Layered Assurance Scheme for Multi-Core Architectures

J. Alves-Foss, X. He and J. Song
Center for Secure and Dependable Systems
University of Idaho
jimaf@uidaho.edu,[xhhe,song3202]@vandals.uidaho.edu

Abstract

As the demand for system virtualization grows, so does the need to securely virtualize a wider range of underlying physical resources which can be shared among multiple guest OSs. Recently, virtualization technology with hardware support has become available on commodity processors and can be utilized to reduce the size of the Trusting Computing Base (TCB). The design of a secure system requires architects to develop a system architecture that satisfies security policies. In this paper we propose an approach for specifying and verifying a layered assurance scheme for multi-core architectures.

1 Introduction

Over the past decade there has been steady activity and progress associated with multi-core architectures, especially after hardware-assisted virtualization technology has been integrated into such architectures, such as Intel VT-x in Intel Core i series and AMD-V in AMD processors. The concept behind virtualization technology is to virtualize underlying physical resources that can be shared among multiple guest OSs. The isolation properties of this technology have become attractive from a security perspective, as has the ability to inspect the details of a virtual machine's execution or the ability to tweak the isolation boundaries [5]. However, while providing services in a Virtual Machine (VM) offers a comparatively harder target for attackers and thus gains benefits, it also introduces some new difficult security challenges (e.g. VM-escape, VM-Based Rootkits) [6, 7, 17]. The addition of multi-core architectures exacerbates concerns as we now have true parallel execution and access to shared resources. Therefore, it becomes increasingly important to provide an assurance scheme for virtualized multi-core architectures.

The design of a secure system requires architects to develop a system architecture that supports implementation of various security policies. Enforcing security policies at the architecture level is attractive because it allows security concerns to be recognized early and can be given sufficient attention in the design stages. A security policy, determined by regulation and doctrine, defines "secure" for a system. The term "security policy for multi-core architectures" covers security requirements that protect multi-core systems from any unauthorized behaviors. In short, to develop a secure multi-core system, it is necessary to enforce a security policy framework as simple and as strong as possible to provide general guidance for secure systems on multi-core architectures. In this paper, we introduce a layered assurance scheme for multi-core architectures and illustrate (using 3 layers) how to formalize the framework.

We proceed in the remainder of this paper as follows. We begin by providing some background on secure architecture in Section 2, and then present our layered bottom-up assurance scheme in Section 3 and corresponding machine model in the next section, followed by an example of such a system in Section 5. We further examine related works in Section 6 and conclude with Section 7.

2 Background

Virtualization technology for commodity processors has entered the hardware extensions era. Multi-core architectures with hardware-assisted virtualization technology have become a prevalent platform for building virtual machine systems. One of

the key issues in virtualization technology is isolation. The main principle of isolation is to guarantee that any application in one VM cannot affect applications executing in a different VM, or that processes running in one VM cannot affect other VMs running in the same machine. If this security assumption is broken, then an attacker from one VM can have access to VMs in the same machine or even to the underlying host system (VMM or bare machine). To protect against this potential vulnerability in high-assurance systems, there are two main steps that must be completed. The first is to obtain a thorough understanding of the desired security properties of a multi-core system. This normally calls for a formal model of the system together with security proofs, often given in some mathematical or logical language. The second is to provide assurance that the implementation of a multi-core system realizes the more abstract formal model. The research community has spent considerable effort on the first area, such as some work on security architecture modeling and on providing security architecture design guidance for architects [14, 19, 3]. However, there is a lack of additional detailed guidance for multi-core systems from the perspective of hardware level. This paper addresses research that expands on this second area.

2.1 Security Architecture Models

We explore security architecture models by first defining the term security property. A security property is an instantiation of a security policy. There may be more than one property that satisfies a given policy. This section introduces a variety of security properties. A security framework generally assumes that there are at least two levels of users within a system, low-level (L) and high-level (H)[1]. The intuitive notion is that high-level users should be protected from low-level users, and information should not flow from high=-level users to low-level users. The purpose of a security property is to prevent low-level users from being able to make deductions about the events of the high-level users, to prevent unauthorized modification of data and to prevent users from affecting availability.

Many security properties are modeled on the concepts of event systems. An event system is a specification of the behavior of the system in terms of events (which could be mapped to state-transitions in a state-machine model). Associated to events are usually inputs and outputs of the system, where the outputs can be seen by users. A sequence of events is called $trace$. The set of traces of the system are usually modelled as the set implemented by the system. For the implementation to satisfy the security the security property, we analyze the set of traces.

To understand security properties, we first need a more precise understanding of traces and specifically of Low Level Equivalency Sets ($LLES$). An $LLES$ is a set of traces that have the same low-level events (in order) that a low-level user can observe. Formally, given a trace τ and a System S:

$$LLES(\tau, S) = \{s|\ \tau|L = s|L \wedge s \in traces(S)\}$$

The expression $\tau|L$ denotes the trace formed by removing from τ all events not in L. The function $traces(S)$ is the set of authorized traces in system S. The following security properties can be expressed with the $LLES$ notation [18]:

- **Noninference:** [15] A noninference security property requires that low-level users should not be able to infer information about high-level users;
- **Noninterference:** [8] A noninterference security property requires that high-level users are prevented from influencing the behavior of low-level users; otherwise, low-level users could infer information about high-level user activities;
- **Non-Deducible Output:** [9] A non-deducible output security property requires that low-level users cannot distinguish the events causing high-level users' output;
- **Separability:** [13] A separability security property requires that no interaction or information flow is allowed between low-level and high-level users.
- **Restrictiveness:** [12] A restrictiveness security property requires that changes in sequences of high-level events do not affect future possible sequences of low-level events.

A large body of evidence suggests that few designers ensure that their systems meet these properties. Besides, the separability security property is too strong because it doesn't allow any interaction between low-level users and high-level users. Consider a system where the only $TopSecret(TS)$ level behavior is to echo all $Secret(S)$ level output events to a TS level device for archiving. This system does not satisfy true separability. Zakinthinos and Lee [18] presented the "perfect security property (PSP)", the weakest security property that does not allow information flow from high-level users to low-level users, but does allow high-level outputs to be influenced by low-level events.

[1]This can be generalized into a lattice-based hierarchy, and does not necessarily imply military-style security levels.

2.2 Perfect Security Property

This section summarizes the details of the perfect security property (PSP) as presented by Zakithinos and Lee [18]:

$$\forall \tau \in traces(S) : \tau | L \in LLES(\tau, S) \wedge \forall p, s : p^\wedge s \in LLES(\tau, S) \wedge s | H = \langle\rangle : \\ \forall \alpha \in H : p^\wedge \langle \alpha \rangle \in traces(S) \Rightarrow p^\wedge \langle \alpha \rangle^\wedge s \in LLES(\tau, S)$$

In this formulation, the trace $p^\wedge s$ refers to the trace formed by concatenating traces p and s, the trace s has no high-level events, while p might have some high-level events. The possibility of α occurring between p and s is only dependent on the preceding high-level events in p. The idea behind PSP is the same as that behind Separability. All possible high-level activity and interleavings must be possible with all low-level activity. The difference is that PSP allows high-level outputs to be dependent on low-level events. This does not reduce security since the low-level user still will not know how he has influenced high-level outputs. One other benefit of PSP is that it is a composable property (as are separability and restrictiveness). If we decompose a system into individual components, and prove that each component satisfies PSP, then the composite system will satisfy PSP – a property that is surprisingly not preserved by all security properties.

Unfortunately, with all of these security properties, analysts are often stuck with a single level of abstraction. Even with composability, we must use the same specification of events and security properties, no matter which level of abstraction of implementation we are evaluating. A lower level may not support concepts of security levels in the context of an application, but may support concepts of separation and controlled information flow. A good layering approach will allow us to transition between levels of abstraction mapping lower level security properties to higher level properties.

Real systems will also use multiple components, working together, to implement the system functionality and security policy. It is not necessarily true that each component will satisfy the full security property, but may satisfy a subset that when combined with other components is sufficient to satisfy the full system security property. A layering approach must also support the concept of sub-policies and partial implementation of security properties with composition.

3 3-Level Framework

Ideally we wish to specify a security framework with any number of levels and components. However, for the purposes of this paper, we restrict ourselves to an exemplary 3-level system in this section. Assurance that a system satisfies a security policy will require examination of all this software to be sure that there are no accidental or malicious mechanisms that could allow malicious users or processes to reach the trusted region. Due to a wide variety of malicious behaviors, assurance would most likely have to specify exactly what each element of software is intended to perform, and to provide evidence that it does it correctly. Similarly, we analyze a multi-core architecture by examining each component from hardware level, hypervisor level until user level. The 3-Level Bottom-Up security policy architecture and will be enforced in a lightweight virtual machine monitor, called IAVMM [10], which is developed for security analysis of Intel 64 architectural features. To be specific, we process it with 3 subsequent stages:

- Identify and examine multi-core hardware architectural features and enforce security policy in IAVMM level;
- Decompose the security policy into components that can be mapped into hardware level;
- Verify that VMM and Hardware level security policy satisfies user level security requirement.

We demonstrate an application system (see Figure 1) to illustrate how to accomplish these steps. The system is simple enough to avoid complexity for the purposes of discussion but sufficient to illustrate the concepts of our approach. Informally, it is a boxes-and-arrows diagram in which we want no channels for information flow other than those explicitly indicated by arrows. We present this architecture through the analysis of a two-partition system with an assumption that each partition system only contains one virtual machine system (VMS). In our work, two generic secure building blocks, MMR and GC, are introduced in the VMM level to enforce security policies. MMR (Message Management Router) does the partition identification, message labeling and routing of typed messages. The GC (Guard Controller) provides a guard for checking authorized message behavior. Specifically, MMR encodes messages with security levels, and GC enforces security policies.

The specifications of this system are (Figure 1(a)): two single-level users, one top-secret and one secret, communicating with a multi-level database (DB). The database implementation is simple and assumes that messages from the users are tagged correctly with the correct security level. Assuming a Bell-LaPadula Model [2], which consists of the simple security property (no “read up”) and the *-property (no “write down”), is applied for user level, we need to make sure that the messages are

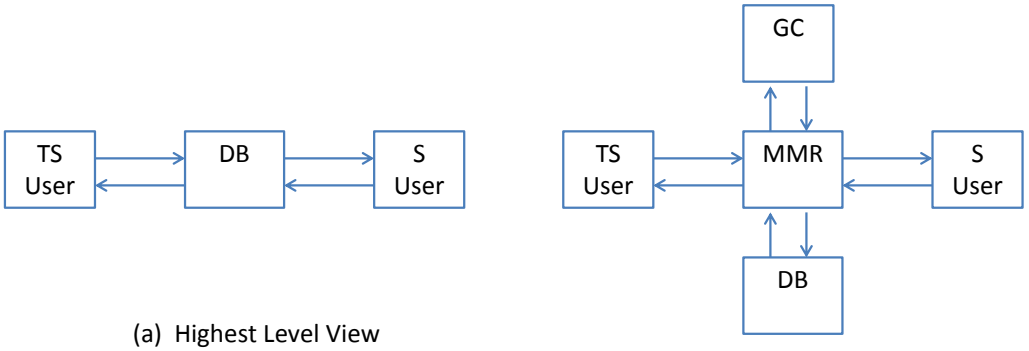


Figure 1. Assurance architecture of the multi-core system

Definition 1. The formal model of a state machine M is defined as:

- $M = \langle \Sigma, \sigma_0, T \rangle$
- Σ is the set of states of the system
- Initial State: $\sigma_0 \in \Sigma$
- $T : \Sigma \rightarrow \Sigma$ defines the allowed transitions between states.
- The notation $\sigma(p)$ denotes the substate of σ that corresponds to the named resource, p, in the system.

labeled correctly so that the DB will not violate the security property. The system designer can design an application level security system (Figure 1(b)) where messages from the users are sent through a MILS message router which first sends the messages to a guard subsystem (GC) which ensures that the messages are tagged correctly and then sends them to the DB. The MMR reroutes return messages to the correct users. A system like this can be fully specified and proven to satisfy PSP.

The system designer can take this implementation, place each user the DB, MMR and GC into separate virtual machines and support the whole system on a multi-core platform. The next step of our validation of this system then involves decomposing and mapping the high level policy (with knowledge of *Secret* and *Top Secret* Users), into the VMM and eventually the hardware level of multi-core architecture. To address this problem, we provide security mechanisms to implement them in memory paging system.

Moreover, even if the system has been compromised, the policy architecture should be responsible for taking appropriate action. It is possible that, with the 3-level security policy framework designed specifically for this implementation on Intel Core i7 series, we could provide software solutions or safeguards for some of the security concerns found in this research. The results of our work could serve to guide future security development for the Intel Core i7 series.

4 Event System Model

In this section, we define a formal model of events in our system, and a mapping between these events and the state transitions defined in the preceding section. Definition 1 presents the traditional model of a state machine. We can refine this model in many ways to represent a system that supports multiple virtual machines and multiple physical machines. The extensions require two things, first is a mapping from the simple model into a model that can represent multiple interacting state machines, one for each virtual machine, and second is a model that allows for the concurrent execution environment of a multi-core processor. Definition 2 presents a refined model of a state machine, represented as a composite of multiple constituent state machines.

Definition 2. The formal model of a state machine M can be subdivided into a collection of composite state machines $\langle \Sigma^i, \sigma_o^i, T^i \rangle$, each representing a virtual machine of the system, or the hypervisor.

- $M = (M^1, M^2, \ldots, M^n)$ n-tuple representing the individual state machines in the composite machine, where $M^i = \langle \Sigma^i, \sigma_0^i, T^i \rangle$
- $\forall \sigma \in \Sigma : \sigma = cs(\sigma^1, \sigma^2, \ldots, s^n)$ where $\sigma^i \in \Sigma^i$
- Initial State: $\sigma_0 = cs(\sigma_o^1, \sigma_o^2, \ldots, \sigma_o^n)$,
- The notation $cs(s_1, \ldots, s_n)$ denotes the composite state of the system.
- The extraction function $\mathcal{S}^i(\sigma) = \sigma_i$ returns the portion of the composite state relevant to sub-machine i.
- $T(\sigma) = cs(\tau^1(\mathcal{S}^1(\sigma)), \tau^2(\mathcal{S}^2(\sigma)), \ldots, \tau^n(\mathcal{S}^n(\sigma)))$ where $\tau^i \in T^i$

State Machine Policy 1: The intersection of substates must be restricted such that execution of τ^i does not interact with substate σ^j in violation of the security property.

State Machine Policy 2: If the execution of τ^i as part of $\tau \in T$ modifies a component of substate $\sigma^j (j \neq i)$, then the transition τ^j in τ must also specify that modification.

In this model, the full machine state is represented as a composite state $cs(s_1, s_2, \ldots, s_n)$. This is not an n-tuple, but the result of a composition function cs. This allows us to model systems where the constituent state machines share some portion of the overall machine state. Although we allow for shared state, we model system transition as a composite of individual transitions of the component state machines. This model allows for execution of multiple sub-machines simultaneously (modeling multi-core and hyper-threading processing).

Note, the model allows too much interaction between sub-machines, hence the inclusion of the two *State Machine Policies*.

- *State Machine Policy 1, (SMP1)*, is included to enforce the idea that we must only allow for authorized interactions between the sub-machines. Only if the overall policy permits interaction can sub-machines can share state.
- *State Machine Policy 2, (SMP2)*, exists for consistency in the models. If sub-machine i can cause a change in state for sub-machine j, then that possible change of state should be modeled as a possible transition in sub-machine j. This allows for a cleaner independent analysis of each machine, and then analysis of their composition.

Definition 3 defines an event as an action, a, acting on behalf of a subject, s, using data from a set of resources (objects), r and possibly modifying members of a set of resources, w. This generic representation of event can be used at any layer of our architecture, with an understanding that a representation at one layer will be an abstraction of the representation at the next lower layer. Our state-machine model presented in the previous section, defines transitions as atomic changes to the state of a system. Events are more complex actions that may involve multiple interactions with the state, and are therefore represented/implemented by a sequence of state transitions as shown in Definition 3[2]. Events that are specified as atomic actions do not allow for interference during their sequence of state transitions, whereas synchronizing events do allow interaction with other synchronizing events.

In Definition 3 we introduce two *Event Policies*. These policies place restrictions on the semantics of the events, specifically on the mapping of events to a sequence of transitions and the use of objects.

- *Event Policy 1 (EP1)* specifies that the sequence of transitions may not modify any object that is not specifically listed in the *write-set*.
- *Event Policy 2 (EP2)* is a specification of determinism and completeness, which states that for two different states of the system, if the contents of objects in the *read-sets* of the event are the same, then the contents of all objects in the *write-set* will be the same after the event.

[2]We will need to check if (a, s, r, w) is permitted for a particular state of the system.

Definition 3. The formal model for events E are defined as follows:

- $\mathcal{E} = \{(a, s, r, w) \mid a \in \mathcal{A}, s \in \mathcal{S}, r, w \in \mathcal{P}(\mathcal{O})\}$ is the set of events of the system.
- $\mathcal{S}$ is the set of subjects
- $\mathcal{O}$ is the set of objects and $\mathcal{P}(\mathcal{O})$ is powerset (set of subsets) of O.
- r and w are two (not-necessarily disjoint) subsets of objects that are accessed by action a; the *read-set* and the *write-set*.
- $\mathcal{E}$ is the set of events, where events correspond to state transition chains

  let $\tau = \tau_0, \tau_1, \ldots, \tau_n \in T^*$ be a sequence of state transitions corresponding to event $e = (a, s, r, w)$ such that:

  $$\tau(x) = \tau_n(\tau_{n-1}(\ldots \tau_1(\tau_0(x)) \ldots))$$

  let σ be the state of the system prior to execution of event e and $\sigma' = \tau(\sigma)$ be the state after execution of τ.
- Events are classifed as *atomic* or *synchronizing*

The formal model for events in a composite system are:

- $\mathbb{E} \subseteq \mathcal{P}(E)$. At any time there may be any number of "active" events in the system.
- Let $\overline{e} \in \mathbb{E}$ be a set of active events, and $e_i, e_j \in \overline{e}$ be two different active events.
  - If e_i and e_j are atomic events, then $(e_i.r \cap e_j.w) = (e_i.w \cap e_j.w) = (e_i.w \cap e_j.r) = (e_i.r \cap e_j.r) = \emptyset$
  - If e_i is a synchronizing event, and $(e_i.r \cap e_j.w) \neq \emptyset \vee (e_i.w \cap e_j.w) \neq \emptyset \vee (e_i.w \cap e_j.r)$ then e_j and e_i must be *partner synchronizing events*.

Event Policy 1: $\forall o \in \mathcal{O} : o \notin w \Rightarrow \sigma'(o) = \sigma(o)$

Event Policy 2: $\forall \sigma_1, \sigma_2 \in \Sigma : (\forall o \in r \cup w : \sigma_1(o) = \sigma_2(o)) \Rightarrow (\forall p \in w : \sigma_1'(p) = \sigma_2'(p))$

Let us consider the ramifications of these policies. First, since the event is specified with respect to an abstraction of the system state (sets of objects), it is possible that a portion of the system state that is not part of a specified object could be modified by the event. $EP1$ does not prohibit this. For example, the instruction pointer and system clock registers are part of the system state, but may not be relevant in a discussion of events at a higher level of abstraction. However, even with the incompleteness of this policy, $EP2$ ensures that the contents of these non-modeled state components can not interfere with the computation of the event. For example, if an event wishes to copy the instruction pointer into a data structure, $EP2$ requires that the instruction pointer be specified as an object in the *read-set* of the event. Otherwise, there may be two states of the system that differ only in the program counter and would therefore satisfy the antecedent of the implication but not satisfy the consequent, violating the policy.

A layered approach to using the event model can be readily developed using the standard commuting theories, as depicted in Figure 2 and Definition 4. We implement higher level events through sequences of lower level events, defined by the implementation function $\mathcal{I}$. We can abstract the lower level state to the higher level state with the abstraction function $\mathcal{A}$.

Although this sounds straight forward, there are many issues that must be addressed to ensure that this abstraction does not violate the security properties of the system. These issues include:

- State Machine and Event Policies defined earlier must hold.
- There must be a clear abstraction mapping between objects of the two levels (referenced in the read-sets and write-sets of events at both levels).
- For all *hidden* components, x, of a state, the value of the hidden component does not change the abstraction or inter-

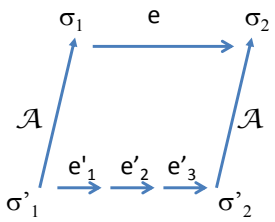


Figure 2. Standard Commuting Diagram for Implementation Correctness

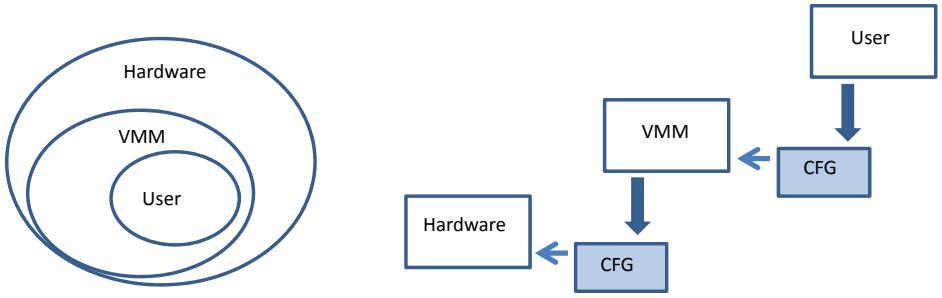
Definition 4. The formal model for commuting:

- $\mathcal{I}$ is an implementation function (mapping event e at a high level to a sequence of events $e_1, e_2, \ldots$ at a lower level.
- $\mathcal{A}$ is a abstraction function that maps the state of the lower level to the higher level of abstraction.
- If $\mathcal{I}(e) = e'_1, e'_2, e'_3$; τ is the transition sequence for e and $\tau'_1, \tau'_2, \tau'_3$ are the transition sequences for e'_1, e'_2, e'_3 respectively, then these functions commute if each of the following is true
  - $\tau(\sigma_1) = \sigma_2 \quad \wedge \quad (\tau'_3 \circ \tau'_2 \circ \tau'_1)(\sigma'_1) = \sigma'_2$
  - $\mathcal{A}(\sigma'_1) = \sigma_1 \quad \wedge \quad \mathcal{A}(\sigma'_2) = \sigma_2$

pretation functions, or cause a violation of the security property or correctness of the commuting diagram.

Now, with the model in place, we can define security properties, such as non-interference, or PSP at the top level of abstraction. For that property to hold, the issues mentioned above must be satisfied. However, it is not required that the lower level have the same "view" of the world; just that it support the higher level view. This is the normal model for computing languages, high level abstractions are refined to lower level implementations, and will work for security as well.

5 Example Layered Assurance



(a) Hierarchical Policy Model

(b) Hierarchical Implementation

Figure 3. Layered Policy and Implementation Models

Consider a system such as that depicted in Figures 1 and 3. In Figure 3(a) we have the hierarchy of authorized states of the system. The hardware supports a large set of authorized system states (states that will not cause a hardware failure

or exception). However, we have the ability to configure the hardware (the cfg box in Figure 3(b)) to limit these states. This configuration consists of setting execution modes and programming MMU page tables, VM modes, etc. The VMM configures the hardware to a limited subset of the allowable states, and the User can configure the VMM.

At the highest layer of abstraction, we have the user view of the world, as depicted in Figure 1(a) and the system designer view (Figure 1(b)). At the highest level, we have an understanding of the security policy, we have subjects and objects mapped to security levels (e.g., Top-Secret and Secret). At the system designer view we understand the security properties of the system. The system designer defines executable entities such as users mapped with security levels, communication channels, and supporting hypervisor level resources such as guards and Extended Page Tables (EPT); all in support of the policy. The system implementor needs to map events such as "VM1 writes message to a specific location inside VM2" into a sequence of events which include locating the memory address of the message by traversing the EPT of the VMS2[3], verifying the memory writeability for VM1, and storing the message into the location VM1 specifies.

We must prove that the security property enforced at this level supports the high level security policy. We do that with many assumptions about the underlying infrastructure and the security policy exported by the lower level. Here we can possibly use PSP and the LLES definitions, but may need to add concepts on non-transitive information flow to allow trusted implementations of the hypervisor and EPT.

5.1 Hardware Layer

At the lowest level we have the hardware platform which supports the concepts of execution in a context. A context is defined as the execution mode (e.g., supervisor/user, privilege ring, VM status) and the set of available resources (e.g., the memory maps in the MMU). The hardware provides mechanisms to set and change the configurations of contexts, and to perform context changes.

Subjects in the hardware model are mapped to the contexts that the hardware supports. All events are bound to the current executing subject of the hardware. In a multi-core model, there may be multiple subjects, one running on each logical processor, or on a collection of processors. The objects of the hardware are the physical resources of the hardware, memory, devices, registers, the MMU, etc. In the end, the hardware exports a model of an executing set of systems, the individual logical processors, and current executing contexts.

The security policy of the hardware does not directly map to the concepts of high and low-level users (as discussed in Section 2. However, we can still map the security policies of the hardware to allowable sequences of events $e_1^{hw}, e_2^{hw}, \ldots, e_n^{hw}$. These will be of the form $\overline{e_1^{hw}}, \overline{e_2^{hw}}, \ldots, \overline{e_n^{hw}}$ where each $\overline{e_i^{hw}}$ will be sequence of events for current contexts, or the context switch events. The set of traces for the hardware, $traces(HW)$, will only contain those events that are allowable within the current contexts. If the context supports virtual memory and MMU-based memory maps, the events will correspond to available virtual memory accesses and MMU maps. At this level of abstraction, verification of the correct behavior of the system includes verification that the hardware supports the configuration data and does not violate the contexts. The security property of the hardware must clearly specify the limitations of the hardware execution model and configuration. We should not attempt to model security levels, or user intent at this level, just the correct implementation of the configuration data.

5.2 VMM Layer

The VMM layer is responsible for defining the contexts of the hardware, and thus will only authorize a subset of the allowable states with a more restrictive policy (Figure 3). The hardware provides the basic security mechanisms of isolation: *virtualization*, *virtual memory*, *memory management*, and *context switching*. It also supports multiple execution units. The hardware supports execution of each logical CPU core in either root-mode or in a guest (virtual machine) mode. The VMM configures the memory maps, assigns usage of the cores to the VMs and manages transitions in and out of virtualization mode.

The events of the VMM now correspond to actions of the individual VMs (in the environment of the VMM) and the control events of the VM (configuring the hardware, establishing the VM contexts). The VMM model of the system partitions the physical resources of the hardware into the subset of resources available to the individual VMs. Events at the VMM level will mostly still be at the same granularity of the hardware level, with additions for VMM specific actions (e.g., creation of a VM, swapping in/out a VM). However, the subjects in the VMM are now mapped to individual VMs and the VMM itself. The objects of the system are still mostly the hardware resources, but also now the VMM data structures representing the

[3]VMS2 refers to Virtual Machine System 2, which contains not only VM2, but also VMM2 system.

context's of the VMs, possible buffers and other internal resources. We need a mapping of these objects onto the hardware and a mapping of the VMM-specific events into hardware events.

In the end, the VMM exports a model of an executing set of virtual systems, the individual VMs, and current executing contexts for those VMs. We still can not model security levels, but we have now separated the hardware (time and space) into VM contexts, and can have VMM rules for behavior of those VMs. For example, if the VMM provides some simple services for inter-VM communication, or for VM management, the VMM configuration data must indicate permissions for access to those services. The user will utilize the services of the VMM through configuration data, and verification of the VMM assures that it supports the policy specified by the configuration data (e.g., no communication occurs between VMs unless authorized in the VMM configuration data).

5.3 User Layer

At the highest layer of abstraction, we have the user view of the world, as depicted in Figure 1(a) and the system designer view (Figure 1(b)). The VMM does not have a concept of top secret or secret, routers, databases or guards. The VMM has the concepts of virtual machines, shared memory regions, inter VM communication, and private resources. The VMM also maps and schedules VMs to particular cores. The configuration of the VMM is what enables us to provide the abstraction to the user level. The user level designer can utilize the VMM concepts of separation and controlled information flow to design a system similar to Figure 1(b). This system will implement guards and routers to ensure properly labeled and filtered messages between two users (with guards, routers and users all mapped to individual virtual machines).

The system designer defines executable entities such as users mapped with security levels, communication channels, and supporting application level resources such as guards and routers; all in support of the policy. The system implementor needs to map events such as "send message to DB" into a sequence of events which include sending message to MMR, routing from MMR to GC, tagging the message in GC, sending it back to MMR, and then routing it from MMR to DB. We must prove that the security property enforced at this level supports the policy. We do that with many assumptions about the underlying infrastructure and the security policy exported by the lower level. Here we can possibly use PSP and the LLES definitions, but may need to add concepts on non-transitive information flow to allow trusted implementations of the GC, DB and MMR.

6 Related Work

Virtualization has existed as a technique since 1960s, but has recently attracted more and more attention from the research and industry communities. Recently, an industry-wide consensus has emerged that a VMM with hardware-assisted virtualization technology can offer a number of benefits. However, various VMM vulnerabilities (e.g. VM escape) allow the program running in a guest virtual machine to completely bypass the hypervisor layer thereby compromising the host system. Bratus et al. [4] propose simple, natural modifications to commodity x86 hardware to provide kernel security isolation by extending and reinterpreting the x86 segmentation mechanism, an alternative solution to the existing hardware protection mechanism. There exists no solid theory of security description at multi-core architecture level, which is a significant motivation to provide an approach to secure system design in which the desired security properties of the multi-core system are formally described.

There has been some work on security architecture modeling and providing security architecture design guidance for architects [14, 19, 3]. However, none of them is specific for multi-core systems, especially multi-core architectures with hardware-assisted virtualization technology. The principle of separation kernel, introduced by Rushby [16] in the early 1980s, is to divide all resources into blocks such that the activities in one block are isolated from activities in another block, unless an explicit communication has been established, and security policies are enforced by trusted applications running in some of those blocks. The conceptual model of separation kernel provides an ideal foundation to create secure VMMs, since the primary common functionality of both of them is to prevent illegal information flow between isolated blocks. Levin et al. [11] extend and provide a separation kernel protection profile to enforce a compound security policy with requirements at the gross partition level as well as at the granularity of individual subjects and resources, hence enhancing protection for secure systems. Multiple Independent Levels of Security/Safety (MILS) [1] is a high-assurance architecture for secure information sharing by separating out the security mechanisms and concerns into manageable components, such as separation kernels, partitioned file systems, and partitioning communications systems that deliver the required guarantee of separation.

7 Conclusion and Future Work

Intel and AMD have offered hardware-assisted virtualization technology to enable efficient virtualization in their CPU design. These advances have created a window of opportunity to provide a profound impact on the reliability of multi-core systems, however, a high-assurance scheme must be proven to correctly implement the security functions in its specifications and effectively mitigate risks to a level commensurate with the value of the assets it protects [11]. Therefore, the design of a secure system requires architects to develop a system architecture that satisfies necessary security policies. An important contribution of our work is the modeling of a layered bottom-up security policy framework in terms of multiple secure architectures that are related by formal mappings. We have proposed an approach to secure and evaluate multi-core architecture design.

Research efforts are still underway to verify the security requirements in our security policy framework, and provide the safeguards to mitigate the risks.

Acknowledgements

This material is based on research partially sponsored by the Air Force Research Laboratory under agreement number FA8750-10-2-0134. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

References

[1] J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–247, 2006.

[2] D. E. Bell and L. J. LaPadula. Security computer systems: Mathematical foundations and model. Technical report, MITRE Corp., Bedford, Mass., 1974.

[3] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *IEEE/AIAA Digital Avionics Systems Conference*, pages 1.C.2 (1–14), Los Alamitos, California, 2008.

[4] S. Bratus, P. C. Johnson, A. Ramaswamy, S. W. Smith, and M. E. Locasto. The cake is a lie: Privilege rings as a policy reousrce. In *Proc. ACM Workshop on Virtual Machine Security*, pages 33–38, 2009.

[5] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. Traps, events, emulation, and enforcement: managing the yin and yang of virtualization-based security. In J. Nieh and A. Stavrou, editors, *Proc. ACM Workshop on Virtual Machine Security*, pages 49–58, 2008.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *HotOS*, pages 133–138, 2001.

[7] J. Fisher-Ogden. Hardware support for efficient virtualization. 2006.

[8] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[9] J. D. Guttman and M. E. Nadel. What needs securing. In *CSFW*, pages 34–57, 1988.

[10] X. He and J. Alves-Foss. A lightweight virtual machine monitor for security analysis on Intel 64 architecture. *Journal of Computing Sciences in Colleges*, 2011.

[11] T. E. Levin, C. E. Irvine, and T. D. Nguyen. Least privilege in separation kernels. In *Proc. of the International Conference on Security and Cryptography*, pages 355–362, 2006.

[12] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[13] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[14] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure software architectures. In *Proc. IEEE Symposium on Security and Privacy*, pages 84–93, 1997.

[15] C. O'halloran. A calculus of information flow. In *Acte de ESORICS 90, Toulouse*, pages 147–159, Oct. 1990.

[16] J. Rushby. Design and verification of secure systems. *ACM*, 15(5):12–21, Dec. 1981.

[17] J. Rutkowska. Security challenges in virutalized environments. In *RSA Conference*, Apr. 2008.

[18] Zakinthinos and Lee. A general theory of security properties. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

[19] J. Zhou and J. Alves-Foss. Security policy refinement and enforcement for the design of multi-level secure systems. *Journal of Computer Security*, 16(2):107–131, 2008.