork` system call requires defining the new semantics. When a Secure Executable process forks, we need to decide whether the new process is still considered part of the old Secure Executable, or whether it should be considered a new Secure Executable. Required considerations include how to preserve traditional `fork` behavior; in particular, the new process should begin with a memory space identical to the old process. POSIX semantics also specify several options with respect to what information should be preserved after the fork, and the system call wrapper must be aware of these options. Since we associate each process with a single Secure Executable, the most natural semantics for fork are to create a new Secure Executable on `fork`.

To obtain copy on write behavior, we use OS support. The OS sends a signal, which is handled by library code. It thusly notifies the Secure Executable when it needs to duplicate a particular page. Each copy requires un-sharing both the page and the regions of the integrity tree protecting those pages.

`clone` (Threads) As described in Section 6, the `clone` system call wrapper registers a new thread restore point in memory.

`exit`. To make sure the Secure Executable does not run unintended code after the `exit` system call has been issued, we wrap the `exit` call in an infinite loop in the system call handler. Thus, if control unexpectedly returns from the system call, no other code is run.

7.4 Return value validation

Being in the unique position of not trusting the OS, we need to ensure no new vulnerabilities arise out of the OS feeding unexpected return values from the system calls. The return values of the majority of system calls must be checked for sanity, matched up with the passed parameters as necessary. For example, the number of bytes written back to the Secure Executable after a `read` system call must be within the range specified by the caller. The system call wrapper is in the right position to ensure this, since it runs within the protected region yet sits at the interface between the Secure Executable and the OS. If a violation of expected semantics such as this is detected, the system call wrapper has two options: it can put the response back into the range allowed by semantics if possible, or it can throw an integrity exception. Throwing an integrity exception

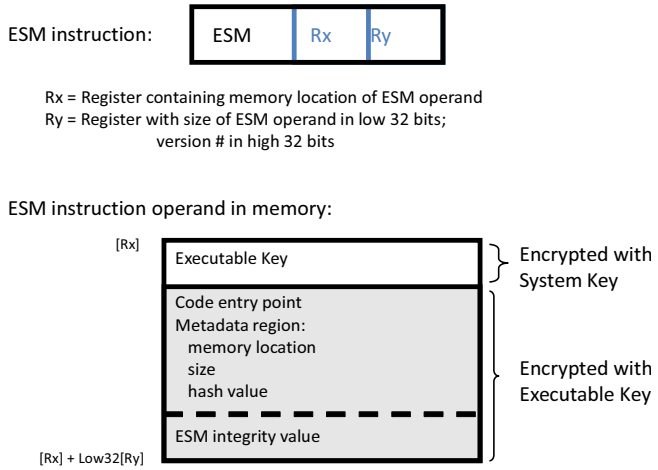

Figure 7: `ESM` (Enter Secure Mode) instruction

is preferable to silently changing behavior since it is more visible and indicates either a bug in the system call wrapper validation or the operating system, or malicious operating system behavior.

8 Hardware changes

Recall that our goal is to provide the hardware mechanisms necessary to build a boundary of protection for sensitive applications. We detail a series of architecture extensions here that enable construction of this boundary in a manner that is transparent to existing applications.

8.1 New instructions

Part of the Secure Executable loading process is to issue the ESM (Enter Secure mode) instruction. The ESM instruction allocates a new Secure Executable ID, while enabling memory protection for the regions identified in the instruction operand.

Enter Secure Mode (`ESM`). The instruction itself identifies two registers, R_x and R_y. R_x points to the memory location of the operand. R_y specifies both the size of the operand, and an operand version identifier (to allow backwards compatibility in the future). The size is in the lower 32 bits of R_y.

The operand, starting at memory location R_x, contains the fields depicted in Figure 7. Executable Key is an AES encryption key, that has been encrypted with the public System Key. The CPU decrypts this value to obtain the Executable Key, which it then uses to decrypt the rest of the operand. The rest of the operand has been encrypted with the Executable Key.

Finally, ESM integrity value is a cryptographic checksum over the cleartext of the rest of the fields encrypted with the Executable Key (starting with the code entry point). This field is encrypted with the executable key, along with the other ones.

RestoreContext. RestoreContext is a privileged instruction issued by the OS to return control to a Secure Executable, comparable to the PowerPC `rfid` instruction. The OS sets the Secure Executable ID Save/Restore (`SEIDSR`) register, described below, to the value that was present when the Secure Executable was interrupted. Then, when the OS issues the `RestoreContext` instruction, the CPU restores the general purpose registers to the state they were in when this Secure Executable was interrupted, resuming execution. The Secure Executable ID (`SEID`) register (not accessible to software) is restored with these registers, so that cache lines owned by this Secure Executable are once again accessible.

DeleteContext. Another privileged instruction, `DeleteContext` instructs the CPU that the Secure Executable indicated in the `SEIDSR` register has finished running, and the CPU can free any CPU resources that were allocated to this Secure Executable. Additionally, the CPU clears all cache lines owned by this Secure Executable.

The OS should issue this instruction when the Secure Executable process exits; however there is no vulnerability resulting from the OS choosing to issue this instruction at other times. If the OS issues this before the Secure Executable process exits, all protected values relating to this Secure Executable become inaccessible (since the CPU will no longer allow a `RestoreContext` to this Secure Executable, and the key to access the memory is discarded). This constitutes only a Denial of Service attack, which we do not claim to prevent. If the OS fails to issue this instruction when the process exits, the CPU resources allocated to this Secure Executable remain in use; however other parties can only use this information to resume execution of the Secure Executable where it left off. This results in repeating the instruction that caused the Secure Executable to exit last time it was running. See the description of the `exit` system call in Section 7 for an analysis of the case that the Secure Executable is resumed by the OS after exiting.

Secure Executable System Call (`sesc`). This instruction behaves like the existing PowerPC `sc` instruction: it throws an interrupt, giving control to the OS system call handler. However, we modify `sc` to give control instead to a Secure Executable-registered system call handler. Thus, we introduce `sesc` to obtain the original `sc` behavior when executing inside a Secure Executable. See the description in Section 7 for more details.

8.2 New registers

Secure Executable ID Save/Restore(`SEIDSR`). This register (privileged access) is used by the OS to save and restore the ID of the current Secure Executable. It is set when a Secure Executable is interrupted, and is accessed by the `RestoreContext` and `DeleteContext` instructions.

8.3 New state

Current Secure Executable ID. This state is not accessible by software, and contains the ID of the currently running Secure Executable. Software can only read or write it indirectly, through the `SEIDSR` register on the `RestoreContext` instruction.
Metadata corresponding to the current Secure Executable. This information is loaded from the Secure Executable metadata region on every return from interrupt. This region is protected by a single keyed hash value, which is stored on chip at all times in the Secure Executable table.

This information includes the following:

- Core dump data key
- Signal handler
- Memory address of the thread restore list
- The memory region mapping table

The memory region mapping table corresponds to the current Secure Executable. This allows quick mapping (on cache misses) from memory location to the memory region ID. Each row in the memory region mapping table has the following fields. This table is simply a read-only cache information stored elsewhere in memory.

- logical memory location and size: to quickly test which region an access corresponds to.
- Share Secret: for quick verification of the MRID share secret on regions shared by more than two Secure Executables
- memory region ID (MRID): index into the Protected Memory Table.

Private System Key. This information also cannot be accessed by software; it is used by the CPU only to decrypt the Executable Key in the `ESM` operand.
Cache Line MRIDs. Cache lines are labeled with the ID of the associated Memory Region.
Secure Executable Table.

- `SEID` Secure Executable ID (implicit: determined by the table position)
- Metadata hash
- Metadata region location.

Protected Memory Table. For each protected memory region, the following is stored on chip:

- `MRID` Memory region ID (implicit: determined by the table position)

- `SEID1` Creator SEID: this is the primary Secure Executable corresponding to the memory region (the one that first created it)
- `SEID2` Secondary SEID: this Secure Executable also always has access; allows quick access check of regions shared two-ways.
- Read only flag (boolean)
- Encryption and integrity key
- Integrity root value
- Share Secret: knowledge of this enables read/write access to this region by other Secure Executables (e.g. other than the ones listed as SEID1 or SEID2)
- Memory Region Size

9 Other Considerations

9.1 Shared Libraries and Dynamic Linking

Our current implementation requires a statically linked binary, since it is unsafe to allow the host machine to load untrusted code into our memory space at runtime. Statically linking allows all code to be verified at compile time, on the build machine. There are a few downsides to statically linked executables. First, they require extra storage space, since common libraries used by multiple applications cannot be consolidated. Second, the libraries cannot be updated once the Secure Executable has been deployed—in particular, bugs in these libraries cannot be fixed without rebuilding the Secure Executable. This property is necessary for Secure Executables because they do not allow untrusted code to access their memory space. Finally, statically linking with GNU libc in particular is only allowed under the Lesser GPL (LGPL) if certain conditions are met. Specifically, the LGPL requires that customers be able to relink vendor applications to replace the libc version [6]. This poses a challenge since in the Secure Executable model, the customer does not necessarily know the Executable key. One way around this issue is to use a private non-GPL version of libc for static linking.

Another potential solution is to use signed libraries, that are loaded and verified at runtime. A Secure Executable library loader, in protected memory, verifies the integrity of the library as it loads it in protected memory, and provides dynamic linking. That is, a module can be linked at build time with the Secure Executable that will load external libraries of unresolved symbols. These external libraries will only be accepted if there is a certificate chain, with the root signed by a party trusted by the developer, attesting to the trustworthiness of the library code. This mechanism provides integrity guarantees of the shared library, while allowing the library to be patched or replaced in the future.

However, it is not as simple as merely dynamically linking to the host libc: the matching certificate must be generated and transmitted to the host.

9.2 Virtual Machines

We consider support for two types of virtual machines (VMs) here. By "hardware" virtual machines we refer to VMs that do not require software assistance in executing non-privileged instructions (e.g., "Type 1", or "bare-metal" VMs). Non-privileged instructions in the guest are run directly on hardware. By "software" virtual machines, we refer to VMs that have a software hypervisor to assist in executing instructions (e.g., "Type 2", or most "hosted" VMs).

While our Secure Executable model can be applied to both virtualization scenarios, we consider hardware virtual machines to be the more relevant case for the high-performance server market. In a hardware virtual machine model, a Domain 0 OS typically controls switching between domains. The guest OS remains outside the TCB; Secure Executables operate as in the non-virtualized case. The Domain 0 OS is also outside the TCB, since hypervisor interrupts are treated as regular interrupts with respect to hiding the register contents. We simply require the Domain 0 OS to save and restore the `SEIDSR` register along with the rest of the registers during domain switching. The trick is that the Domain 0 interrupt handler needs to invoke the `RestoreContext` instruction to return to a Secure Executable, if one was running (the SE-ID register is set) during the hypervisor interrupt.

Software virtual machines require a different approach, since the software hypervisor needs access to the sensitive application in order to process its code and data. Thus, we wrap the VM inside the Secure Executable boundary. This is illustrated in Figure 8. Note that any VM that emulates instructions in software must be included in the trusted code base, since the instructions and data are considered sensitive. Any exception to allow an untrusted virtual machine to safely handle sensitive instructions and data would require expensive multi-party secure computation approaches, which we do not consider here.

In neither scenario is the host OS part of the TCB. The interesting question is whether, in software VMs, the guest OS must be part of the TCB. We show it does not have to be. Building the software VM as a Secure Executable offers an environment where the VM is protected from the host OS. With support from the software VM, constructing virtual Secure Executables protects guest applications from both the host OS *and* the guest OS. Compare this to Overshadow, which protects guest applications in software VMs from the guest OS but not the host OS. The TCB in our case is just the guest application plus the VM.

The Secure Executable uses an ESM operand encrypted for the virtual machine to decrypt. The virtual machine monitor, in turn, emulates the ESM instruction semantics, performing memory encryption and integrity checking. We assume for the sake of simplicity here that these cryptographic operations are performed in software; however, they can be hardware accelerated by a CPU with suitable cryptographic extensions. In principle, since the VM is already a Secure Executable, and thus protected by hardware, the VM does not need to

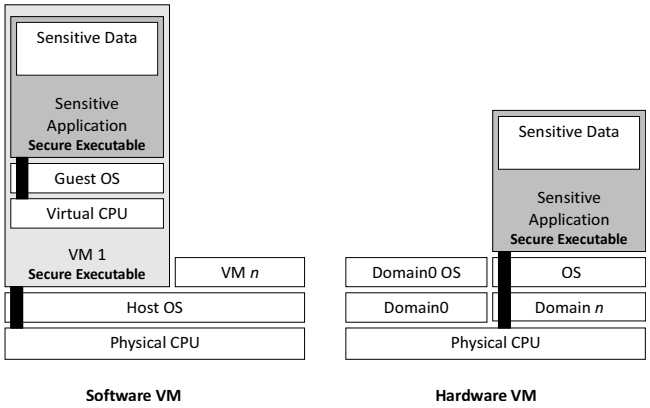

Figure 8: Virtual Machine Configurations

encrypt the application data at all, except when those pages are accessed by the guest OS. In turn, the VM runs in its own Secure Executable on the physical CPU. A certificate signing the public key of the VM, signed by the VM developers, asserts that the private key is available only to VMs running on Secure Executable-enabled CPUs. This builds the chain of trust, establishing for the application developers that the trusted base is limited to the guest application code, the VM, and the SE-enabled CPU.

To illustrate this point, consider first that the guest OS has been compromised. The encryption and integrity protecting mechanisms built into the VM ensure that the guest OS cannot access sensitive information in the guest Secure Executable. When the host OS has been compromised, the Secure-Executable mechanisms in hardware ensure that the VM cannot be tampered with. The chain of trust is established as follows. A public-private key pair is embedded with the VM, encrypted inside the Secure Executable binary so that only the target SE-CPU can access this VM keypair. The VM developer asserts the privacy of the VM key by signing the public key embedded in the VM Secure Executable. Finally, the developer of the guest Secure Executable encrypts the guest Secure Executable so that only parties knowing the VM private key (in particular, the VM targeted for the SE-CPU) can decrypt the Secure Executable. This isolates the root of trust, guaranteeing that the application developer only need to trust the VM software and SE-CPU (and of course, the SE-CPU factory and VM developer for key management).

9.3 Achieving OS Transparency

We believe that OS kernel support for these chip modifications is a reasonable requirement. It is still of interest to make Secure Executable support completely transparent to the OS, however, to allow legacy operating systems to run unmodified. In related work, Overshadow [4] and Flicker [13] provide this.

There are several aspects of our approach to consider with respect to OS

transparency. In particular, the interrupt handlers must be able to save and restore the application registers correctly. Moreover, when the interrupt handlers return control to a Secure Executable, the CPU must be able to identify the Secure Executable as such, without help from the OS. We ensure that no return to an application besides a Secure Executable can accidentally be labeled as a Secure Executable return (which would crash the application).

We use as an indicator a new instruction, `se-jump`. If on `rfid` the return is to an address containing this instruction, this indicates a return to a Secure Executable. Since these are invalid instructions in the current PowerPC architecture, no correct program would return to this point unless it is a Secure Executable.

In the case of a Secure Executable, when an interrupt occurs, recall that the CPU saves the registers to a restore point in application memory. At this time, the CPU also changes the program counter to point to a `se-jump` instruction in the restore point. When returning from the interrupt, this `se-jump` instruction restores the correct program counter from a register.

Note that we need the functionality of a jump (instead of, e.g., a `se-no-op` instruction) to handle system call returns. This is because PowerPC semantics specify that a system call success is specified by the return point. A system call returns to either the instruction following the entry point, or the following instruction. Providing the `se-jump` instruction allows the application return-from-interrupt code to determine the success value, and return to the appropriate code location.

This approach has an added complication: in PowerPC, Linux relies on the value of the program counter to resolve instruction page faults. By hiding this value from the OS, replacing it with an instruction in the restore point, an unmodified OS is unable to determine which page needs to be brought into memory. We can get around this with legacy operating systems by throwing a data page fault instead of an instruction page fault. In this case, the OS determines the location of the page fault from the data fault register instead of from the program counter.

We also require the OS to indicate to us when a Secure Executable has exited, so the processor can free the attached resources. A potential approach is to maintain a privileged process (distinct from the OS, but with hooks telling it when processes exit) that maps processes to Secure Executable IDs. This privileged process is then in charge of informing the CPU when a Secure Executable exits.

Finally, the overcommit workaround described in Section 4.1 requires OS support. This means overcommit must be enabled to achieve OS transparency.

10 Implementations

10.1 Build process

System call wrappers are linked in (or, the `sc` handler if using the `sesc` approach described in Section 7.2). The Secure Executable also needs unencrypted loader

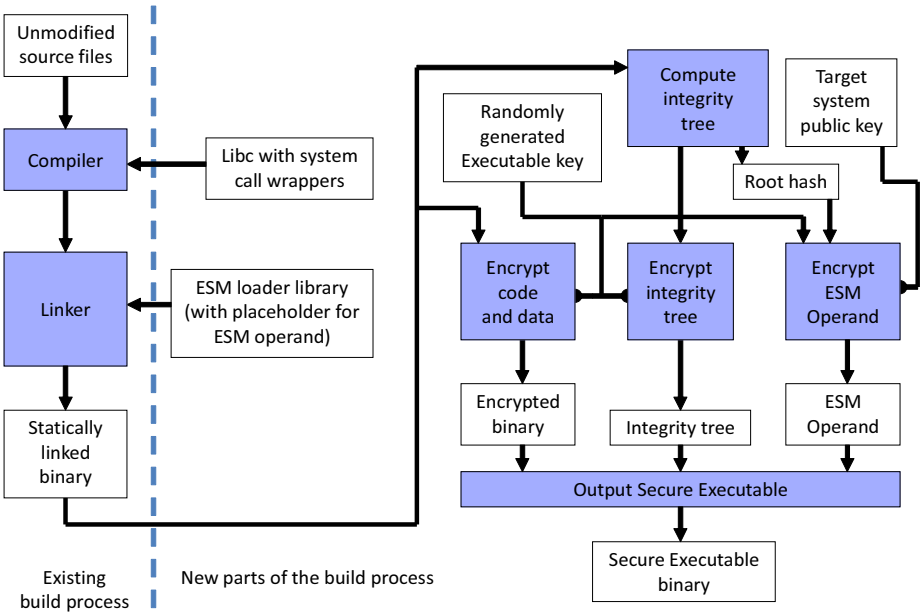

Figure 9: Build Process. Operations are represented with a blue background; data is represented with a white background. For encrypt operations, the encryption key applied points the side.

The Executable Key protecting this Secure Executable is randomly generated. This will be used to encrypt the static code and data sections, as well as in the computation of the integrity tree.

We now discuss how the ESM integrity value is computed by the build machine and how the initial integrity tree is constructed. In the secure build environment, the Secure Executable binary is statically linked with loader code and libraries. The initial integrity tree is included in the binary; the loader must insert it into the unprotected memory region. As discussed in Section 4.1 above, the bulk of the address space assigned to the integrity tree may be unused; therefore we use a compact representation of the tree on disk. This representation is a list of blocks of the tree as contiguous region (position, size, value) tuples. The loader simply copies these regions into the appropriate memory location (adjusting for the tree offset). Overcommit techniques (described

above) ensure that only the pages that are used are allocated physical memory pages.

Finally, all sections of the ELF that overlap the protected region are encrypted with the newly generated Executable Key. The ESM operand detailed in Figure 7 is constructed by concatenating the various parameters, encrypting with the executable key, then encrypting the executable key with the target system key and attaching.

10.2 Performance analysis

It is beyond the scope of this project to obtain performance results from a full CPU simulation. Future work will include both extending a POWER architecture system simulation to provide cycle-accurate timing results, and expanding the system wrapper library to the point where we can support existing off-the-shelf software. Ideally, measurements will look at the performance in real software.

Instead, we built a simple cache simulation to validate the memory protection model. This simulation shows that it is practical to implement the integrity trees as described in this paper. Even though verifying a memory location potentially requires verifying several locations, up to the memory root, we see that with normal cache hit rates, the overhead due to the integrity trees is very reasonable.

Figure 10 (left) shows the new overhead per load, as a function of the measured cache hit rate. A sequence of 10^6 addresses sampled from an exponential distribution was chosen to simulate a normal application. The hit rate and number of RAM accesses was measured. The cache is first put through a warm-up phase of 10^6 accesses before we begin measuring, to avoid being affected by the cold-start cache misses. This is not ideal, since it fails to capture the cost overhead at the start of program execution, but is necessary for the results to be independent of the simulation length. This limitation can be properly addressed in the future by measuring real software usage patterns across a full simulator.

A fixed cache size (4096 rows with 8 associative columns of 32 words each) is used for all simulations. For a normal application, the RAM accesses measured matches the number of load instructions when the cache hit rate is near zero. As the cache hit rate increases, the number of RAM accesses decreases linearly (following from the definition). The simulation was repeated, with the same address distribution parameter, for a Secure Executable. We then vary the exponential parameter to obtain different cache hit rates. The Secure Executable cache hit rate is lower than the normal application hit rate since cache lines are occupied with integrity tree values in the Secure Executable case. To capture the performance penalty of this aspect, the values are plotted against the normal application hit rate corresponding to this distribution.

Figure 10 (right) repeats this simulation, measuring stores instead of loads. While the cache logic is significantly different in the Secure Executable case, we see the measured performance exhibits similar behavior to the loads test. Note that in both figures, the integrity tree lookups dominate at low cache hit rates. Even so, the bulk of the integrity tree levels remain in cache, such that only

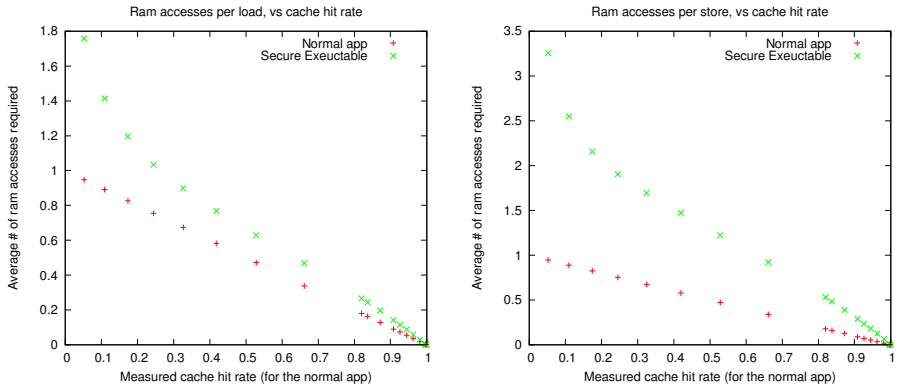

Figure 10: Left: Ram accesses per load. Right: RAM accesses per store

a few locations at the lower level need to be brought in. For cache hits, the integrity tree does not even need to be consulted. This is reflected in the better performance at high cache hit rates.

These figures do not address cryptographic performance. Since the number of cryptographic operations is equivalent to the number of RAM accesses, these graphs can be trivially adjusted to reflect this performance penalty by scaling the Y axis to the ratio of the latency of a hardware cryptographic verify and decrypt operation plus RAM access vs. a RAM access alone.

11 Conclusion

11.1 Limitations of this approach

First, we use a separate binary for every target machine. This does not require re-compilation, merely re-encryption. If we consider a Distribution Server to be the party that encrypts the code for each target CPU, the developer must trust the Distribution Server with the code and static data. This is necessary since this server decides who can read the code (and attached data). We thus assume this Distribution Server is part of the trusted build environment; however, this may not always be a convenient assumption.

A separate useful approach to this issue is to install a single Secure Executable on the target CPU, which we will call a Deployment Server. This Deployment Server will then decrypt future binaries that are distributed under a single key, and re-encrypt them for the target CPU. This allows a kind of boot-strapping of the application distribution: once a single trusted entity exists on the target system, it can be use to safely re-target other executables for that CPU.

Next, our implementation is on a RISC-like architecture; this may be stretching the definition of RISC since we require multiple memory accesses in a single

instruction. However, existing `POWER` instructions already have this requirement. Moreover, we still fit in the definition of a "load/store" architecture (under which each instruction can do only one or the other), since the `ESM` instruction is implemented as a "load".

Finally, it can be argued that we are minimizing the software trusted code base and optimizing application transparency at the expense of increasing hardware complexity, and thus the size of the hardware trust base. Acknowledging that we are increasing hardware complexity, we make the case in this paper that the sum of changes required is still reasonable. We require only a specific set of changes, which each have only a limited scope.

11.2 Final notes

This paper outlines the comprehensive set of hardware changes that together enforce the software isolation guarantees the Operating System is expected to provide, as well as protecting from physical attacks, malicious Operating Systems, and malware. This in turn minimizes the size of the software trusted code base. Notably, we achieve this in a manner mostly transparent to the application developers, and compatible with existing software. By providing a convenient way to achieve fundamental guarantees about the security of a software execution environment, the adoption of such an architecture will address some of the most pressing security issues today.

References

[1] IBM 4764 PCI-X Cryptographic Coprocessor (PCIXCC). Online at `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`, 2006. 7

[2] Daniel Bates. Eight million people at risk of ID fraud after credit card details are stolen by hotel chain hackers. *(UK) Daily Mail*, August 25, 2008. 1

[3] Rhys Blakely, Jonathan Richards, James Rossiter, and Richard Beeston. Britain's MI5: Chinese cyberattacks target top companies. *The Times (of London)*, December 3, 2007. 1

[4] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008. 5, 19, 26

[5] Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM*

conference on Computer and communications security, CCS '07, pages 389–400, New York, NY, USA, 2007. ACM. 4

[6] Free Software Foundation. GNU lesser general public license version 3. `http://www.gnu.org/licenses/lgpl.html`, June 2007. 24

[7] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on Oblivious RAM. *Journal of the ACM*, 43, Issue 3:431–473, May 1996. 7

[8] Jerry Harkavy. Illicit software blamed for massive data breach: Unauthorized computer programs, secretly installed on servers in Hannaford Brothers supermarkets compromised up to 4.2 million debit and credit cards. *AP*, March 28, 2008. 1

[9] IBM. IBM secure blue processor architecture. Online at `http://www-03.ibm.com/press/us/en/pressrelease/19527.wss`, year. 2

[10] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society. 4

[11] David J. Lie. *Architectural support for copy and tamper-resistant software*. PhD thesis, Stanford, CA, USA, 2004. Adviser-Horowitz, Mark. 3, 14

[12] John Markoff. Russian gang hijacking PCs in vast scheme. *NY Times*, August 6, 2008. 1

[13] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems(EUROSYS)*, 2008. 5, 26

[14] Jonathan Stempel. Bank of NY Mellon data breach now affects 12.5 million. *Reuters*, August 28, 2008. 1

[15] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. pages 160–171. ACM Press, 2003. 4, 14