Specifying and Verifying Hardware for Tamper-Resistant Software

David Lie John Mitchell Chandramohan A. Thekkath[*] Mark Horowitz

Computer Systems Laboratory
Stanford University
Stanford CA 94305

Abstract

We specify a hardware architecture that supports tamper-resistant software by identifying an "idealized" model, which gives the abstracted actions available to a single user program. This idealized model is compared to a concrete "actual" model that includes actions of an adversarial operating system. The architecture is verified by using a finite-state enumeration tool (a model checker) to compare executions of the idealized and actual models. In this approach, software tampering occurs if the system can enter a state where one model is inconsistent with the other. In performing the verification, we detected an replay attack scenario and were able to verify the security of our solution to the problem. Our methods were also able to verify that all actions in the architecture are required, as well as come up with a set of constraints on the operating system to guarantee liveness for users.

1. Introduction

There are many systems that try to ensure some measure of software tamper-resistance and copy-resistance [8, 11, 18, 26, 27]. In this paper, we focus on a particular hardware approach called XOM, which stands for eXecute Only Memory [14]. XOM embeds cryptographic keys and functionality on the main processor chip to provide an extremely high level of security for applications. XOM makes no assumptions about the trustworthiness of the operating system or physical memory. Since the XOM hardware is responsible for access control, the operating system only controls resource allocation. However, to manage resources effectively, the operating system retains a higher privilege level than a user process, and an adversarial operating system may abuse this to try to access user data. On the other hand, an adversary with physical access to the system may monitor the pins of the processor and modify or observe values on the bus to memory [10]. Cryptographic keys under XOM control must protect program code and data in spite of these active attacks. The XOM hardware requirements and likely execution speed have already been studied [14], and XOM offers the promise of tamper-resistant code, at moderate hardware cost and minimal drop in execution speed. However, there is no value in paying for the additional hardware unless the architecture provides verifiable security.

The primary goal of our verification study is to determine whether XOM systems are able to provide reliable security to software distributors. We evaluate XOM security using a model with three parts: the XOM machine with its security mechanisms, user code, and an adversary that tries to access user data or modify the user state without being detected by the security mechanisms. A secondary goal is to see whether the system has been over-designed. After the system has passed its primary test, we go back and remove parts of the machine model to see if the system still passes the safety tests. If the system is still safe despite the removal of some components, then we can examine whether those components are necessary. A final goal is to verify that the security measures do not prevent the user from making forward progress.

Our verification uses the Murφ model checker to simultaneously search two different models. One model, the *actual* model, contains an adversary with a set of abilities. The model checker tries all possible combinations of these abilities, trying to violate the safety guarantees of the model. To check those guarantees, we compare the state of this model with an simpler, *idealized* model that does not contain an adversary. If the two models ever become inconsistent, then the adversary has managed to tamper with the state of the user in the model.

We will begin in Section 2 by giving some background on the XOM architecture and on Murφ. In Section 3, we formally specify the two models of the hardware we use, as well as our model of the attacker. The next section explains how the two models are combined into one model

[*] Microsoft Research, Mountain View, CA 94043

that Murφ uses to perform a joint state exploration. Section 5 gives the results of our verification. In this section we describe how our tool helped find a solution for memory replay attacks. We also show one instance where our method was able to find an action in the model which did not provide any extra security, as well as give details on the constraints that must be placed on an operating system to guarantee forward progress or liveness for the user. In Section 6 we compare our method against existing and related work. Finally we conclude this paper with Section 7.

2. Background

The XOM architecture [14] is a generic microprocessor architecture that maintains separate *compartments* for programs. On-chip compartments are isolated using architectural tags, with encryption used to maintain the isolation when data is written off-chip. XOM is fundamentally different from architectures that use secure coprocessors [24, 27] in that it provides a higher level of security by implementing the secret keys and tags directly on the main processor. Each user program in a XOM machine has a unique key, called a *compartment key* that the XOM machine associates with a *XOM ID tag*. The program is initially encrypted with the compartment key. When the code is about to be executed, it is read from memory and decrypted and tagged with the program's XOM ID. Any on-chip data or code that belongs to a program is also tagged with that program's XOM ID. The tags act as an identifier of the writer of the data, and thus determine who can read the data.

The XOM machine tags data with the XOM ID as a proxy for encrypting it, deferring the encryption to when the data leaves the chip boundary to be stored in memory. When data is stored to memory, it is encrypted with the compartment key, and a hash of the data and its address is added to protect against the tampering of memory values. This prevents an adversary from substituting random values in place of encrypted values, or from copying encrypted values from one address to another. Only a program that knows the user's compartment key may modify or view the contents of the compartment. The architecture records the writer of data and ensures it matches the reader. Thus, if an adversary tries to tamper with data by overwriting it with some faulty value, when the user tries to read that data, the architecture will detect a user/writer mismatch and halt the user. A program may allow another program to read its data by explicitly moving that data from its private compartment to a *shared compartment*. In this way, programs in separate compartments may selectively share data.

The XOM architecture can be built on top of any existing architecture through modest modifications to the processor hardware, and has been shown to be realistically implementable [7, 14]. Because cryptographic operations are performed on every memory operation, the XOM architecture relies on caching to mitigate the performance impact. Values in the cache are not encrypted, but tagged with XOM ID's.

XOM defends against attacks where the adversary has subverted the operating system to her needs. Even though an adversarial operating system can execute both privileged and unprivileged instructions, it cannot forge the user's XOM ID. Thus, the XOM machine should prevent the adversary from tampering with the user's data by checking the data's XOM ID tag against the XOM ID tag of the active program. It is important to note that the security enforced by XOM is completely orthogonal to the security enforced by a processor's privileged and unprivileged modes. Even privileged instructions must meet the access requirements of the compartment security model.

Some additional instructions must be introduced to the instruction set for users to take advantage of the XOM hardware. `Secure load` and `secure store` instructions are provided to allow programs to preserve the XOM ID of data when storing it from registers into the cache. Later, when the data is flushed to memory, it is encrypted with the compartment key indicated by the XOM ID tag. The operating system needs to be able to save the state of the interrupted user program to perform a context switch. To support this functionality without allowing the operating system to read user data, XOM provides the `register save` instruction. The XOM machine encrypts the register contents and stores the cipher text in the target register. The operating system can then store the state of an interrupted process, but does so safely since it is only allowed to handle the encrypted state. When the operating system wishes to restore the register contents it uses the `register restore` instruction. The XOM machine returns those values back to the registers. A hash is added to ensure that the encrypted values have not been tampered with and to make sure the encrypted values are restored back to their original register. An additional measure is put in place to guard against a replay attack from the operating system. The operating system could potentially save encrypted registers and restore them multiple times, thus replaying a section of user state. To prevent this, the *register key* used to encrypt the user registers is invalidated each time the user is interrupted with a trap, and a new key is generated. By changing the key, the XOM machine ensures that the register state of a user can be restored at most once. The instructions pertinent to this verification are summarized in Tables 1 and 2.

In performing our verification, we used the Murφ [4] model checker. Murφ uses explicit enumeration to check the state space of a model. A model describes the system to be checked as a state machine by providing an initial state and a set of next-state functions. The next-state functions are specified by a set of "rules", which have a precondition

Instruction	Description
i1. `def $rt, immediate`	Generic register definition where `immediate` is written into register `$rt`. This models any non-memory operation that writes to a register.
i2. `use $rt`	Generic use of register `$rt`. This models any non-memory instruction that reads a register.
i3. `secure store $rt,addr`	This stores the value of register `$rt` to the memory location at `addr`. The store sets the XOM ID tag in the cache to the value of the user who caused the store.
i4. `secure load $rt,addr`	This loads the register `$rt` with the memory value at `addr`. If the load hits in the cache, the XOM ID tag of the cache data is checked against the user's XOM ID. If the data is in memory, the data is decrypted with the user's key and the hash verified before loading into the register.

Table 1. User Instructions

Instruction	Description
i5. `register save $rt,$rs`	The operating system uses this instruction to store a user register. The XOM machine encrypts and hashes the user register `$rs` with a temporary *register key* and stores it to register `$rt`. The register `$rt` is tagged with the operating system's XOM ID. The hardware atomically performs task, the operating system can only access the encrypted value and the hash. The register key is inaccessible to the operating system and is stored as part of the architectural state of the machine.
i6. `register restore $rt`	The operating system uses this instruction to restore registers saved with `register save`. The machine decrypts the data, verifies the hash, and returns the data to its original register. The *register key* is revoked each time the XOM machine traps to the operating system.
i7. `prefetch addr`	The operating system can move a value from memory into cache, even if its XOM ID does not match the key the data in memory is encrypted with. However in the cache, the data is tagged, not with the XOM ID of the operating system, but with the XOM ID of the key used to decrypt it from memory.
i8. `write_cache addr`	The operating system can change the value of a cache location. The new data is tagged with the operating system's XOM ID.
i9. `invalidate addr`	The operating system can invalidate a cache location causing the data to be destroyed. The data is not flushed to memory.
i10. `flush addr`	The operating system can flush a cache location with `addr` from the cache to memory. The value is encrypted with the key indicated by the XOM ID tag, hashed and the placed in memory. The XOM ID of the cache location does not have to match the XOM ID of the operating system.
i11. `trap`	The operating system can cause the processor to trap to the operating system at any time by sending an interrupt. However, each time this happens, the *register key* used to encrypt and decrypt user registers from any previous traps is invalidated and can no longer used.
i12. `return`	The operating system can return use of the processor to the user at any time.

Table 2. Privileged Instructions

guard, and a set of actions that modify the current state to produce a new state. A precondition is a boolean statement based on the current state of the machine. Murφ performs the state exploration by starting with the initial state and exhaustively searching for all successor states. Murφ finds and executes rules whose precondition is satisfied by the current state to identify successor states. Murφ verifies the correctness of each new state against a set of safety criteria to determine if any of the states are illegal. Safety criteria in Murφ can be specified as a set of invariants, which are boolean statements that are evaluated every time a new state is found. When Murφ detects an error, it outputs a counter example that indicates the states it traversed to reach the error state. Murφ has been successfully used in other work to verify both security protocols [17, 22] and computer hardware [13, 25], but this is the first instance we are aware of where it has been used to verify tamper-resistance.

Model checkers, in general, have some limitations. First, they verify models of systems, not the systems themselves. Models abstract details of the system to make the size of the state space tractable for the model checker. This is often done by simplifying functionality and by scaling down the models. Second, model checkers can only explore a finite number of states, and may miss states to save memory. For example, rather than explicitly remembering the states it finds, Murφ saves a smaller, randomized low-collision hash of them. There is some probability that a collision will result in missed states, but because the hash is randomized, successive verifications of the same model reduce this probability.

3. XOM Models

In constructing the XOM models we make several assumptions about the types of errors we are trying to catch. First, we employ a "black box" model for cryptographic functions [5]. This means that encrypted data cannot by decrypted by the adversary. However, if two plain text values are equal, they will have the same cipher text values. We also assume that hashes are collision free, and that programs have been properly written, so that all data sharing occurring in the shared compartment is intentional.

Finally, we do not model what we call "information" attacks. These are a class of attacks where by observing many runs in a XOM machine, the adversary can surmise some information about a program. For example, the adversary is able to obtain a trace of memory locations the user accesses. In essence, the adversary learns something about the user — some bits of information have been leaked. While these attacks exist, they are outside of the scope of our verification framework.

3.1. Abstracting the instruction set

To reduce the state space of our models, we consider a simplified instruction set, in comparison to the instruction set available on a real XOM machine. We seek to reduce the instruction set to just the operations that affect the flow of data and information. For example, generic register operations are amalgamated under the `def` and `use` operations, while control flow operations such as branches and jumps are left out. Similar simplifications were performed to analyze Java in [6]. The instructions available to the user are summarized in Table 1.

An adversarial operating system can execute both user instructions and privileged kernel instructions. The additional privileged instructions available to the adversary are summarized in Table 2.

3.2. The actual model

The model of the physical hardware in a XOM machine, called the *actual* model contains three homogeneous arrays. Each array represents one of the storage levels in the machine: registers, cache, and memory. The records stored in the arrays have different properties. For example, they all have a data value property, as they can all be used to store data. However, not all have XOM ID tag, hash, or key properties.

A state in the model S_{actual} contains the three arrays, as well as a single bit that indicates whether the *user* or the *adversary* is executing on the machine.

$$\begin{aligned}\boldsymbol{S_{actual}} = <& r_0, r_1 \ldots r_x;\\ & c_0, c_1 \ldots c_y;\\ & m_0, m_1 \ldots m_z;\\ & mode >\end{aligned}$$

The state has 3 storage classes, each representing a storage level. A register r_i, has a data value d, a XOM ID tag t, and if the value is encrypted, a key k, and a register hash h, associated with it. Similarly, a cache line c_i, has a data value d, XOM ID tag t, and a memory address a. Finally, memory locations m_i, have data values d, keys k, and address hashes h. The $\varnothing$ value means that the parameter is not defined. For example, register values can be in plain text, meaning they have no key or hash value associated with them. Similarly, an unused cache location has a $\varnothing$ address value. Finally, either the adversary or the user can be executing as indicated by the $mode$.

$$\begin{aligned} r_i &: \text{record } \{d : \boldsymbol{D}; \quad t : \boldsymbol{P}; \quad k : [\boldsymbol{P}, \varnothing]; \quad h : [\boldsymbol{R}, \varnothing]\}\\ c_i &: \text{record } \{d : \boldsymbol{D}; \quad a : [\boldsymbol{M}, \varnothing]; \quad t : \boldsymbol{P}\}\\ m_i &: \text{record } \{d : \boldsymbol{D}; \quad k : \boldsymbol{P}; \quad h : [\boldsymbol{M}, \varnothing]\}\\ mode &: [adversary\_mode, user\_mode]\end{aligned}$$

The model has four basic data types, to which members of the records belong. $\boldsymbol{D}$ is a set of distinct data values, $\boldsymbol{P}$ is a set of principals and can either be the user or the adversary, $\boldsymbol{R}$ is a set of register hashes, one for each register in the model, and $\boldsymbol{M}$ is a set of memory addresses in the model.

$$\begin{aligned}\boldsymbol{D} &= [0 \ldots w, \alpha]\\ \boldsymbol{P} &= [user, adversary]\\ \boldsymbol{R} &= [0 \ldots x]\\ \boldsymbol{M} &= [0 \ldots z]\end{aligned}$$

The α in $\boldsymbol{D}$ represents a value that was created by the adversary, which is used to model data injection attacks. The XOM machine uses tags as proxies for keys, so tags and keys are type-equivalent. Similarly, the pre-image for a memory hash is an address so they are also type-equivalent. Initially, the model has $\varnothing$ in all its storage locations and the *mode* is set to *user* mode. The maximum number of data values w, number of registers x, number of cache lines y, and number of memory locations z parameterize the model.

If the model detects tampering, it prevents the user from continuing execution by causing a *reset* action. The reset action is a special action that sets the entire machine state to a legal state. In our model, the reset condition sets the state back to the initial state of the model. The user is able to perform the operations specified in Table 1. The actions that produce the next state for each instruction are:

i1. Define data value $a \in [0 \ldots w]$ in register i:
$r_i = \{d = a, t = user, k = \varnothing, h = \varnothing\}$.

i2. Use register i:
if $r_i.t \neq user$
then *reset*.

i3. Store value at register i into address j:
if $r_i.t \neq user$
then *reset*
else if j is in cache such that $\exists\ c_l.a = j$
    then $c_l = \{d = r_i.d, a = j, t = r_i.t\}$
    else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
    $c_l = \{d = r_i.d, a = j, t = r_i.t\}$.

i4. Load memory address j into register i:
if j is in the cache such that $\exists\ c_l.a = j$
then if $c_l.t \neq user$
    then *reset*
    else load from cache $r_i = \{d = c_l.d, t = user, k = \varnothing, h = \varnothing\}$
else load from memory: if $m_j.k \neq user \vee\ m_j.h \neq j$
    then *reset*
    else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
    $c_l = \{d = m_j.d, a = j, t = user\}$,
    $r_i = \{d = m_j.d, t = user, k = \varnothing, h = \varnothing\}$. [1]

The model checks that the user's data has not been tampered with. As explained in Section 2, a hash validates two components — the integrity of the encrypted data and the validity of the address it is being loaded from. These two checks are modeled separately. Preventing the adversary from creating any data that is encrypted with the user's key simulates the integrity check of the data. On the other hand, the validity of the address is modeled by keeping a shadow copy of the original address of the data in h.

Here is a section of the model in the Murφ description language. This particular section describes instruction i3:

```
Rule "User secure store"
  !isundefined(reg_i.data) &
  mode = user_mode
==>
Var
  cache_l : cache_range;
Begin
  if (reg_i.tag != user) then
    reset();
  endif;
  -- find the data in the cache
  cache_l = find_data(addr_j);
  if (!isundefined(cache_l)) then
    -- hit in the cache,
    -- write data to cache
    cache_l.data := reg_i.data;
    cache_l.addr := addr_j;
    cache_l.tag := reg_i.tag;
  else
    -- cache miss (code not shown)
  endif;
endrule;
```

A “`--`” indicates the following text on that line is a comment. The precondition is the boolean expression before the “`==>`,” which checks that the register has data in it, and that the processor is not in adversary mode. The body of the function, after the `Begin` statement checks the permissions on the register, checks if the data is in the cache, and if so, writes it to the cache. The actions for servicing a cache miss are left out for brevity.

This model has some safety conditions that are checked:

1. Each cache location must have a different address tag. This ensures the cache is implemented correctly.

2. User data (data that is not α) is either tagged with the user's XOM ID or encrypted with the user's key. This ensures the access control guarantees are correct.

3.3. The idealized model

The *idealized* model is a very simple version of a XOM architecture. It has no caches since they are invisible to the user, and does not contain an adversary. The model has a two storage levels: registers and memory locations. The state of the model can be expressed as:

$$\begin{aligned} \boldsymbol{S_{ideal}} &= < r_0 \ldots r_x; \quad m_0 \ldots m_z > \\ r_i &: [0 \ldots w, \varnothing] \\ m_i &: [0 \ldots w, \varnothing] \end{aligned}$$

[1] We note that this definition prevents the user from reading uninitialized memory values. Such a read is considered a programmer error, which we do not cover in this paper.

The parameters of the model are w, x and z which are the number of data values, registers, and memory locations respectively. We see the model is much simpler and each storage class only has a data value property. As in the actual model, the initial state has all locations initialized to $\varnothing$. There is only one user in this model and the user can perform the following actions:

i1. Define data value $a \in [0 \dots w]$ in register i:
$r_i = a$.

i2. Use register i:
this action is a null action.

i3. Store value at register i into memory location j:
if $r_i \neq \varnothing$
then $m_j = r_i$.

i4. Load an initialized memory location j into register i:
if $m_j \neq \varnothing$
then $r_i = m_j$.

The same register store action in the actual model is written as the following in the idealized model:

```
Rule "User secure store"
  !isundefined(reg_i.data)
==>
Begin
  mem_j.data := reg_i.data;
endrule;
```

The model has no safety conditions since the absence of an adversary means it cannot be tampered with.

3.4. The adversary

The actual model also includes an adversary that can perform actions to modify the state of the machine. In the model checker, the adversary is given a set of primitive actions that it can perform. We then rely on the model checker to exhaustively try all combinations of these actions in an attempt to break the XOM machine. It is important to ensure that the adversary is adequately powerful to model all attacks, but constrained so as not to be capable of things not possible in reality. Based on our earlier "black box" model for cryptography, and assuming reasonable countermeasures against physical tampering of the hardware, our adversary is not able to:

1. Access registers not tagged with its ID.

2. Decrypt values for which it does not have the key.

3. Access keys stored on the XOM machine.

4. Forge hashes.

The actions that the adversary can perform are the basic user operations and the kernel mode operations detailed in Table 2. Aside from the 12 basic instructions available to the adversary, we add two composite instructions. These instructions are composed from several basic instructions, but do not require a free a register or cache line.

i1. The adversary can define data in a register i:
$r_i = \{d = \alpha, t = adversary, k = \varnothing, h = \varnothing\}$.

i2. The adversary can use a register i:
if $r_i.t \neq adversary$
then *reset*.

i3. The adversary can store a register i to address j:

```
if $r_i.k = \varnothing$
then if $r_i.t = adversary$
    then if $j$ is in the cache such that $\exists\ c_l.a = j$
        then $c_j = \{d = r_i.d, a = j, t = adversary\}$
        else pick an $l \in [0 \dots y] : c_l.a = \varnothing \rightarrow$
         $c_l = \{d = r_i.d, a = j, t = adversary\}$
    else reset.
```

i4. The adversary can load a cache location i to register j:
if $c_i.t = adversary$
then $r_j = \{d = c_i.d, t = adversary, k = \varnothing, h = \varnothing\}$
else *reset*.

i5. The adversary can save register i to register j:
if $r_i.k = \varnothing$
then $r_j = \{d = r_i.d, t = adversary, k = r_i.t, h = i\}$.

i6. The adversary can restore register i to register j:
if $r_i.h = j$
then $r_j = \{d = r_i.d, t = r_i.k, k = \varnothing, h = \varnothing\}$
else *reset*.

i7. The adversary can prefetch memory location i into cache location j:
if $m_i.h = i$
then $c_j = \{d = m_i.d, a = i, t = m_i.k\}$
else *reset*.

i8. The adversary can write cache location i:
$c_i = \{d = \alpha, a = c_i.a, t = adversary\}$.

i9. The adversary can invalidate a cache location i:
$c_i = \{d = \alpha, a = \varnothing, t = adversary\}$.

i10. The adversary can flush cache location i:
given $c_i.a = j \rightarrow m_j = \{d = c_i.d, a = c_i.k, h = j\}$.

i11. The adversary can trap to adversary mode. When doing so, the register key is revoked, so all encrypted registers are cleared:
if $mode = user$

then $mode = adversary$,
$\forall\ r_i \in [r_0 \ldots r_x]$: if $r_i.k \neq \varnothing$
then $r_i = \{d = \alpha, t = adversary, k = \varnothing, h = \varnothing\}$.

i12. The adversary can return to user mode and allow the user to execute:
if $mode = adversary$
then $mode = user$.

c13. The adversary can copy a memory location i to another memory location j:
$m_i = m_j$.

c14. The adversary can copy a register i to register j:
if $r_j.t = adversary$
then $r_i = r_j$
else *reset*.

In modeling the adversary, we make two simplifying assumptions. First, in primitive i5, the adversary is not allowed to encrypt an already encrypted register. The same is true for an adversary trying to decrypt a value that has not been encrypted. While a real adversary could encrypt or decrypt register contents an arbitrary number of times by executing the appropriate instruction over and over again, our models are finite so we cannot model this. Our models can easily be extended to allow an arbitrarily long chain of these instructions, but this would create a larger state space. Thus, for efficiency, we restrict these operations to be performed only once on any value.

Second, we do not allow the adversary to store encrypted register values to memory in primitive i3. The only operation that could be performed on an encrypted value is primitive i6, which only operates on values in registers. As a result, storing the values to memory means at some point, the adversary will have to load that encrypted value back from memory to another register to do anything with it. Since, this is already modeled in action c14, modeling this functionality again is unnecessary.

In the description of the XOM architecture, attacks were classified into three categories: spoofing, splicing, and replay [14]. We will show how the primitive actions above allow an adversary to at least implement all three of these attacks. While, the adversary does not know the user's key and thus cannot insert chosen text into the user's compartment, she can still attempt to randomly write values there. A spoofing attack can be performed by actions i1, i8 and i10. A splicing attack is one where the adversary tries to copy valid, user encrypted values from one location to another. Actions c13, c14, and i7 allow an adversary splice registers, memory locations and caches respectively[2]. Finally, a replay attack involves an adversary who actively records data, waits for the user to overwrite that location with different data, and then inserts the old, stale data. To replay a register, the adversary executes c14 and then i11 to copy and return control to the user. When the user overwrites the old register value, the adversary executes i11 again, and uses c14 to copy the saved data back. To replay a memory location, the adversary can either use c13 instead of c14, or use i9 to prevent a new value in the cache from reaching memory.

4. Verification

Our primary goals are to verify two properties of the XOM architecture: that adversaries cannot read user data and that adversaries cannot modify user data without being undetected.

Since the XOM machine only permits principals to read registers that have been tagged with the correct XOM ID, we verify the first property by checking that data created by the user is never tagged with the adversary's XOM ID. This is actually the second safety condition in the actual model given in Section 3.2. However verifying the actual model alone does not ensure that the adversary has not modified user data. The difficulty is that there is no condition to check the user's data against in the actual model. The key observation is that there can be no tampering by the adversary on the idealized model by virtue of the fact that there is no adversary. Thus, the idealized model can be used as a "golden" model against which we can check the state of the actual model. For this, assume we have the function f that checks whether a certain state in the actual model matches a certain idealized model state:

$$\begin{aligned} f : & f(Actual\ Model\ State, Idealized\ Model\ State) \\ & = \{true, false\} \end{aligned}$$

We can verify tamper resistance by exploring both the idealized and actual model states simultaneously. This involves concatenating the idealized model state with the actual model to create the "joint" state. We also label every action in the actual model as either a user action or an adversary action. User actions are ones that would be performed by a user, and as a result, these actions have analogues in the idealized model. All other actions are considered adversary actions and have no analogue in the idealized model. User actions affect the state of the idealized and actual portions of the model state, while adversary actions only affect the actual portion of the joint state. We apply f to each new joint state created to verify consistency. If this check above holds for all the states found, then the adversary is not able to make the actual state inconsistent from the idealized state, which leads to the conclusion that the adversary was not able to tamper with the user's data.

[2] Note that actions i5 and i6 do not constitute splicing attacks since the copied values are inaccessible to the user due to the XOM ID tags on the registers.

The merging of the models must be done in a way that does not restrict the state exploration of either model, otherwise this may result in the Murφ failing to find states where the two models might have been inconsistent. Murφ rules have a guard condition that states when a particular rule can be applied. The idealized model is what the user should think she is running on, so user actions in the joint model derive their guards from the idealized model. The one caveat is all user actions can only execute when the actual model is in *user_mode*, so this check is added to all user guards. The bodies of those rules are a combination of the actions that modify the idealized model and the actual model in parallel.

Adversary actions have no corresponding actions in the idealized model and so they are guarded by elements from the actual state. Naturally, the adversary actions only modify the actual state of the model. Because of this, we see that adversary actions change the actual state but leave the idealized state unchanged. As an example, we provide a section of the Murφ description that combines the same user action from the actual and idealized models given in Sections 3.2 and 3.3. Because the states of the models are now combined into the same name space, elements from the idealized model are prefixed with an "i", while elements in the actual model are prefixed with an "a":

```
Rule "User secure store"
  -- only guarded by idealized state
  !isundefined(ireg_i.data) &
  -- must be in user mode
  mode = user_mode
==>
Var
  acache_l : cache_range;
Begin
  -- actual model part
  if (areg_i.tag != user) then
    reset();
  endif;
  acache_l = find_data(addr_j);
  if (!isundefined(acache_l)) then
    acache_l.data := areg_i.data;
    acache_l.addr := addr_k;
    acache_l.tag := areg_i.tag;
  else
    -- cache miss (code not shown)
  endif;
  -- idealized part
  imem_j.data := ireg_i.data;
endrule;
```

We have outlined the method for combining the two models – all that is left is to defined f. Our XOM CPU is based on a RISC-like load/store architecture. As a result, assembler instructions either move data between registers and memory, or they perform logical or arithmetic operations on register values. As such, from the point of view of a running program, the only state that needs to be consistent is the register state since that is what is used to do any computation that could produce output. Because of this property, our function f turns out to be very simple:

$$
\begin{aligned}
&f : (S_{actual}, S_{idealized}) \rightarrow \{true, false\},\\
&\text{if } \forall\ S_{actual}.r_i.id = user \wedge\\
&\qquad S_{actual}.r_i.data = S_{idealized}.r_i.data\\
&\text{then } f = true\\
&\text{else } f = false
\end{aligned}
$$

The XOM architecture only allows programs to read registers that are tagged with their XOM ID. The above f is true if registers tagged with the user's ID, and thus are accessible to the user, have the same data in both the actual and idealized model. If a XOM machine was built on top of a CISC machine, where arithmetic or logical operations may be performed directly on memory values, then values in memory and cache must always be consistent as well. This does not preclude a definition for f, but would make the definition more complex.

One of the limitations is that the model must approximate the real machine by scaling down the number of elements in the model to limit the state space. The verification was performed on a scaled down model with 3 memory locations, 3 cache locations, 3 register elements, 2 user data values and 1 adversary data value. All three attacks detailed in Section 3.4 require a minimum of two user data values, in the case of a splicing or replay attack, or a user value and an adversary value in the case of a spoofing attack. Thus, the largest number of different data values we need to support in our models is three. As a result, we require three elements in each storage level so that the adversary can move all possible values around without overwriting some. While, this is not proof that all cases will be covered in our approximated model, we believe that our model does cover all the attacks capable by our adversary model. We were able to run slightly larger models (with one of the storage levels increased to 4) but did not find any more errors. With more memory, larger models could be verified.

With this method that combines our two models in a specific way, we were able to verify the XOM architecture and find errors that we will detail in the next section.

5. Results

In performing our verification, we found a way for an adversary to perform replay attacks on memory values. With our tool, we were able to verify that our solution to the error is correct. We then searched the XOM architecture for extraneous actions and were able to find one. Finally we checked for liveness guarantees and were able to show that when certain constraints were placed on the operating system, the user could be guaranteed forward progress. The models ran for approximately 4-6 hours on a Sun Workstation with 8GB of memory and 1Ghz UltraSparc 3 processors.

5.1. Replay attacks on memory

We were able to find an exact sequence of events that allow adversary to replay values in memory. This existence of this attack was also suggested in [7, 14, 21]. Our method also helped us find and implement a safe solution to the memory replay problem.

We start by noting that [14] proposes that a hash of a memory region can be used to protect that region from replay. We model this hash by creating a second memory array that shadows the memory in the actual model. Because the hash is meant to be kept in a replay-proof register, we make this shadow memory inaccessible to any of the adversary actions. A *calculate hash* function models the calculation of the hash by copying the contents of memory in the actual model into the shadow. A *verify hash* function then checks the contents of the shadow memory against the contents of memory in the actual model and an exception is thrown if the two do not match.

However, [14] does not precisely define when and where the *calculate hash* and *verify hash* functions should be invoked. Since the hash updates would be expensive in reality, we decided to try to reduce the frequency of the updates. We take advantage of the fact that cache locations are on chip and thus cannot be modified by the adversary without detection. We implemented a model where the hash is only updated when a cache line is flushed to memory and verified when a memory location was read into the cache. While this appears to be all right at first glance, Murφ was able to find an attack where the adversary would invalidate cache lines before they were flushed to memory so that the old value in memory became the current value. It is possible to exploit this vulnerability as shown in Table 3.

Even though the hash matches the memory value A, the last value written was B, so the adversary has successfully performed a replay attack. This problem arose because the write to an address, which occurs when the value is written into the cache, is not atomic with the update of the hash, which occurs when the value is flushed to memory. With Murφ we are able to show that against an adversary who cannot invalidate cache lines, delaying the update of the hash in this way is safe. However, since many architectures support his operation, this optimization is generally unsafe. As a result, we must be sure the hash is updated whenever a value is written to the cache.

Action	**$**	**M**	**H**
1. User writes A to cache.	A	$\varnothing$	$\{\varnothing\}$
2. Cache is flushed to memory.	$\varnothing$	A	$\{h(A)\}$
3. User writes B to cache.	B	A	$\{h(A)\}$
4. Adversary invalidates cache.	$\varnothing$	A	$\{h(A)\}$

Table 3. Updates of the hash cannot be delayed as shown here with $ is the contents of the cache, M is the contents of memory, and H is the value of the hash.

Since of reducing the frequency of hash operations failed, we try instead to reduce the cost of each hash calculation by using an incremental hash. An incremental hash uses a function to add and remove elements from a hash efficiently. An example of an incremental hash that uses the exclusive-or function is given in [3] and an implementation appears in [7]. Such a hash would make updates to the hash more efficient as we would not need to read all memory locations to recalculate the hash. Instead, we simply remove the old value from the hash, and add the new one. Again, we were able to find a weakness. Essentially, Murφ exploited the fact that when we read in the old value to remove it, we do not actually verify that the old value is the correct value. A clever adversary can insert a different value at this point to create havoc. For example, the adversary could insert a value that will cancel out the value that the user is about to write, thus leaving the hash unchanged. The adversary is then free to replay an old value, since the hash was not updated as shown in Table 4.

Again, though the last value written is actually A, the hash for B validates correctly. It seems that the only way to defeat this is to verify that the value being removed is the correct value, which requires reading in all of memory when updating the hash. Unfortunately, this negates the benefit of the incremental hash.

From these two failed hash implementations, we were able to create a successful hash implementation. The first optimization failed because hash calculations are not atomic with updates to memory. Thus we call the *calculate hash* function every time the user writes values to the cache. Similarly, we must call *verify hash* every time a value is read into the cache from memory and before we recalculate a new hash as shown by the second failed optimization. Unfortunately, this method may be inefficient because the entire memory region must be read each time the hash is calculated or verified. Some techniques, such as hash trees and caching may be applied to this problem to help mitigate its performance impact [7].

Action	**\$**	**M**	**H**
1. User writes A to cache.	A	$\varnothing$	$\{h(A)\}$
2. Cache is flushed to memory.	$\varnothing$	A	$\{h(A)\}$
3. User writes B to cache.	B	A	$\{h(A) - h(A) + h(B)\} = \{h(B)\}$
4. Cache is flushed to memory.	$\varnothing$	B	$\{h(B)\}$
5. Adversary replays A in memory.	$\varnothing$	A	$\{h(B)\}$
6. User writes A to cache.	A	A	$\{h(B)\}$
7. Cache is flushed to memory.	$\varnothing$	A	$\{h(B) - h(A) + h(A)\} = \{h(B)\}$
8. Adversary replays B in memory.	$\varnothing$	B	$\{h(B)\}$

Table 4. An incremental hash cannot be used as shown here with \$ is the contents of the cache, M is the contents of memory, and H is the value of the hash.

5.2. Extraneous actions

We removed actions from the model to see if they were necessary for security. With Murφ we found one action in the model that appeared to be extraneous. When the user loads data from memory, it is not necessary to check that the data is actually encrypted with the user's key. It is sufficient to simply tag the register that the data is stored to with the key that the data was encrypted with. In other words we can change the user action to:

```
4. Read memory address $j$ into register $i$:
   if $j$ is in the cache such that $\exists\ c_l.a = j$
   then load from cache $r_i = \{d = c_l.d, t = c_l.t, k = \varnothing, h = \varnothing\}$
   else load from memory if $m_j.h \neq j$
        then *reset*
        else pick an $l \in [0 \ldots y] : c_l.a = \varnothing \rightarrow$
         $c_l = \{d = m_j.d, a = j, \boldsymbol{t = m_j.k}\}$,
         $r_i = \{d = m_j.d, \boldsymbol{t = m_j.k}, k = \varnothing, h = \varnothing\}$.
```

Later, when the user tries to use the data with a `use` operation in Table 1, the machine will check the tag anyway. Though unnecessary for security, specifying the key in the instruction does make the hardware more efficient. The XOM architecture allows encrypted programs to select whether data stored to memory should be encrypted or in plain text. Having the program specify the whether the data it loads is encrypted or not, saves the hardware from having to maintain that information.

5.3. Liveness

We were also able to show that the user can be guaranteed forward progress given that a set of constraints are placed on the operating system. Because the operating system is untrusted in a XOM machine, the machine does not ensure that users are always able to make forward progress. However, a "properly working" operating system should be able to guarantee forward progress. A properly working operating system must demonstrate two things. First, it must show that the user is always able to make forward progress. Second, the operating system must show that it is able to perform all functions it needs to perform.

The first condition can be met by modifying the conditions under which the operating system actions will execute. Note that the function f, defined in Section 4, still evaluates to true even if data that exists in the idealized model is not present in the actual model. This means that the verification will not catch the case where the operating system either destroys user data or moves user data without moving it back. In such a case, when the user tries to access data, she will find that it has been overwritten with operating system data, and the machine will reset because of an XOM ID tag violation.

To find the constraints that must be placed on the operating system to guarantee user forward progress, we redefine the reset action to be a violation of a safety property so that Murφ will report it as an error. From the counter example, we can deduce additional guards on adversary actions that would prevent that particular sequence of actions from happening. In effect, these guards, turn the adversary from a malicious operating system to a properly working one. The general constraints are:

1. If the operating system moves the user's data away, the operating system must move it back before returning to the user.

2. The operating system must not destroy any user data by overwriting it.

These constraints ensure that the operating system does not destroy any of the user's data or render it inaccessible by moving it to a place that the user is not aware of.

By adding more detail to the model we were also able to find another constraint on the operating system. This is a race that can cause the user to inadvertently lose data. To cause this error to happen, we must model the caches in more detail. Modern processor cache lines usually contain multiple register words, while our simple model hash cache lines that are one register word long. Longer cache lines also have single XOM ID tag associated with them. They also have valid bits that indicate which words in the cache line belong to the XOM ID tag. When the XOM ID tag of a cache line changes, all previous data in that cache line is invalidated. The race occurs if the user writes data to a cache line and then the operating system writes data to the same cache line. Even if the operating system's data is written to a different address in the same cache line, the write by a different XOM ID causes the old data to be destroyed. As a result, the operating system is constrained from sharing any cache lines with user applications. This constraint falls under the second constraint listed above.

The second requirement, that the operating system is able to perform all functions despite the above constraints, is more difficult to prove. Essentially this shows that the operating system is still able to interrupt applications and manage system resources amongst applications. At the end of a state exploration, Murφ displays a list of all actions executed along with the number of times they were executed in the state exploration. With this information, we can show that the operating system is still able to execute all the actions it needs to, despite the additional constraints on when they can execute. We note, however, that this does not necessarily show that the operating system can execute actions *when* it needs to — it simply shows that there exist conditions within the model for the operating system to execute all it's required actions.

6. Related work

Theorem proving is a formal verification alternative to model checking. The design of IBM 4758 Secure Coprocessor [23] used theorem proving techniques to verify its security. While theorem proving is not restricted to a finite number of states, it significantly more difficult and time consuming to use.

The idea of performing verification by checking for consistency between a higher level model and a lower level model has been detailed in work on refinement maps [2]. Our technique differs from refinement maps in that we ensure that every transition in the idealized model has an existing transition in the actual model, while a refinement map specifies the converse. There has also been some work that verified security by asserting an equivalence between an idealized model and a model with certain actions available to the adversary. One specification method using equivalence between a realistic model and an idealized attack-impervious model is outlined in [15], with related ideas presented earlier in [1]. Prior work on CSP and security protocols, also uses process calculus and security specifications in the form of equivalence or related approximation orderings on processes [19, 20].

Lastly, none of the techniques presented in this paper are Murφ specific. A variety of other model checkers such as SPIN, TLA+, or SMV could have been used [9, 12, 16].

7. Conclusion

We have given a formal specification of the XOM architecture that supports tamper-resistant software. We then developed a method to verify the specifications. In this verification, we used the Murφ model checker to perform the simultaneous state exploration of an actual and an idealized model. We verified two safety properties. The first ensures that an adversary was never able to read data that belongs to the user. This is done by making sure that user data is always either tagged in the XOM hardware with the user's ID, or encrypted and hashed with the user's key. The second safety property ensures that the adversary can never tamper with the user's data without the user's execution being halted. This is done by splitting all transitions in the XOM models into user caused actions and adversary caused actions. We then simultaneously explore two models of the XOM architecture — one that only the user could affect and the other that both the adversary and the user could affect. If one model ever becomes inconsistent with the other, than the adversary will have successfully tampered with the user state.

After successfully detecting design errors by model checking, we corrected the errors and found no further problems. With this, we were able to show how a replay-proof register can be used to protect memory from replay. We were also able to find one action in the specification that was not necessary for security, but is required to make the implementation practical. Lastly, we were also able to put some constraints on what actions in the operating system can perform so as to guarantee the user forward progress. In this way, we were able to show that it is possible to build a well behaved operating system.

Acknowledgements

We would like to acknowledge Yuan Yu for his helpful comments. The research presented in this paper was performed with the support of DARPA F29601-01-2-0085 and NSF CCR-0121403.

References

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 143:1–70, 1999. Expanded version available as SRC Research Report 149 (January 1998).

[2] M. Abadi and L.Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] M. Bellare, R. Guerin, and P. Rogaway. XOR MAC's: New methods for message authentication using finite pseudorandom functions. *CRYPTO'95, Lecture Notes in Computer Science*, 963, 1995.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[5] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[6] S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, Nov. 1999.

[7] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and merkle trees for efficient memory authentication. In *Ninth International Symposium on High Performance Computer Architecture*, pages 295–306, 2003.

[8] P. Gutmann. The design of a cryptographic security architecture. In *The Usenix Security Symposium*, 1999.

[9] G. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[10] A. Huang. Keeping secrets in hardware: The microsoft XBox case study. Technical Report 2002–008, Massachusetts Institute of Technology, May 2002. http://web.mit.edu/bunnie/www/proj/anatak/AIM-2002-008.pdf.

[11] M. Kuhn. The TrustNo1 cryptoprocessor concept. Technical Report CS555, Purdue University, Apr. 1997.

[12] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2002.

[13] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[14] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference Architectural Support for Programming Languages and Operating Systems*, pages 168–177, Nov. 2000.

[15] P. Lincoln, M. Mitchell, J. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In M. Reiter, editor, *Proc. 5-th ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, 1998. ACM Press.

[16] K. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[17] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. In *IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[18] Microsoft palladium initiative - technical FAQ. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/news/PallFAQ2.asp.

[19] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW VIII*, page 98. IEEE Computer Soc Press, 1995.

[20] S. Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, 1996.

[21] W. Shapiro and R. Vingralek. How to manage persistent state in DRM systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[22] V. Shmatikov and J. Mitchell. Analysis of a fair exchange protocol. In *Seventh Annual Symposium on Network and Distributed System Security*, pages 119–128, 2000.

[23] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *Proceedings of the22nd National Information Systems Security Conference*, Oct. 1999.

[24] S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, Feb. 1998.

[25] U. Stern and D. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[26] The Trusted Computing Platform Alliance, 2000. `http://www.trustedpc.com`.

[27] J. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.