Lockbox: Helping Computers Keep Your Secrets

Technical Report UCSC-WASP-15-02
November 2015

D J Capelis
mail@capelis.dj

Working-group on Applied Security and Privacy
Storage Systems Research Center
University of California, Santa Cruz
Santa Cruz, CA 95064
http://wasp.soe.ucsc.edu/

Abstract

The realization that modern architectures lack sufficient security features is not a new one. Many grand visions of "trusted computing" remain unfilled. After millions of dollars of research money, huge expenditures on the part of industry and rarely seen levels of cooperation between hardware vendors, users remain without substantial security features in their systems. Those features that do exist remain unused and few users urge progress. Even the strongest advocates of trusted computing have quietly ratcheted down their expectations and the current proposed uses for the TPM [32], a chip that was supposed to bring about revolutionary security benefits to modern computing, represent a significant departure from the original vision of the project. Despite a rapidly increasing need for security, trusted computing systems remain unimplemented, unrealized or without adoption.

Over the last few years, I designed an alternative system called *LockBox*. While it is not possible to fully characterize an unimplemented system, the design represents what I felt at the time to be the next generation of trusted computing technologies. And indeed, as the last generation of trusted computing hardware has faded away, Intel has put Software Guard Extensions (SGX) [1] on their product roadmap. Intel's SGX technology presents a surprisingly similar design to *LockBox* and accomplishes many of the same goals. The design work we've accomplished, along with the preliminary data we've gathered about how a system like *LockBox* might work, is perhaps even more critical now that hardware with a very similar design is slated to hit the mass market within a few years.

First I explain the design tradeoffs we made in designing *LockBox* and highlight where our system made different choices than Intel appears to be making with SGX. Second, I describe the technical design of *LockBox* in detail. Finally, I share the design ideas and data we gathered over our time exploring these types of systems and highlight a few areas that may be increasingly relevant as we see these systems develop a broader ecosystem and head towards widespread adoption.

1 Goals

A key issue which prevented adoption of the last wave of trusted computing platforms focused on user control [2]. Almost all previous trusted computing platforms were proposed as a way to secure content from users. While this may be a valid goal in some cases, the purpose of these systems is to remove control from the end-user of the device, which did not make them particularly popular. One of the key design principles in *LockBox* is that the end-user is given tools to control the security of their computer. In a reality where user-error is a large contributor to producing many security issues that people face, trusting a user to make security decisions is an unfortunately controversial approach. However, the only way to provide security features that enable users rather than restrict them involves putting security decisions in their hands. The merits of a security system that is controlled by someone other than the end-user is irrelevant if the end-user rejects that system. *LockBox*'s approach provides a feasible path to enabling security features for end-users.

LockBox allows a user to tell the system which applications they trust. *LockBox* provides a mechanism for the user to provide sensitive data to the system and ensure that only the application they trust with that data has access to it. *LockBox* ensures that this data remains secure if the trusted application is correctly programmed and it ensures this security even if the management software between *LockBox* and the trusted application is malicious.

LockBox provides these assurances by embedding features for *Trusted Loading*, *Trusted Memory*, *Trusted Runtime* and *Trusted Channels* into the architecture. The *Trusted Loading* features ensure that the user can identify and correctly load trusted applications. The features for *Trusted Memory* ensure that only the trusted application can access the secrets assigned to it, and the *Trusted Runtime* features provide the trusted application with protection from malicious privileged code. Finally, the *Trusted Channels* features provide the user with a way to exchange data with trusted applications.

LockBox enables users to protect themselves from bugs that result in completely malicious actions on the part their operating system or even their hypervisors. This is a critical distinction as many previous trusted platform proposals require an entire trusted software stack where all management software is required to be correct and bug-free. Yet, operating system bugs are all too common [12, 20] and as hypervisors get more complex, we are likely to see a similar progression. *LockBox* provides a way for data to remain secure even when these systems fail.

The design of *LockBox* requires defining a mechanism for the security system to prevent untrusted management software from compromising sensitive data while still allowing the management software to do its job. In general, this is accomplished based on a *request/verify* procedure where the trusted application makes requests to the operating system, but *LockBox* ensures a trusted entity can verify that the request was properly performed and the management software is properly functioning. Since *LockBox* preserves the management software's abilities to manage the system, malicious management software is allowed to manage processes and even kill a trusted application. However, even if the application is killed *LockBox* ensures that sensitive data cannot be retrieved. Reclaiming protected memory can only occur after that memory has been zerofilled.

These features move beyond the old definition of a trusted computing platform and incorporate real features that provide security benefits in computing systems. Systems that implement these features are necessary to enable an increased range of activities on computing systems. Current environments don't achieve the levels of security required for the full range of tasks users would like to do on their computers. Systems like *LockBox* may be necessary to achieve those goals.

2 Related Work

A large body of work has developed around trusted computing. However, the most visible and well known type of trusted computing platform remains something very different than what we discuss here. The Trusted Computing Group [33, 32] is a widespread industry effort supported by virtually every major technology company and produces a shipping product. For many years, their research dominated the discussion around trusted computing. There are also academic systems, like Terra [11], with similar goals, though Terra is implemented entirely in a hypervisor. Like the *LockBox* design, these systems provide a mechanism for memory protection called sealed storage and, in the case of the TCG's system, involve hardware modifications. Yet, unlike *LockBox*, both Terra and the TCG's system are designed to produce a full trusted software stack in which software at all levels of the machine is trusted. These types of trusted platforms are very different from *LockBox* because of their reliance on this trusted software stack.

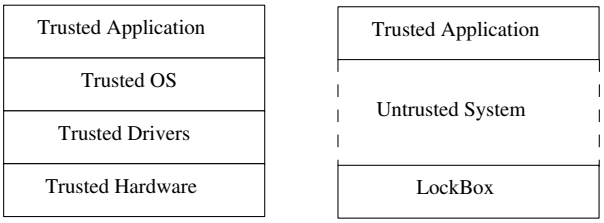

Figure 1: Full Trusted Stack vs. LockBox Design

Another set of systems, XOM [16, 17], AEGIS [30] and others [3, 9] are all systems which also embed security features into the hardware. XOM and AEGIS are of particular interest as they do not require a trusted software stack. Also in this category are the hypervisor-based systems Overshadow [7] and HARES [31]. While all these systems wrestled with many of the same issues present in the *LockBox* design, all of them chose to implement memory protection using an encryption layer. *LockBox* takes a different approach and eschews the use of encryption algorithms. Under *LockBox*, if one piece of code isn't authorized to see the contents of a page, then that piece of code isn't allowed to see the data in any form. The hardware returns zeros on reads and discards writes. This alteration fundamentally changes the relationship between the operating system and the rest of the computer organization.

This change requires new solutions in some areas that previous work does not address. Swapping, for instance, becomes an issue when the operating system isn't allowed any way to view protected memory. The reason *LockBox* is different from all other previous work in this respect is that unfortunately, allowing an operating system even an encrypted view of memory provides more information than it might seem. While the operating system may not be able to decode the exact contents, the OS will be able to track which data changes at which times and how much, which is an overlooked information leak in these systems. Not only is AES's 128-bit blocksize an issue, but memory's random access nature precludes the use of chaining modes without substantial performance degradation. The information gained by watching which portions of memory changed can be quite detailed. In a related technical report "When Encryption is not Enough: Memory Encryption is Broken", an attack against these types of systems is outlined and we describe the suite of tools we developed to analyze these weaknesses. Further, since *LockBox* doesn't require the use of encryption to implement its core security features it not only has the potential to perform better, but the system does not require specific cryptographic algorithms to be set in stone.

In most systems, the end-user has little control over which applications are allowed to use these

system's security features. Therefore, the user is often placed in a situation where an undesired application uses the security infrastructure not just to prevent them from, say, copying protected content, but also to create powerful rootkits [19]. Further, this lack of control by the end-user has been a large inhibitor to the adoption of trusted platforms [2]. *LockBox* addresses this issue by placing the user directly in control of the authorization and attestation processes. One of the research challenges in this proposal is defining how security architectures can directly interact with users in a sensible way. In a related technical report "A Short Study on Security Indicators" we present data from a user study that provides insight on how some of these interactions can be made more successful. *LockBox* stands alone in involving the user directly in the attestation and authorization workflow of a trusted platform. This is the only viable way for trusted computing to achieve acceptance in the marketplace and even by the full security community, parts of which have grave concerns regarding current systems.

Finally all of these systems specify either a trusted hypervisor or trusted hardware[1]. *LockBox* on the other hand, was designed to be able to run as either a trusted hardware design or as a pure hypervisor framework. This allows legacy machines to use *LockBox* with a microhypervisor while allowing for a smooth transition to a hardware implementation at a later time.

Some other pieces of the field are relevant to the development of these types of systems. Single Address Space Operating Systems [34, 6, 15] provide the basis for *LockBox's* SLB structure. Hypervisor monitoring systems like [27, 10] contribute some implementation level tips and tricks to make closely meshed page tables in hypervisor systems perform well. Work on hypervisor nesting with bluepill attacks [21] on Xen [4] as well as KVM's [14] production level implementation of hypervisor nesting have paved the way for hypervisors to emulate hardware features without eliminating the user's ability to run a more feature-rich hypervisor that accomplishes other objectives. In addition, various microhypervisor systems like Bitvisor [28] and SecVisor [24] have provided interesting insights into how security features should be incorporated into small hypervisors. Systems like vTPM [5] demonstrate the feasibility of simulating security hardware inside virtual machine monitors.

Multics's memory protection rings [22, 23] were an early form of memory protection and an important comparison point for future designs. Other approaches to system security include operating system hooks to produce fine grained security policy [29, 18] and better tools [8, 25, 13] to help software eliminate security defects and move closer towards correctness. These mechanisms will continue to be an important line of defense in the security of the overall system. Systems like *LockBox* complement much of the existing work in the field of security and provide applications with a last additional line of defense when these other approaches fail.

3 Technical Design

The *LockBox* design outlines a collection of architectural features to improve system security. Targeted towards implementation in either a nesting hypervisor or an FPGA, *Lockbox* is designed to examine how end-users could be provided additional hardware security features almost invisibly. The benefit of a nesting hypervisor is that a user can still run their own hypervisor on top of the system. This means users can still obtain the benefits of a feature-rich hypervisor without expanding the Trusted Code Base of *LockBox*.

In general, *LockBox* was designed to ensure that once a trusted application is provided with sensitive data, that data remains secure so long as the trusted application was correctly programmed and *LockBox*'s features are correctly implemented. *LockBox*'s features are designed to ensure that

[1]XOM specifies that part of the architecture features could be implemented in a hypervisor, but acknowledges a few important elements would still require specialized hardware.

users will be made aware of whether or not their secrets are being entrusted to applications running in a secure and trusted context. Even in the case that the trusted application contains bugs, an attacker no longer receives complete access upon compromising intermediate software, but is forced to break both the management software *and* all of the user's trusted applications protecting the data they desire. This substantially raises the bar required to steal sensitive data on a computer.

The design of *LockBox* revolves around four different sets of features:

Trusted Loading These features define the concept of an application and an application instance, and enables LockBox to identify an application instance when it makes a request to LockBox. The architecture grants trusted contexts and trusted identifiers to only the applications a user identifies as one they trust.

Trusted Memory These features protect memory from unauthorized users, including the operating system and other system level software. These features ensure that code running in a trusted context can make use of protected memory which cannot be read or written by other code running on the machine.

Trusted Runtime These features isolate code running in a trusted context from interference from supervisors and/or hypervisors running at a greater privilege level than the application, but at a lower privilege level than *LockBox*.

Trusted Channels These features allow code running in a trusted context to receive input from devices in a manner which identifies the trusted recipient to the user and ensures that this input cannot be read by any supervisors and/or hypervisors running at a greater privilege level.

3.1 Trusted Loading

The *Trusted Loading* features of the *LockBox* design provide the notion of identity and allow the architecture to maintain the concept of an application. This allows low-level components to identify applications and provides the basis on which access determination can be made.

Determining which applications a user trusts is another fundamental issue these features address. A trusted platform must be careful about the code it allows to run within a trusted context. If a trusted platform allows all code to run in trusted contexts, malware and other types of undesirable software can use the security features of the architecture to hide data from the end-user. On the other hand, if a trusted platform doesn't provide a mechanism for the user to allow code to gain access to the security features, then these security features are hardly useful. *LockBox* contains mechanisms to create secure trusted contexts only for those applications that the end-user authorizes.

3.1.1 User Access Device

Allowing a user to specify exactly which applications they trust requires the trusted platform design to directly interact with the user to determine their list of trusted applications. Since users tend to have trouble making trust determinations based on a set of hexadecimal numbers, a user interface is critical towards allowing the end-user to comfortably interact with the security system. A human-readable identifier (typically a unicode string) must be created that the end-user will recognize when prompted to input data destined for a trusted application.

Fortunately, an existing metaphor can be reused. People are comfortable with carrying keys to open door locks, car locks and other types of physical security mechanisms. *LockBox* is designed

to interface with end-users using a device that they would use just like a key; simply inserting the device into a special designated socket. The key provides digital storage populated with information read directly by *LockBox*. Typically, writing to the device should require the user to physically release a hardware interlock on the key itself once it's inserted into a machine. Users program their keys using a computer or device they trust.

The user's key contains the following information for each application they trust:

A human-readable identifier This is the human readable name the architecture will use to identify the trusted code to the end-user. It should be chosen by the end-user.

At least one of the following machine-readable identifiers:

A cryptographic hash On architectures which support a hashing mechanism in hardware, users can simply store a cryptographic hash that the loaded executable must match.

A full executable image An exact image of what the loaded executable must look like. The hardware compares this against memory contents to check for a match.

While the storage itself must be capable of being read directly by the architecture, the data on the key isn't secret. It simply contains a list of human-readable identifiers that correspond to a list of machine-readable identifiers.

The simple case for the User Access Device is that it is a simple storage device with a hardware interlock which is manually programmed by a user on a trusted machine. However, more complex possibilities exist. If the *LockBox* design were widely deployed it wouldn't be unreasonable to expect that stores might sell User Access Devices pre-populated with hashes for all the most popular trusted programs. In a corporate environment, an employee's name badge could serve wirelessly as their User Access Device. The device could be updated each morning as it interacts with the building's security system when the employee enters; signed updates would ensure security and the employer could transparently provide their employees with consistently up to date User Access Devices data. *LockBox*'s design can support much more complex environments than covered here.

3.1.2 Creating Trusted Contexts

While *LockBox*'s design does not trust privileged code to maintain the security properties of an application, it does trust privileged code to manage a machine's resources. Creating a new trusted context is something that can only be done by privileged code. However, in order to assign the human-readable identifier and create the trusted context the architecture *checks* that the operating system correctly loads the program into protected memory and that the program matches the user's machine-readable identifier. If *LockBox* successfully verifies that the privileged code has correctly loaded a trusted application, a new trusted context is created. Otherwise, the check fails and the trusted context is not granted. This trust but verify model allows operating systems and other magnagement software to serve in its traditional resource allocation and management roles without allowing it to violate security constraints.

3.2 Trusted Data

From the moment a new trusted context is created, new protected memory is allocated for the code that runs the context. The ability to utilize protected memory is crucial for trusted applications. While this memory is originally managed by privileged code, once it is actually allocated to a trusted context, it cannot be read or written by any other code on the machine, including the

operating system. This is a change to the way memory protection works in current architectures. This set of features provides the *Trusted Data* part of the design.

3.2.1 Security Lookaside Buffer

Code inside a trusted context does not have the same memory model as code outside a trusted context. Within a trusted context, an architecture-defined portion of the address space is reserved for protected memory access. Access to addresses within this range does not use the normal page table or Translation Lookaside Buffer (TLB) mechanism for virtual memory but instead uses a component called the *Security Lookaside Buffer* (SLB), which is a simple variant of a standard TLB. The SLB is backed by a separate set of memory security tables which determine the security properties of memory within this address range as well as the mapping of virtual addresses to physical pages.

This separate set of page tables is necessary to prevent a wide range of attacks that occur when privileged code is allowed to modify standard page tables for protected memory. Instead, when allocating new protected pages, privileged code must specify the identifier of the trusted context which will own the page and an explicit mapping between the protected virtual address space and a physical frame. Once *LockBox* accepts this allocation it writes the new mapping into a private storage location. Once this mapping is written, the memory arbiter begins to enforce the stipulation that the newly allocated physical frame may not be written by anything except a processor core running in a trusted context. Memory remains protected until either an entire trusted context and all its associated data is unprotected, in which case a trusted I/O controller overwrites the entire space, or the trusted context voluntarily releases the page using the page release mechanism described below.

The SLB can be implemented as a separate component or as extensions to the TLB. In either case, it must be managed by hardware and contain the following information:

- A mapping of the virtual page number to a physical frame number
- The identifier for the trusted context which owns the page
- A valid bit

In order to keep the complexity requirements to a minimum, the SLB simply uses the same set of tables for all code running in a trusted context. Since the SLB does not allow write and read operations from one trusted context to complete or return valid data when issued on memory which belongs to another trusted context, there are no security issues with having a shared mapping. Thanks to the large 64-bit address space found in modern processors there is plenty of room to allow for partitioning of the address space.

This maps particularly well to hardware, but can also be implemented easily in a hypervisor. In a hypervisor prototype these features can be implemented using page tables. The trusted I/O controller can be virtual, just a piece of software. This would essentially allow the operating system to use the virtual I/O controller as a hypercall interface to interact with *LockBox*. The SLB, while an important part of the hardware design, can be entirely implemented using the TLB in the hypervisor based design.

3.3 Trusted Runtime

The set of features in the *Trusted Runtime* section of the *LockBox* design ensures that code in a trusted context cannot simply be subverted by the management software on the machine. This set of features allows the upper portion of a software stack to run on top of untrusted management

software. To keep this trusted code running inside a trusted context secure, the *LockBox* design includes several changes. First, secure data inside registers cannot be exposed on a context switch. Second, as shown by geometry attacks [26], arbitrary transfer of control into a protected code page cannot be allowed as it is equivalent to allowing arbitrary code execution from that same page. Finally, to enable swapping types of operations there needs to be a sensible mechanism to allow privileged code to deallocate protected pages and flush a representation of them to disk.

3.3.1 Program Status Page

In the *LockBox* design, *Trusted Runtime* features are enabled with the help of a preallocated page within every executable which gets loaded into a trusted context. This page is called the *Program Status Page* (PSP) and its address serves as an identifier for the trusted context. The page contains the following information in a layout defined by the individual implementation:

Register Flush Set Pointer This is a pointer to an area that will store the working set of registers when a context switch occurs. It is loaded by the architecture on entry to the context.

Restore Lock This field contains space for a lock. This lock is set during the restore handler to avert timing issues and properly deal with concurrency.

Requested Program Counter Contains a copy of what the Program Counter (PC) was before being overridden when the PC (re)entered the trusted context.

Page Release Address Contains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid. This area is initialized to an invalid address.

Swap Handler Address Contains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Interrupt Handler Constains space for a pointer. When set to an address that is not inside the protected address space, this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Address Contains space for a pointer. When set to an address that is not inside the protected address space this pointer is invalid and no handler exists. This area is initialized to an invalid address.

Trusted Data Length Contains an integer set by a trusted I/O controller when a trusted channel has been closed.

Trusted Device Type Contains an integer set by a trusted I/O controller when a trusted channel has been opened. The value of zero is invalid.

Default Register Flush Set Space for flushing one set of registers. When the PSP is initially created, the *Register Flush Set Pointer* field points to this area.

These fields are used by the trusted application and *LockBox* to enable the following features:

3.3.2 Protecting Registers

In normal operation, an operating system can pre-empt a process at any point and view its architectural state. Unfortunately this is no longer acceptable when a security system is in place which allows a program to place data in its architectural state which privileged code is not permitted to see. Therefore, any time the processor forces a transfer of control to a location outside the trusted context, the registers are flushed to area specified by the trusted context's *register flush set pointer* and cleared.

3.3.3 Preventing Arbitrary Jumps

Not only is there a need to prevent arbitrary jumps back into code running in a trusted context for security reasons, but there is also a need to restore the sets of registers which get purged every time the control flow unexpectedly leaves the trusted context. For both issues, the solution is the same. Upon re-entering code that is part of a trusted context, the processor should override the PC and enter the code at the beginning. The old PC value gets placed in the program status page in the *Requested Program Counter* field. In a hypervisor version, this can be implemented using the NX bit on the page tables.

This means at the beginning of every program designed to run in trusted mode there should be either an instruction which transfers control to a restore handler or the restore handler itself. This handler can take care of such tasks as restoring a program's register set from the program status page, resuming execution at the point the program was interrupted or checking the *Requested Program Counter* field to determine if control flow should go to a different location. If the requested program counter is set to a value permitted by the application, the application transfers control to that location.[2]

3.3.4 Concurrency

The restore handler not only checks for a valid entry point and restores the registers, but plays a critical role in ensuring that *LockBox* works correctly with multiple threads in either a unicore or multicore system. The *Restore Lock* field in the PSP is the main tool the handler and architecture use to coordinate. When the architecture transfers control to a protected memory region, it acquires this lock. If the lock cannot be acquired, then the architecture may either retry, or return control to the operating system. This allows the restore handler to ensure that each execution context receives a unique location to flush its registers.

To take care of control transfers during the time the restore handler is running, the architecture does not enable flushing registers to protected memory until after the *restore lock* is released. Instead of flushing the register set, the architecture releases the *Restore Lock* and transfers control back to the OS. Normally, the *Restore Lock* is released at the end of the restore handler with an atomic write instruction. This write signals to the architecture that the context is again prepared to receive execution contexts and the current execution context is ready to writeback its registers to the area specified by the *Register Flush Set Pointer*.

While the *Restore Lock* serializes execution in the restore handler, it does not prevent applications from taking advantage of multiple execution contexts. (i.e. multiple threads) The operating system is free to pass a thread ID to the restore handler in a register, (or any other mechanism defined by the OS ABI) which the restore handler can use to resume the appropriate thread. The

[2]Implementations with concerns about the performance of this section of the architecture could implement portions of it in hardware or provide instructions which will accelerate these operations.

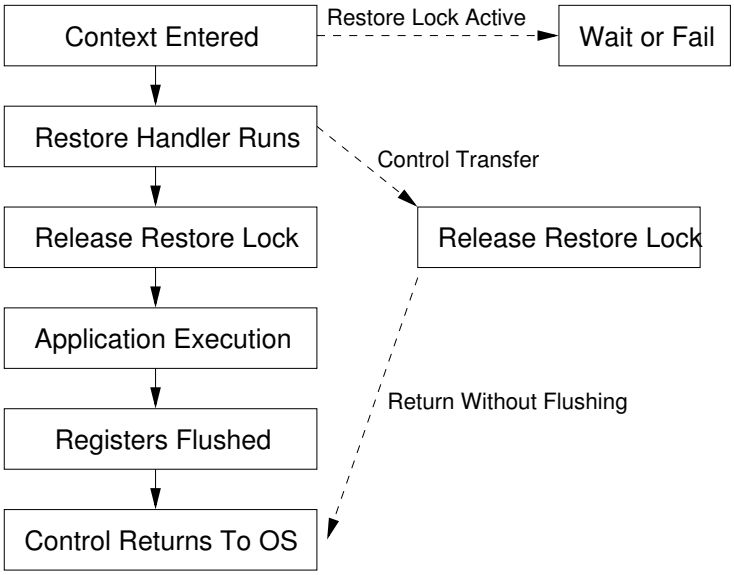

Figure 2: Context Switch Handling

architecture reads the value of the *Register Flush Set Pointer* field *on entering the context* to determine where the thread's register set will be flushed. The restore handler sets the pointer for where the *next thread* will flush its registers and thus can ensure that each thread within the trusted application flushes their registers to a unique spot. The restore handler should also record the location of the *Register Flush Set Pointer* for the current thread in a table so that later invocations of the restore handler knows where to find the thread's register flush set to properly restore execution after a context switch occurs. Lastly, the restore handler zeros the PC from the register flush set of the current thread so future invocations of the restore handler can detect whether or not the current thread has finished. (When the current thread finishes, the non-zero value in the PC will get flushed to the flush set.)

In the case of a single threaded application, the locking mechanisms are still required to prevent overwriting valid register flush sets from context switches inside the restore handler, but the remaining issues are simplified. The *Register Flush Set Pointer* can remain at its default value, pointing to the default *Default Register Flush Set* and the restore handler simply returns when the operating system tries to enter the protected context while an existing thread is running somewhere else in the context.

3.3.5 Page Release Mechanism

LockBox's design contains a page release mechanism which allows a trusted context to release a page of protected memory from its control. This is an important portion of the architecture; without it, an operating system would be unable to swap protected memory. So this mechanism allows a trusted context to release a page from security protections. If the context does so, it is responsible for either purging it of secrets or replacing the data with an encrypted version which can be handled by untrusted code. Existing work with encrypted swap shows low overhead and hardware acceleration instructions are now present in almost all new chips produced by major vendors. In this environment, the performance of "software encryption" for swap is likely negligible.

To perform a swap operation, the operating system sends a signal to the process which lets the process know that the operating system had determined that the page at a specified address should

be swapped out. The program can either refuse to unprotect the page and risk that the operating system will then decide to kill the entire process by requesting the architecture destroy the entire trusted context, or it can comply. If the program chooses to comply, it will likely want to replace the contents of the page with an encrypted version. The program will then place the address in the *Page Release Address* field of the PSP and transfer control back to the operating system, which can then release protections on the page, and flush it to disk.

If the trusted context ever tries to read or write to a protected address for which no mapping exists, the processor jumps to the *Swap Handler Address* listed in the PSP for the trusted context. The handler should request the operating system swap the page back in and generate an access violation if the OS does not have the page. A well-written handler will also want to verify that the contents are correct and decrypt them if needed. Various cryptographic mechanisms can be used to verify integrity, including cryptographic signatures and cryptographic hashes. If the PSP does not contain a valid handler address, the processor traps to the operating system just as if it was unable to load a TLB entry for that address.

3.3.6 Co-operative Swapping

Trusted computing architectures such as [7, 17] allow operating systems to swap protected pages without much issue through the use of cryptography. These architectures provide two views of memory, one encrypted and one decrypted. Applications within a certain trust domain are given a decrypted view of memory while everything outside it, including the operating system, is given an encrypted view. This works well for swapping as the Operating System can simply flush the encrypted copies of the page to disk and restore the encrypted version as needed. In *LockBox*'s design however, the lack of a PKI or any sort of hardware encryption prevents us from following this model. Instead, *LockBox* relies on a *co-operative swapping* mechanism.

To set up co-operative swapping, an application performs the following steps:

1. The application produces a key based on trusted information requested from the user through a trusted channel.

2. The application uses *mlock()* or a similar call to request that the operating system wire down the pages that contain: 1) The code used to handle swap requests 2) The code used to encrypt and decrypt pages 3) The page where the key is stored.

3. The application installs a handler for swap-out requests with the operating system.[3]

4. The application places the address of the handler for swap-in requests in the *Swap Handler Address* field of the PSP.

Once these operations are performed, the application continues running as normal; if at any time the OS wants to swap out a page of protected memory, the program uses the *Page Release Mechanism* described in previous section.

3.4 Trusted Channels

Trusted Channels are needed for a trusted application to communicate with the user and obtain secret information. The initial design work around *LockBox* largely focused on trusted input. This allows for users to provide secret data to trusted applications without fear of that data being

[3]Recall that the *Requested Program Counter* field allows for applications to permit code from outside the trusted context to jump to certain well-defined addresses within the protected address space.

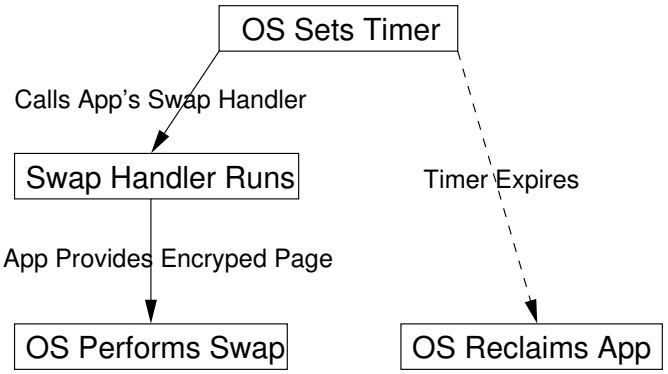

Figure 3: Co-operative Swapping

intercepted. The minimum trusted output necessary to implement *LockBox* is the application name. More complex outputs are only needed if applications need to display confidential data. For some common usecases, (i.e. password entry) securing only input is reasonable and allows us to implement real security features without the complexity of outbound trusted channels.

To create a trusted channel, the application requests one from the operating system. Assuming the operating system decides to comply with this request, the OS sets a trusted I/O controller to handle interrupts from the device with which the channel was requested. The trusted controller writes the type of the device into the *Trusted Device Type* field of the trusted context's program status page. From this point, a trusted channel is now open.

The trusted DMA engine then provides the device with the human-readable name associated with the trusted context. The device can then display this information to the user. This information allows the user to know two things: 1) Their information will be going to a trusted application and will be stored in secure memory 2) Their information will be going to the trusted application they see displayed by the device. This allows them to determine whether or not to use the device to reveal secrets to their trusted applications. If there is no name displayed, the user will know that the hardware is not securing any of the secrets they provide.

When data comes in from the device on the other side of the trusted channel, the trusted controller places it at a protected page pointed to by the *Trusted Data Address* field in the trusted context's PSP. It then clears the trusted device type field, writes the length of the data written to the page to the *Trusted Data Length* field and closes the channel. Outbound channels will be implemented using a very similar mechanism, but in reverse.

4 Implementation Concerns

While *LockBox* was never completed as we designed it. If someone were interested in continuing this work, here are some concerns they might find relevant.

4.1 Hardware Cost

Several hardware alterations are required to the processor to implement *LockBox*. Due to the wide variation in other components of the system, such as the motherboard, memory interface and peripheral devices, I limit my analysis of *LockBox*'s hardware cost to the on-chip modifications.

The following on-chip modifications are present in LockBox:[4]

[4]In this case I consider the memory controller, despite being on-chip in recent Intel processor designs, to be an

Caches The caches must be modified to store an additional flag of metadata. This flag indicates whether the line's security properties must be looked up in the Security Lookaside Buffer.

Security Lookaside Buffer What the TLB does for address translation, this new structure does for security constraints. This structure queries the memory arbiter to determine the security properties of the page.

Register File Due to the extremely sensitive timing of this component, it is critical the changes to the register file be as noninvasive as possible. The only alteration to the register file is that during a context switch, registers may need to be flushed by hardware. Instead of modifying the register file directly, a hardware assisted register flush can be implemented by running code from a small ROM. In this case, the register file itself will be able to remain unmodified with only the addition of a small ROM and minor control logic. Engineering challenges for each specific VLSI design will dictate the level of modifications possible for the register file.

4.2 Deployment

Deployment is always a focal point of any change to core technologies. *LockBox*'s design has the following deployability advantages:

- *LockBox* gives control of security to the consumer buying the machine.
- *LockBox* can be deployed in software if compatible hardware is not available, or without additional layers of software if compatible hardware is used.
- Security threats against hypervisors and operating systems continue to increase. *LockBox*'s design provides users with a way to keep some security even when these layers cannot be trusted.

4.2.1 Developing the Software Ecosystem

After exploring the tradeoffs with *LockBox* in a hypervisor, and proving the feasibility of the hardware design, there remains a need to show the various types of security features that a system like this would enable. In particular, one of the most exciting areas of work is designing the modifications which will be needed to implement support for *LockBox* in most modern programming languages. There is lots of potential for integrating this work into mainstream compilers and making the security benefits of the system usable by adding a compiler flag during the compilation process. The compiler could emit a program that would invoke the *LockBox* framework and store secret data within the hardware. Since there is no requirement for cryptographic signing of programs, no certificates will be needed and nothing will need to be configured. Designing this type of high level interface would be key for achieving adoption of this technology.

There are several other interesting applications that could be developed with a prototype of this technology. Unfortunately the effort needed to develop these applications is beyond the time and resources of the initial rounds of my dissertation work. Future research projects could experiment with the following applications of the *LockBox* design:

Secure Input Libraries A modified input stack for a typical X11 session which can automatically request secure keyboard input for any trusted application which comes into focus. This could automatically provide secure keyboard input to any trusted application which asks for it.

off-chip component.

Secure Linking and Loading The linking and loading environment may need to be modified to support some aspects of *LockBox*. New methods for using shared code will need to be found and some changes in this area will be examined.

Secure Web Browsers This application is the one which most inspired this design. The ability to type passwords into a web browser on public terminals without concern that the machine has been remotely programmed to store all keystrokes is something which should be available to all computer users.

Secure Virtual Machine Monitors With the increasing interest in virtualization and containerization technology, it would be interesting to take a virtualization or container framework and modify it to allow all of its guest systems to use this framework. This would allow the construction of a container system which, if compromised by an attacker, would not reveal data from the guest systems and be able to provide increased isolation for virtual machines.

5 Project Status

This technical report outlines the design behind the *LockBox* system. The system has not yet been published in a peer-reviewed venue primarily due to the difficulty of implementing and verifying changes in computer architectures. While we still believe our lab's roadmap using the OpenSPARC FPGA core is a feasible path to hardware implementation. The high barriers with fundamental architecture change span from the hardware up through every layer of the software stack. This ultimately proved to be beyond the resources available to us during the project timeline.

The challenging aspects of proposing fundamental security change at an architectural level are particularly enhanced by the very understandable desire for such proposals to come with demonstrable data on performance overhead and hardware cost. Science is well served by these demands for empirical data. Yet in this project, getting realistic performance data on real applications entails not only real implemented hardware, but a rearchitecture of operating system and application code. This is a tall order and does raise questions about the feasibility of the efforts we propose, we will note that several other efforts in this area have implemented change at a similarly disruptive scale without building an entirely working system first. Most notably, the Trusted Computing Group's efforts to create and embed the Trusted Computing Module into x86 computing systems.

In many ways, without a full working system, we end where we started: with a design for a more secure computing architecture we believe could be successful and provide security benefits for users.

We hope this technical report inspires others to build upon our ideas.

6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1018928. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Bibliography

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, p. 10.

[2] R. Anderson, "Cryptography and competition policy: Issues with trusted computing," *Workshop on Economics and Information Security*, pp. 1–11, 2003.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1997, p. 65.

[4] 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 *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 164–177.

[5] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vtpm: virtualizing the trusted platform module," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4, pp. 271–307, 1994.

[7] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2008.

[8] C. Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," *Linux Expo*, 1999.

[9] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 221–232.

[10] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 51–62.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 193–206, 2003.

[12] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[13] IBM, "Software: Purify," *IBM Rational*. [Online]. Available: http://www.ibm.com/software/awdtools/purify/

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Linux Symposium*, 2007.

[15] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *ASPLOS-V: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 175–186.

[16] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *The Ninth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 35, no. 11, pp. 168–177, 2000.

[17] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[18] B. McCarty, *SELinux*. O'Reilly, 2004.

[19] D. K. Mulligan and A. Perzanowski, "The magnificence of the disaster: Reconstructing the Sony BMG rootkit incident," *Berkeley Technology Law Journal*, vol. 22, p. 1157, 2007.

[20] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security and Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[21] J. Rutkowska and A. Tereshkin, "Bluepilling the Xen Hypervisor," *Black Hat USA*, 2008.

[22] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, 1974.

[23] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3, pp. 157–170, 1972.

[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[25] J. Seward *et al.*, "Valgrind, an open-source memory debugger for x86-GNU/Linux." [Online]. Available: http://www.ukuug.org/events/linux2002/papers/html/valgrind/

[26] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," *Proceedings of the 14th ACM conference on Computer and Communications Security*, pp. 552–561, 2007.

[27] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," in *16th Annual Conference on Computer and Communications Security*, 2009.

[28] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai *et al.*, "BitVisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 121–130.

[29] B. Spengler, "Detection, prevention, and containment: A study of grsecurity," *Libres Software Meeting*, 2002.

[30] G. 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*. ACM New York, NY, USA, 2003, pp. 160–171.

[31] J. Torrey, "HARES: Hardened Anti-Reverse Engineering System," in *SyScan*, 2015.

[32] Trusted Computing Group, "Trusted platform module specification v1.2 rev103," *Trusted Computing Group*, 2006.

[33] ——, "Trusted software stack specification v1.2 rev103," *Trusted Computing Group*, 2006.

[34] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," *SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 304–316, 2002.