A

Model Checking Distributed Mandatory Access Control Policies

PERRY ALEXANDER, The University of Kansas
LEE PIKE, Galois, Inc.
PETER LOSCOCCO, National Security Agnency
GEORGE COKER, National Security Agency

This work examines the use of model checking techniques to verify system-level security properties of a collection of interacting virtual machines. Specifically, we examine how local access control policies implemented in individual virtual machines and a hypervisor can be shown to satisfy global access control constraints. The SAL model checker is used to model and verify a collection of stateful domains with protected resources and local MAC policies attempting to access needed resources from other domains. The model is described along with verification conditions. The need to control state-space explosion is motivated and techniques for writing theorems and limiting domains explored. Finally, analysis results are examined along with analysis complexity.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security, Verification, Design

Additional Key Words and Phrases: Access control, model checking, virtualization

ACM Reference Format:
Perry Alexander, Lee Pike, Peter Loscocco, and George Coker, 2015. Model Checking Distributed Mandatory Access Control Policies. *ACM Trans. Info. Syst. Sec.* V, N, Article A (January YYYY), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Access control is a classic example of a cross-cutting, system-level property that cannot be implemented or evaluated in a single component. In modern distributed systems, access control is implemented by multiple, communicating components that together must satisfy system-level properties. Among the more common mechanisms for implementing security is *mandatory access control (MAC)* where resource access is governed by a collection of rules defining rights that processes have with respect to system resources. When examining system-level security properties, it is essential to consider access control holistically even when assured that individual components implement access control correctly. While verifying access control policies for individual components to reveal component level problems is essential, integrated analysis is still required to assure system-level security properties.

A common concept in virtualization is implementing virtual platforms as collections of virtual machines (VMs). Each virtual machine provides services to the platform

This work is sponsored by the United States Department of Defense.
Author's addresses: P. Alexander, Information and Telecommunication Technology Center and Department of Electrical Engineering and Computer Science, The University of Kansas; L. Pike, Galois, Inc; P. Loscocco and G. Coker, National Security Agency.

that range from device access to full operating systems. Although the promises of this new technology are significant, virtualization introduces new challenges to developers when making assurances about their systems. Among these challenges is realizing system-level security policies as a collection of services running on individual, decentralized virtual machines.

In this work we consider mandatory access control in a decomposed Xen Dom0 [Barham et al. 2003] designed to support measurement, remeasurement and attestation [Haldar et al. 2004; Coker et al. 2008, 2011] using a Trusted Platform Module (TPM) [Trusted Computing Group 2007] and Virtual TPM (vTPM) [Berger et al. 2006] based infrastructure. In the Xen virtualization environment, the Dom0 domain delivers platform and operating system services to other domains. Dom0 runs a monolithic Linux kernel that must operate in a maximally privileged mode. Recent research examines the decomposition of Dom0 into smaller domains, each providing a system service with least privilege [Cihula 2006; Coker 2007]. The system motivating this work decomposes Dom0 into a collection of infrastructure and service providing virtual machines. Each virtual machine operates with least privilege, implementing their own access control policy.

The collection of virtual machines providing Dom0 services—called the *Supervisor Virtual Platform* (SVP)—must exchange data and share resources via Xen hypervisor provided Inter-VM Communication (IVC). SVP domains must interact to perform their tasks and guest operating systems must have access to devices, data, and operating system primitives. Yet, if the entire platform is to be trustworthy, access to data and resources must be controlled holistically. Specifically:

— Secrets must be protected and held confidentially
— Keys protecting secrets must themselves be protected
— Virtualized services must be restricted to authorized parties

while still allowing the platform to boot and run as expected when properly configured.

To check collections of domains for these properties, our model defines and composes models enforcing access control policies for the hypervisor and individual domains providing services. The hypervisor policy governs communication between domains while domain specific policies govern access to resources. The model implemented in SAL [Bensalem et al. 2000] checks for successful boot and absence of disallowed resource accesses. Successful boot of a domain occurs when the domain acquires all needed resources while successful boot of the platform is successful boot of all its domains. Policies, domains, and resources are easily modified to support the SVP designers' need to explore alternatives. As such the approach and model are easily applied to a wide collection of similar problems.

2. SYSTEM ACCESS CONTROL ARCHITECTURE

Stated in the classical security vocabulary, *principals* in our system are Xen virtual machines—called *domains*—while *objects* are resources—services and data—required for boot and operation. For example, a virtual platform controller is a domain that requires data from a store and virtual machine build services from a domain builder. Each domain attempts to acquire resources it needs by sending messages to domains it believes provide those resources. A domain is considered successfully booted if it acquires all resources necessary for its operation. Access control decisions in scope for our analysis govern: (i) when communication between domains is allowed; and (ii) when a domain should have access to an object. These access control decisions are made collectively among numerous virtual machines and the hypervisor. Further complicating access control decisions is composability of resources. For example, a key and encrypted data that are not separately problematic together result in clear data that should remain confidential.

Access control is implemented in each virtual machine and the hypervisor using an SELinux [Mayer et al. 2007] style Flask [Spencer et al. 1999] implementation. Each domain has an associated Flask configuration file that reflects its associated access control policy. As is standard practice, domains and resources are assigned types used to encode access privileges. We assume that security types are assigned with integrity when domains and resources become available. Specifically, when a new domain is deployed, the build system is trusted to faithfully assign the correct security type to the new domain.

The Xen hypervisor is a distinguished principal that provides and controls access to IVC mechanisms. IVC provides a trustworthy medium for requesting resources and delivering services and data. The hypervisor is unaware of communication content—it is only aware that one domain wishes to communicate with another and provides a mechanism if allowed. Once the communication mechanism is established, the hypervisor does not monitor message content. It is trusted to provide messaging services with integrity and confidentiality among communicating domains. Specifically, it will not tamper with messages nor will it leak messages to unauthorized domains.

The policy subset governing IVC communication between principals is referred to here as *Platform MAC*. When one virtual machine attempts to communicate with another, Platform MAC is used to determine whether an IVC channel should be established. We view Platform MAC in our work abstractly as a single policy statement over communication. We do not try to specify its implementation as would be required in an actual SELinux policy file.

Each additional domain beyond the hypervisor is configured separately with its own access control policy. This policy is referred to as *Local MAC* and governs access to local resources. When a request arrives via IVC, local MAC is used to determine whether that request will be fulfilled and how the principal is allowed to respond.

Both Platform MAC and Local MAC for individual domains may change during the platform operation. For example, Platform MAC is always updated from a boot MAC policy to a run time policy when the boot sequence completes. It may also be updated when new domains are added to the virtual platform. Similarly, Local MAC may be updated by domains based on system observations or a directive from an authorized domain.

Figure 1 represents how Platform MAC and Local MAC both play a part in any object request. If one domain wishes to request an object from another, the hypervisor's Platform MAC first determines if the request should be allowed. This decision is based only on the security types of the communicating domains. If the requesting message arrives at the domain potentially providing the object, that domain's Local MAC policy determines whether the request will be honored. Thus, all access control decisions are distributed across the virtual platform.

The distributed and stateful nature of domains, resources and MAC policy complicates analysis. Because system resources traditionally managed by Dom0 are now controlled by a collection of virtual machines, it is necessary for access control to be distributed among those virtual machines and the hypervisor. As there is no central control, domains act independently without coordinated decision making. Furthermore, during boot, shutdown and normal system operation, domains, resources, and MAC policy may change state. To establish platform trustworthiness properties, the collection of distributed, dynamic policies must be analyzed at the system-level. Analyzing individual policy components within the hypervisor or individual domains may establish trust in local properties but is insufficient for guaranteeing platform properties.

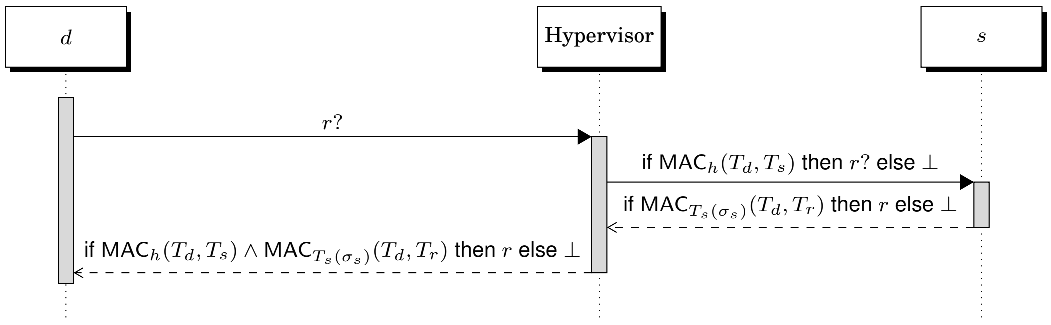

d
Hypervisor
s
$r?$
if $\mathsf{MAC}_h(T_d, T_s)$ then $r?$ else $\perp$
if $\mathsf{MAC}_{T_s(\sigma_s)}(T_d, T_r)$ then r else $\perp$
if $\mathsf{MAC}_h(T_d, T_s) \wedge \mathsf{MAC}_{T_s(\sigma_s)}(T_d, T_r)$ then r else $\perp$

Fig. 1. Sequence diagram documenting d requesting r from s. Communication from d to r may be disallowed by the hypervisor; access to r may be disallowed by s; or r may be provided. Note that $r?$ denotes a request for r and $\perp$ denotes a failed request.

3. SEMANTIC MODEL

Our underlying semantic model for MAC policy extends the model presented by Hicks et al. [2007]. Distinctions include: MAC policy changing with domain state; the hypervisor MAC policy and domain MAC policy together governing all requests for resources and services; and assessment must be performed over domains as a system rather than over individual domains in isolation.

3.1. Two-Phase Access Control

Formally, let T_i be a type associated with resources and domains whose state is invariant. Let $T_i(\sigma)$ be a similar type dependent on state σ. Intuitively, $T_i(\sigma)$ represents a principal whose state changes and impacts access control decisions made with respect to it. Denote p as belonging to type T_i by $p : T_i$ and similarly for $p(\sigma) : T_i(\sigma)$. Given a destination domain, $d(\sigma_d) : T_d(\sigma_d)$, making a request and a source domain, $s(\sigma_s) : T_s(\sigma_s)$, controlling the requested object, $r : T_r$, d in state σ_d is authorized to access r if: (i) the hypervisor's platform MAC policy allows communication between domains of types T_d and T_s; and (ii) local MAC policy associated with domains of type T_s in state σ_s allows domains of type T_d access to the object. This is expressed as a relation, α:

$$\alpha(d, s(\sigma_s), r) = \mathsf{MAC}_h(T_d, T_s) \wedge \mathsf{MAC}_{T_s(\sigma_s)}(T_d, T_r)$$

where MAC_h is the hypervisor's MAC policy and $\mathsf{MAC}_{T_s(\sigma_s)}$ is the MAC policy associated with $T_s(\sigma_s)$.

MAC_h specifies allowed communication among domain types. Given domain types T_d and T_s, MAC_h specifies whether a domain of type T_d should be allowed to make requests of a domain of type T_s independent of state. As the hypervisor is agnostic to domain states and communication content, MAC_h depends only on domain type and cannot depend on domain state. MAC_h is referred to as the Platform MAC.

$\mathsf{MAC}_{T_s(\sigma_s)}$ specifies allowed access among resources and domain types. Given a resource of type T_r that a domain of type T_s in state σ_s controls, $\mathsf{MAC}_{T_s(\sigma_s)}$ specifies access types for a domain of type T_d. For individual MAC access control decisions, $\mathsf{MAC}_{T_s(\sigma_s)}$ depends on the state σ_s which T_s is in. However, domain s cannot be aware of the state of domain d and can depend only on its type. $\mathsf{MAC}_{T_s(\sigma_s)}$ is referred to here as local MAC.

3.2. Domain State

The state of domains is expressed as a function $\sigma : T_r \rightarrow T_{status}$ where T_r is the object type and T_{status} the status type. Given a resource $r : T_r$, its status in σ is $\sigma(r)$. T_{status} is defined as the enumerated type:

$$T_{status} = \{\texttt{needed}, \texttt{notNeeded}, \texttt{clear}, \texttt{encrypted}, \texttt{sealed}\}$$

where values indicate resource status.

The `needed` and `notNeeded` values express cases when the domain does not currently have access to an object. If an object is `needed`, the domain will try to acquire it in a `clear` state. If an object is `notNeeded` it will be ignored.

The `clear`, `encrypted`, and `sealed` status values express cases when the domain has access to an object. The `clear` status value indicates a domain has access to the object in the clear—the desired final state for a `needed` object required by the domain. The `encrypted` and `sealed` values indicate that access to an object has been obtained, but keys and/or encryption facilities are required to use it. The distinction between `sealed` and `encrypted` is the resources required for obtaining `clear` data. `sealed` resources require access to a TPM [Trusted Computing Group 2007] or vTPM [Berger et al. 2006] and a key for unsealing. `encrypted` resources require a key and may be decrypted locally or using an external decryption resource.

Domains change state by attempting to acquire needed resources or composing existing resources. If an object is `needed`, the domain will attempt to obtain it by sending requests to other domains it believes may provide it. Resources acquired in this manner will change state to `clear`, `encrypted` or `sealed` based on how the remote domain delivers the resource. If the object obtained is `clear`, then the acquisition process ends. If an object is `sealed` or `encrypted` and the domain acquires keys and/or services to unseal or decrypt it, the domain's state changes to show the object as `clear`. If a domain is able to acquire the key for an `encrypted` or `sealed` object and a resource for decryption or unsealing, the object's status is changed to `clear`. Note that a domain will not seek keys or cryptographic services unless they are specified as `needed` in its state.

3.3. MAC Properties

Several properties must hold for all correct MAC_h and $\mathsf{MAC}_{T_s(\sigma_s)}$ relations. We assume that both MAC_h and $\mathsf{MAC}_{T_s(\sigma_s)}$ are total. Specifically, MAC_h specifies a single access control value for every T_d and T_s pair. Similarly, $\mathsf{MAC}_{T_s(\sigma_s)}$ specifies a set of allowed access types for each $T_s(\sigma_s)$ and T_r pair. When an access control decision is requested, there will always be one and only one result.

Because the domain and range of MAC_h are equivalent, we may be able to say more by examining reflexivity, symmetry, and transitivity properties. First, MAC_h is reflexive. This is not a strong property as it asserts domains have access to their own resources. Note that if an object is in an encrypted or sealed form, the object is treated as available only in an encrypted or sealed form. The associated clear text object becomes available following decryption or unsealing regardless of access control decisions. The reflexive property does not imply that domains send requests to themselves for their own resources. It simply captures the mathematical property that domains always have access to their own resources.

Unfortunately, we can say much less about transitivity of MAC_h. Transitivity holds among domains A, B, and C if when A is allowed to send requests to B and B to send requests to C, then A is allowed to send requests to C. Formally:

$$\mathsf{MAC}_h(T_A, T_B) \wedge \mathsf{MAC}_h(T_B, T_C) \Rightarrow \mathsf{MAC}_h(T_A, T_C)$$

There are clearly circumstances where A should not communicate with C because B does – separation between A and C being one example. Thus, MAC_h cannot be assumed transitive. Neither is MAC_h antitransitive as there are cases where legitimate communication should occur between A and C. Thus, any property related to transitivity must be explicitly specified for each analysis.

Symmetry is similarly difficult to assert uniformly across all MAC_h instances. Symmetry among domains A and B holds if when A is allowed to send requests to B, then B is allowed to send requests to A. Formally:

$$\mathsf{MAC}_h(T_A, T_B) \Leftrightarrow \mathsf{MAC}_h(T_B, T_A)$$

Like transitivity, there are circumstances where symmetry simply cannot hold, but neither can we assert asymmetry or antisymmetry must hold. Thus, no assumption is made about symmetry of MAC_h. Any property related to symmetry must be explicitly specified for each analysis.

Mathematical analysis of MAC_h and $\mathsf{MAC}_{T_s(\sigma_s)}$ provides little in the way of desired global properties that can be checked before analysis. However, it does provide guidance for defining theorems to be verified about specific MAC_h and $\mathsf{MAC}_{T_s(\sigma_s)}$. Specifically, antisymmetry and antitransitivity with respect to specific resources and resource combinations prove to be important guides for setting specific analysis goals.

4. MODELING FRAMEWORK

We analyze our model using model checking techniques, specifically the SAL [Bensalem et al. 2000] model checking tool set. The base model defines a collection of domains communicating via a common hypervisor that is faithful to the formal model. Specifically: (i) the hypervisor accepts requests for resources and delivers those requests as allowed by Hypervisor MAC policy; (ii) each domain honors requests received as allowed by Local MAC policy; and (iii) each domain attempts to acquire resources it requires and changes operational state when those resources are acquired. The stateful nature of domains and MAC policy mandates explicit modeling of state, motivating our use of model checking techniques.

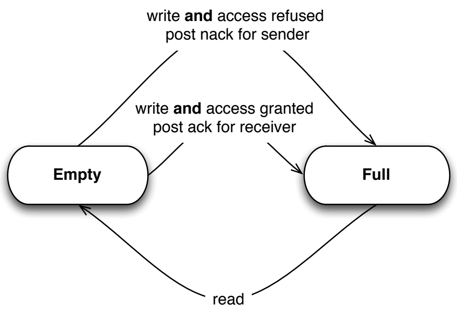
4.1. Hypervisor Model

The hypervisor model used to exchange messages among domains is shown abstractly in figure 2(a) and concretely in figure 3. The model is parameterized over `access?`, a relation that defines Platform MAC policy ($\mathsf{MAC}_h(T_d, T_s)$). Specifically, `access?` maps pairs of domains to a Boolean value indicating whether communication is allowed. Domains wishing to communicate do so by assigning a message to `dataIn` and setting `act` to `send` when the hypervisor is in an empty state.

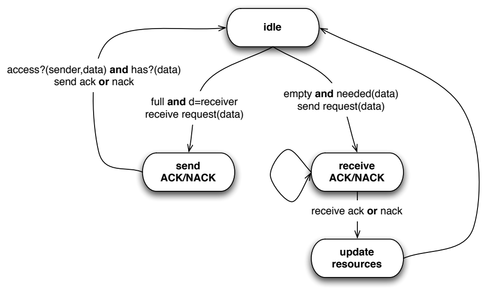
If the `access?` relation allows communication, the message is moved from `dataIn` to a communications buffer to await receipt by the receiving domain. If the `access?` relation does not allow communication, a `nack` message is placed in the communications buffer indicating communication failure. The hypervisor state is then set to full allowing the message recipient to retrieve the stored message.

When the hypervisor is in a full state, the receiving domain copies the communication buffer contents and sets `act` to `read`. The receiving domain then sends an `ack` message back to the sender. Because Platform MAC is not symmetric, acknowledgement messages are checked by inverting the `access?` relation.

Our rationale for modeling message passing explicitly is the stateful, distributed nature of access control. Our experience analyzing other distributed systems [Frey



(a) Hypervisor model transition diagram.



(b) Domain model transition diagram.

Fig. 2. State transition diagrams for system models.

```
hypervisorD [access? : [[DOMAIN,DOMAIN] -> BOOLEAN]] : MODULE =
BEGIN
  INPUT dataIn: MESSAGE, act: ACTION

  OUTPUT hypervisorState : HYPERVISORSTATE, buffer: MESSAGE

  INITIALIZATION
    hypervisorState = hypervisorEmpty;
    buffer = dataIn;

  TRANSITION
  [
    act = send AND hypervisorState = hypervisorEmpty -->
       buffer' = IF (access?(dataIn.sender,dataIn.receiver)
                    AND NOT ack?(dataIn.payload))
                   OR
                   (access?(dataIn.receiver,dataIn.sender)
                    AND ack?(dataIn.payload))
                THEN dataIn
                ELSE (# payload := nackVal,
                        sender := dataIn.receiver,
                        receiver := dataIn.sender #)
                ENDIF;
       hypervisorState' = hypervisorFull;
    []
    act = receive AND hypervisorState = hypervisorFull -->
      hypervisorState' = hypervisorEmpty;
    []
    ELSE -->
  ]
END;
```

Fig. 3. Complete model for hypervisor message exchange model.

et al. 2002; Kong and Alexander 2000; Dieckman et al. 1998] suggests that without central coordination among domains and the hypervisor, it is not possible for us to dismiss request ordering out-of-hand and the role of state change out-of-hand. The devil is always in the details.

4.2. Domain Model

The model used to specify communicating domains is shown abstractly in figure 2(b) and concretely in figures 4 and 5. The model is parameterized over `d`, the domain's identity; `localAccess?`, a relation that defines the Local MAC policy ($\mathsf{MAC}_{T_s(\sigma_s)}$); `init`, a relation that initializes resources; and `updateState`, a relation that defines how a domain changes state. Domains wishing to communicate do so by assigning a message to `dataIn` and setting `act` to `send` when the hypervisor is in an empty state. When composed with the hypervisor model, the `dataIn`, `act`, and `send` variables are shared, facilitating communication.

A domain's state, σ, is defined by the `has?` array initialized by the `init` relation. The `has?` array specifies the status of each object known to the system as one of `clear`, `encrypted`, `sealed`, `needed`, or `notNeeded` as described in the formal model. The first three values specify the encryption status of a held object. The final two values specify whether the domain will attempt to acquire an object it does not have.

`localAccess` is a relation over domain IDs and resources indicating when a request for a resource will be honored. When a request for a resource is received from a domain, evaluating `localAccess` indicates whether the request should be honored. It is assumed that a domain always has access to its own resources, thus `localAccess` is ignored when a domain accesses resources locally.

As a domain model changes state, it sends messages to acquire `needed` resources and updates the local state of the object based on its success in doing so. How a domain responds to data acquisition attempts is specified by `updateState`, a function from domain state to domain state. `updateState` defines what new data can be inferred from known data. When a new object is acquired, the `has?` array is updated directly with the acquired object's encryption status. `updateState` is then run to update the system state given newly acquired data. `updateState` runs whenever state may have changed—even due to a previous execution of `updateState`.

The `updateState` function works by examining every specified data status combination it is aware of. Effectively, it implements rules for inferring new state values as a collection of simple conditionals. While requests to other domains gather resources, `updateState` specifies what to do with them. Its only argument is a domain's `has?` array as the `has?` array captures the entire mutable state of a domain's data.

For example, if a domain needs a data blob D that is encrypted with key k, it will attempt to acquire both D and k by sending requests to other domains. As each is acquired, the domain's state is directly updated to reflect their individual status. `updateState` is run each time a new resource is acquired. When k's status is `clear` and D's status is `encrypted`, the `updateState` function will change the status of D from `encrypted` to `clear` if the domain possesses or has access to a decryption capability. Similarly, data that is in a `sealed` state is unsealed if its associated key is present and its domain has access to TPM or vTPM services.

The model shown graphically in figure 2(b) is encoded by SAL transitions shown in figure 5. The model state is initialized to an initial state, `driverInit`, where two transitions are possible. If the hypervisor model is in a full state and the buffered message is addressed to the domain, then the model processes the message. If the hypervisor model is in an empty state and the domain needs an object, then the model sends a message requesting the object. If there is no message for the domain and no object is needed, the domain idles.

The first transition defines the case where a message is in the hypervisor buffer, is addressed to the domain processing the message, and is not an `ack`. If this condition holds, the domain's object state, `has?`, is updated, and the hypervisor's `act` input is set to `receive` to allow further message processing. An acknowledgement is prepared by

```
domain [d:DOMAIN, localAccess?:[[DOMAIN,RESOURCES] -> BOOLEAN],
init:[RESOURCES -> DATASTATUS],
updateState: [DATASTATUSARRAY -> DATASTATUSARRAY]] : MODULE =
BEGIN
  LOCAL driverState : DRIVERSTATE
  LOCAL has? : DATASTATUSARRAY
  LOCAL requested : PAYLOAD
  GLOBAL dataIn: MESSAGE, act: ACTION
  INPUT buffer: MESSAGE, hypervisorState: HYPERVISORSTATE
INITIALIZATION
  driverState = driverInit;
  has? IN p:DATASTATUSARRAY |
             FORALL (i:RESOURCES) : p[i] = init(i);
  dataIn = (# payload:=Z,sender:=domBuilder,receiver:=vtpmManager #);
  act = idle;
  requested=Z;
```

Fig. 4. Signature and initialization sections from the generic model instantiated to construct communicating domains.

observing the local MAC specification to determine if the message should be acknowledged positively or negatively. The domain model moves to a state where it sends the acknowledgement message and then returns to its initial state to await further messages.

The second transition defines the case where the domain needs access to an object it does not have. A message is prepared requesting the needed object and is addressed to a domain identified by the `resourceLocator` relation. This relation indicates where a given object can be obtained and is included primarily to control state-space size. Initial models randomly selected domains to request resources from and quickly became intractable. The requesting message is stored in the hypervisor input and processed accordingly by setting `act` to `send`.

After sending the request, the domain enters a state where it waits for acknowledgement. If a positive acknowledgement is received, the domain's object array is updated accordingly. If a negative acknowledgement is received, the object array is held invariant over the state change. In both cases, the domain returns to its initial state and restarts its message passing process.

4.3. Adversary Model

The adversary considered is a domain that participates in the system as: (i) a misconfigured, but otherwise benevolent actor; (ii) a hostile actor outside the system; or (iii) a hostile actor masquerading as a legitimate domain. We follow principles established by Dolev and Yao [1983] in that keys and hash values cannot be guessed by any agent and must be obtained directly or by observing communication.

Each time a domain obtains a new resource, it attempts to change state by deriving new information based on that resource. Given a key and data encrypted with that key, an agent also possesses the unencrypted data. Similarly for sealed data, wrapped keys, and hashes. A misconfigured or hostile domain accesses information it should not by requesting it directly or requesting the pieces needed to create it. The same operations used by correctly configured domains to access data are used by misconfigured or hostile domains.

The Platform MAC policy (MAC_h) prevents communication to or from a domain of which it is not aware. If a domain is not specified in MAC_h, the hypervisor will not allow communication with that domain. Thus, modeling a hostile outside domain is useful only in conjunction with bad MAC_h configurations.

```
TRANSITION
[
  driverState = driverInit AND hypervisorState = hypervisorFull
    AND d = buffer.receiver AND NOT ack?(buffer.payload) -->
    has?' = updateState(has?);
    act' = receive;
    dataIn' = (# payload:=
                  (IF localAccess?(buffer.sender,buffer.payload)
                   THEN IF has?[buffer.payload]=clear THEN ackVal
                        ELSIF has?[buffer.payload]=encrypted THEN ackEnc
                        ELSIF has?[buffer.payload]=sealed THEN ackSealed
                        ELSE nackVal
                        ENDIF
                   ELSE nackVal
                   ENDIF),
                 sender:=d,
                 receiver:=buffer.sender #);
    driverState' = driverTest2;
  []
  (EXISTS (r:RESOURCES) : has?[r]=needed)
                          AND driverState = driverInit
                          AND hypervisorState = hypervisorEmpty -->
    has?' = updateState(has?);
    act' = send;
    dataIn' IN m:MESSAGE | has?'[m.payload]=needed
                            AND m.payload/=ackVal AND m.payload/=nackVal
                            AND m.payload/=ackEnc AND m.payload/=ackSealed
                            AND m.sender=d AND m.receiver/=d
                            AND m.receiver=resourceLocator(d)(m.payload)
        ;
    driverState' = driverTest1;
    requested' = dataIn'.payload;
  []
  driverState = driverTest2 AND hypervisorState = hypervisorEmpty
    AND d = buffer.receiver -->
    act' = send;
    driverState' = driverInit;
  []
  driverState = driverTest1 AND hypervisorState = hypervisorFull
    AND d = buffer.receiver -->
    act' = receive;
    has?'[requested] = IF buffer.payload = ackVal THEN clear
                       ELSIF buffer.payload = ackEnc THEN encrypted
                       ELSIF buffer.payload = ackSealed THEN sealed
                       ELSE has?[requested]
                       ENDIF;
    driverState' = driverInit;
  []
  ELSE -->
]
END;
```

Fig. 5. Transition section from the generic model instantiated to construct communicating domains.

```
svp: MODULE =
  (([] (id:PARTICIPANTS) :
      domain[id,localMAC[id],initDomain[id],updateStateDomain[id]])
      || hypervisorD[platformMAC]);
```

Fig. 6. System model constructed from a hypervisor and several instantiated generic domain models.

Benevolent, misconfigured domains and hostile insiders are modeled by defining a domain requesting resources it should not obtain. The simplest way to do this is simply define a domain that requests all resources. The trade-off is that including such domains quickly causes state space explosion. As will be seen later, we are able to model one or two such domains before encountering severe run time penalties.

4.4. Systems Models

Figure 6 is an example system model that instantiates a collection of domain models with a single hypervisor model. SAL constructs an array of domain models using identifiers from the `PARTICIPANTS` type and composes those models asynchronously. It then instantiates a single hypervisor model and composes that with the domain array.

In SAL, the notation `([] id:PARTICIPANTS)` is a universal quantifier over the type `PARTICIPANTS` where the asynchronous composition operator `[]` is used rather than conjunction. What the specification in figure 6 defines is the asynchronous composition of one instance of `domain` for each member of type `PARTICIPANTS` instantiated with initialization, transition, and access control functions. Specifically, `localMAC[id]` is the local MAC policy, `initDomain[id]` is the local resource initialization function, and `updateStateDomain[id]` is the state transformation function for domain `id` while `platformMAC` is the hypervisor model's platform MAC policy.

A significant benefit of this approach is the ease with which new system models are defined. To add a new domain, one simply adds a new value to the `PARTICIPANTS` type and updates the actual values instantiating `localAccess?`, `init`, and `updateState` relations to model the new domain. These additions occur orthogonally to domains that are previously defined. Additionally, the `access?` relation in the hypervisor must be updated to allow communication with the newly added domain. Updating a domain is similar where each function is modified rather than new elements added to `PARTICIPANTS`. This feature is critical for the consumers of our analysis results who are continuously updating the analyzed design.

4.5. Theorems

Correctness conditions are classified as soundness and completeness conditions represented as safety and liveness properties respectively. Soundness conditions represent properties that must be enforced by the access control policy. Such conditions include restricting access to keys, domain's local data, and services provided by domains. Completeness conditions represent properties that must be present in the system for correct function. Such conditions include domains obtaining data necessary for their function and that the system can reach a booted state in the presence of the access control policy.

Soundness conditions are security properties specific to the goals of each access control policy. They specify that domains do not have access to resources they should not. Such properties represent classical LTL safety properties of the form $G(\neg p)$ – "globally not p" – where p specifies a condition that must not hold. Such properties include protection of data and keys; operation ordering; and integrity of hashes.

A concrete example of a soundness property is that '*no domain other than the vTPM Manager should have access to vTPM Manager data.*' vTPM data is initially encrypted and should only be decrypted by its associated vTPM and never shared. The corresponding LTL invariant is:

$$\forall d : PARTICIPANTS \cdot G(d \neq \texttt{vtpmManager} \Rightarrow \sigma(\texttt{d}, \texttt{vtpmManData}) \neq \texttt{clear})$$

and its SAL representation is:

```
safeVtpmManData: THEOREM svp |- (FORALL (d:PARTICIPANTS) :
     G(d/=vtpmManager => NOT has?[d][vtpmManData]=clear));
```

By extension, satisfying the theorem ensures the vTPM Manager does not redistribute its data following decryption and no domain holds the encrypted vTPM Manager data and its associated key. Because keys and the data they encrypt may always transform into clear data, if any domain holds encrypted vTPM Manager data and its key, it will hold vTPM Manager data in the next state. In effect, the theorem ensures the confidentiality of the vTPM Manager's data in domains other than the vTPM Manager.

The theorem `safeVtpmManData` and all such correctness conditions take the form of an invariant or safety property over the subset of reachable system states. The antecedent in the global condition `d/=vtpmManager` causes the invariant to be vacuously true for the domain named `vtpmManager`. All other domains must satisfy the consequent of the condition `NOT has?[d][vtpmManData]=clear` specifying that they do not hold `vtpmManData` in the clear.

Completeness conditions define function correctness properties that must hold for all access control policy instances. They specify that all domains reach states that support normal operation. Such properties represent classical LTL liveness properties of the form $G(F(p))$—"globally, eventually p"—where p specifies access to necessary resources. Such properties include acquisition of needed resources; boot progress; and presence of measurements. Completeness conditions ensure proper system function is maintained in the presence of any access control policy.

A concrete example of a completeness property important in our model is that *the vTPM always, eventually has access to vTPM data.* vTPM data is initially encrypted and must be retrieved by the vTPM along with its key. The corresponding LTL theorem is:

$$G(F(\sigma(\texttt{vtpmManager}, \texttt{vtmpManData}) = \texttt{clear}))$$

and its SAL representation is:

```
vtpmManagerStarts: THEOREM
    svp |- G(F(has?[vtpmManager][vtpmManData]=clear));
```

Unfortunately, stating the theorem positively—along all paths the vTPM acquires its data—results in a theorem that is computationally complex. In even simple models, such theorems are frequently unprovable due to their computational complexity. For this reason we approximate the liveness condition proof by searching for the negation of the desired boot result. The negation of the LTL theorem is:

$$F(G(\sigma(\texttt{vtpmManager}, \texttt{vtmpManData}) \neq \texttt{clear}))$$

asserting that *the vTPM eventually, always does not have access to vTPM data*. Proving this theorem shows that the vTPM never obtains its data, while disproving it provides a witness to successful boot with respect to the vTPM in the form of a counterexample.

We are ultimately interested in finding witnesses that can be further examined and not successful proof of the theorem. All completeness theorems are stated in the same manner by specifying the presence or absence of resources and services.

The new property is a safety property. We make one additional approximation dropping the eventually quantifier resulting in the LTL theorem:

$$G(\sigma(\mathtt{vtpmManager}, \mathtt{vtpmManData}) \neq \mathtt{clear})$$

For the SAL property describing vTPM data above, the following safety property is checked:

```
notVtpmManagerStarts: THEOREM
    svp |- G(has?[vtpmManager][vtpmManData]/=clear);
```

As noted, `notVtpmManagerStarts` takes the form of an invariant or safety property that is far easier to check. No state sequences need be explored implying each state is visited only once and the model checker halts when a single counterexample is found.

The approximation can also be understood informally by looking at the structure of boot. All successful boot sequences in the boot model enter a stuttering state exhibiting desired properties of correct system boot. Technically, a successful boot sequence does not terminate, but settles into a state that transitions only to itself. Thus the term “stuttering”. If we can show this state is never entered, we show that boot is never successful and our access control policy is in some way too strong. Conversely, if we can show that state is entered, we show the system *potentially* boots.

The efficiency gained by approximating the liveness property with the failure of a safety property is not without cost. First, the approximation is clearly not sound. SAL will terminate as soon as the first counterexample is found, not all counterexamples as implied by negating the original theorem. Remember that we are not verifying boot, but verifying that boot can happen in the presence of the access control policy. Thus, finding a case when boot occurs is sufficient for our purposes.

Second, if the counterexample found is a degenerate case, it represents a false-positive. Each counterexample must be checked manually using the SAL simulator to determine if it represents the stuttering state at the end of a state sequence consistent with the original liveness theorem. The trace is examined to ensure that intermediate states are entered and the resulting state is in fact the desired boot result. This is not difficult, but must be done for each discovered counterexample to help ensure validity of the boot path. Note that we encounter degenerate cases primarily due to errors in the boot model during development and not due to bad access control policies.

4.6. Constructing Analysis Models

An access control policy is analyzed by defining and instantiating hypervisor and domain models with specific policies and resources and analyzing them with respect to baseline theorems plus any platform specific theorems. As noted earlier, the Platform and local MAC policies are specified by the `access?` and `localAccess?` functions respectively. Resource initialization is specified by the `init` relation that provides initial values for the internal `has?` function. Additionally, the `resourceLocator` function may be updated to reflect misbehaving domains.

Figures 7, 8, and 9 show representative local access control, platform access control, and system state representations. Each is accompanied by its logical equivalent to provide some intuition for the simplicity of moving from logical formalism to the analysis tool. To emphasize, these are representative examples only and can easily be modified or replaced.

```
nothingButVtpmManData(d:DOMAIN,data:RESOURCES): BOOLEAN =
  (d=vtpmManager AND data=vtpmManData);
```

Fig. 7. Local access control policy fragment allowing the vTPM Manger to request its data.

```
platformMAC(s,r: DOMAIN): BOOLEAN =
  (IF r=tpm THEN (s=vtpmManager OR s=domBuilder)
   ELSIF r=domBuilder THEN (s=tpm OR s=controller)
   ELSIF r=vtpmManager THEN (s=tpm OR s=vtpm)
   ELSIF r=controller THEN (s=domBuilder OR s=store OR s=vtpm OR s=measurer OR s=attestation)
   ELSIF r=store THEN (s=domBuilder OR s=controller OR s=vtpm)
   ELSIF r=vtpm THEN (s=vtpmManager OR s=store OR s=measurer OR s=controller)
   ELSIF r=measurer THEN (s=controller OR s=attestation)
   ELSIF r=attestation  THEN (s=controller OR s=measurer)
   ELSIF r=nameServer THEN TRUE
   ELSE TRUE
   ENDIF);
```

Fig. 8. Example Platform MAC policy implemented in the hypervisor.

Figure 7 is a simple policy that allows the vTPM Manager to request its data. It implements the semantics of $\mathsf{MAC}_{s(\sigma_s)}$ defined as:

$$\forall \sigma_s \cdot \mathsf{MAC}_{s(\sigma_s)}(d,r) \equiv (d = \texttt{vtpmManData}) \wedge (r = \texttt{vtpmManager})$$

$\mathsf{MAC}_{s(\sigma_s)}$ defines the only condition when a request for `vtpmManData` will be honored and is state invariant. Specifically, the snippet is true when the requesting domain is `vtpmManager` and the requested resource is `vtpmManData` in any system state.

The following definition of MAC_h provides the semantics of the hypervisor access control policy, implemented as `platformMAC` in Figure 8, governing communication between domains. Recall that MAC_h defines a relation between domains indicating allowed communication. Both the semantics and implementation are relatively simple relations making them both simple to write and simple to verify.

$$\begin{aligned}
\mathsf{MAC}_h(r,s) \equiv\ & r = \texttt{tpm} \wedge s \in \{\texttt{vtpmManager}, \texttt{domBuilder}\} \vee \\
& r = \texttt{domBuilder} \wedge s \in \{\texttt{tpm}, \texttt{controller}\} \vee \\
& r = \texttt{vtpmManager} \wedge s \in \{\texttt{tpm}, \texttt{vtpm}\} \vee \\
& r = \texttt{controller} \wedge s \in \{\texttt{domBuilder}, \texttt{store}, \texttt{vtpm}, \texttt{measurer}, \texttt{attestation}\} \vee \\
& r = \texttt{store} \wedge s \in \{\texttt{domBuilder}, \texttt{controller}, \texttt{vtpm}\} \vee \\
& r = \texttt{vtpm} \wedge s \in \{\texttt{vtpmManager}, \texttt{store}, \texttt{measurer}, \texttt{controller}\} \vee \\
& r = \texttt{measurer} \wedge s \in \{\texttt{controller}, \texttt{attestation}\} \vee \\
& r = \texttt{attestation} \wedge s \in \{\texttt{controller}, \texttt{measurer}\} \vee \\
& r = \texttt{nameServer}
\end{aligned}$$

Figure 9 represents two snippets from the larger function initializing the `has?` structure representing domain state. This function is called to initialize state when a domain model is started and is included as an example of how a system state is represented. The first block represents initialization of the TPM state indicating that it has access to several data elements (*e.g.* `k2Val` and `hashK1`) and services (*e.g.* `crypto` and `extendPCR`) in the `clear` state that it may provide to other domains.

```
initLocalSvpLine : [DOMAIN -> [RESOURCES -> DATASTATUS]] =
  (LAMBDA (d:DOMAIN) :
    (IF d=tpm THEN (LAMBDA (v:RESOURCES) :
        %% TPM starts with services and NVRAM contents.  Technically
        %% k2 is not in the TPM, but the TPM will eventually provide it
        IF v=k2Val OR v=crypto OR v=extendPCR OR v=hashK1 OR v=hashWK2
        THEN clear
        %% Everything has access to resources in memory
        ELSIF v=k1Val OR v=schema THEN clear
        ELSIF v=k2Val THEN sealed
        ELSE notNeeded
        ENDIF)
     ...
     ELSIF d=domBuilder THEN (LAMBDA (v:RESOURCES) :
        %% K1 is in memory, crypto services are needed and hashk1
        %% is needed to verify k1
        IF v=hashK1 OR v=crypto THEN needed
        %% Everything has access to resources in memory
        ELSIF v=k1Val OR v=schema THEN clear
        ELSIF v=k2Val THEN sealed
        ELSE notNeeded
        ENDIF)
     ELSIF
     ...
     ENDIF)
```

Fig. 9. Example domain state representation. Note the use of nested lambdas taking advantage of SAL's curried functions.

The second block represents the Domain Builder indicating that it has data and services, but also that it needs `hashK1` and a `crypto` capability. The TPM is always considered running as it has all its required data and services in the clear, the Domain Builder will not be considered running until the state of `hashK1` and `crypto` change to `clear`. The Domain Builder model will seek to acquire these necessary resources from domains known to provide them.

In both blocks, the default for any resource is `notNeeded` indicating that a domain does not have or need the resource and will not try to seek it. In models with bad actors, domain state will indicate that resources are needed that in actuality are not causing the domain to misbehave and try to acquire those resources. Furthermore, several resources are available that are not actually within the domain, but are available in system memory. There is no distinction between such resources and those available in the domain itself.

Each model investigated is constructed by instantiating the hypervisor model's `access?` parameter with a specific platform MAC policy and instantiating each domain's `localAccess?`, `init`, and `updateState` parameters with local MAC, initial resource state, and state transition functions respectively. The ability to swap in new functions to consider other models is a key contribution of this work and is critical to supporting system designers in real-time.

5. MODEL CHECKING RESULTS

Analysis is performed by customizing the baseline system model and checking soundness and completeness properties. Theorems remain invariant over all analysis activities as they represent security properties that must hold. Some analyses include additional theorems, but theorems described previously are checked for all models. System models define an access control policy and the set of domains it governs. Such sets of domains range from normative systems to those representing various threats in the form of rogue domains. The intent is to show the access control model allows normal system boot while preventing security problems in badly configured systems or systems with bad parts.

5.1. Analysis Models Evaluated

Specific models examined for each access control policy set investigate: (i) boot failure; (ii) operational failures; and (iii) rogue domains with respect to each access control model and are listed in table II. Boot failure models determine if a system can boot. Operational failure models determine if policy failure at run-time exposes secrets to unauthorized domains. Finally, rogue domain models examine the impacts of a domain within the system or a domain external to the system actively attempting to acquire unauthorized resources.

A boot failure model examines both successful and failed boot by examining a minimally constrained system with respect to completeness and soundness theorems. The system will be allowed to attempt boot in every way possible, whether it leads to a booted state or not. Successful completeness theorems for individual domains and the complete system indicate that there is a successful boot sequence under the specified access control policy. Recall that a successful completeness theorem actually results in a counterexample showing the successful boot sequence. Counterexamples are used directly to check boot, but are also useful for debugging and sanity checking.

Verified soundness theorems indicate that in both successful and failed boot sequences secrets are protected. Due to the distributed nature of domains, a single domain entering an unexpected state and failing to boot properly will not prevent other domains around it from booting. Domains requiring access to the failed domain may also enter unexpected states and fail to boot. Analyzing boot failure models with respect to soundness theorems determines that all reachable system states are safe with respect to critical resources regardless of final boot correctness.

Operational failure occurs when some aspect of the access control policy allows object access that should be disallowed. Although we assume the implementation of the access control enforcement mechanism is correct, it is possible for policies to be incorrect. Operational failure models specifically determine that an access control policy allows system boot to a known good state and keeps the system in a safe state during normal operation.

Rogue attacks occur when a domain inside or outside the system attempts to gain access to resources it should not have access to. While operational failures are a result of implementation errors, rogue attacks are intentional. We model a rogue domain by initializing its state to cause it to need resources it should not have access to. Then, use the same correctness conditions verified for operational failure to check to see if unauthorized resources are obtained.

5.2. Baseline Theorems

Table I lists a collection of baseline theorems verified for all system models. Each entry lists a theorem's name, type and basic semantics of the theorem. The details of individual theorems are specific to the domain. To understand the approach, one need only understand we are checking properties that must be enforced by access control and that must be present in the system. Note also that the theorem named `svp` does not represent a theorem expressed in LTL, but a deadlock check over the entire model that is necessary for the validity of other theorems.

Two pairs of soundness theorems check to ensure data is held confidentially and that resources are not available before they should be. Specifically, `safeVtpmManData` is the theorem discussed earlier while `noVtpmBeforeMan` is a new theorem that checks to ensure that vTPM services are not provided before the vTPM Manager is available.

Table I. Example collection of core theorems verifying domain boot and protection of services and data.

Theorem Name	*Type*	*Property Checked*
`safeVtpmManData`	Soundness	Confidentiality of vTPM Manager data
`safeVtpmData`	Soundness	Confidentiality of vTPM data
`noVtpmBeforeMan`	Soundness	No vTPM running before the manager
`noVtpmSvcBeforeData`	Soundness	No vTPM services before data acquired
`noVtpmWithoutMan`	Soundness	No vTPM running without the manager
`notDomBuilderStarts`	Completeness	Domain Builder starts
`notVtpmManStarts`	Completeness	vTPM Manager starts
`notVtpmStarts`	Completeness	vTPM starts
`notControllerStarts`	Completeness	Controller starts
`notStoreStarts`	Completeness	Host Storage starts
`notMeasurerStarts`	Completeness	Measurer starts
`notNameServerStarts`	Completeness	Name Server Starts
`svp`	Deadlock check	No deadlocks

Note: Type indicates whether it is a soundness or completeness theorem and *Property Checked* indicates property semantics. `svp` refers to calling the deadlock checker on the `svp` model and does not name an actual theorem.

Table II. Models evaluated against theorems from table I.

System	*# of Domains*	*Boot*	*Single Rogue*	*Double Rogue*
SINIT Line - Measured by SINIT	5	855	Complete	Complete
SVP Line - Supervisor VP up	7	1,238	Complete	Partial
UVP Line - Supervisor VP plus drivers	15	7,625	Complete	Partial
UVP - Supervisor VP through user VP	19	30,889	Partial	None

Note: Numeric *Boot* value indicates worst case run-time for theorems checked in seconds. Complete, Partial, and None indicate completeness of the analysis. Each domain is aware of 13 resources that it may possess or attempt to acquire.

Similar theorems are checked for an individual vTPM instance. All completeness theorems check to ensure that critical domains can boot into good states and have the form of `notVtpmManagerStarts` previously.

Theorems listed in table I form the core collection of verification conditions that all models are evaluated against. Additional theorems are added for specific cases and to extend correctness conditions to other aspects of the model. A typical model requires verification of roughly 40 theorems.

6. EVALUATION

The SAL [Bensalem et al. 2000] BDD-based finite state model checker, deadlock checker, and bounded state model checker were used to check theorems over multiple access control policies in the context of normal operation, boot failure and rogue attacks. As SAL provides only a well-formedness checker and not a complete type checker, the PVS [Owre et al. 1992] prover was used to verify type safety of all specifications. The SAL BDD-based simulator was used extensively during debugging and checking counterexamples.

6.1. Complexity Issues

In our initial naive verification studies we defined a collection of domains that aggressively attempted to acquire all resources from all other domains. Specifically, each domain examined its state and non-deterministically requested an object it did not have from another, arbitrary domain. Thus, every analysis considered all possible requests by all possible domains in all possible orders. This approach established the total correctness of access control policies. However, state space complexity became overwhelming after including only five domains with eight total resources.

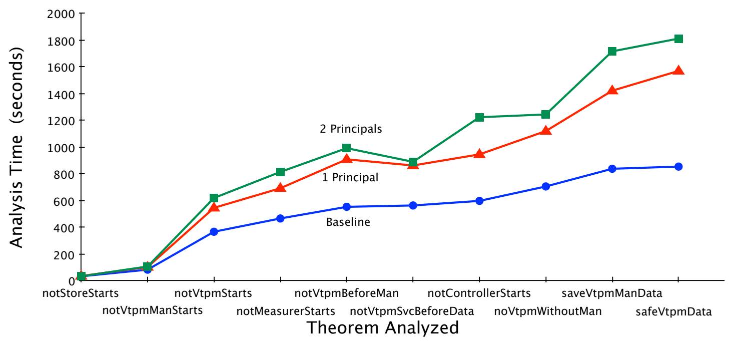


Fig. 10. Impact of allowing domains to request needed resources from arbitrary sources. Circles indicate a baseline with no principals making arbitrary requests. Triangles and squares indicate time complexity of one and two principals making arbitrary requests respectively.

Our first approximation limits the domains each resource can be requested from. The system analyzed uses a name server indicating where domains may find resources. In effect, this approximation implements that name server function. Figure 10 shows the impact of including the name service capability. Five domains are configured to request resources from only a limited set of sources providing a baseline for comparison. Then two additional experiments allowed one and two domains to make arbitrary requests respectively. As shown in Figure 10, allowing one domain to seek its resources from any other domain results in a significant complexity increase of roughly 150%. Allowing two domains to seek their resources from any other domain results in an additional complexity increase, but of lower magnitude.

Our second approximation allows domains to request only resources that they need. The same five principal model where all domains request only needed resources is run as a baseline and two additional experiments allow one then two domains to seek arbitrary resources whether needed or not. As shown in Figure 11, allowing one domain to seek all resources results in a substantial increase in complexity. Allowing two domains to see all resources results in a complexity increase of nearly two orders of magnitude.

The conclusion of these early studies is that a brute force analysis of access control policies allowing all domains to attempt access to all resources is not feasible. The two mechanisms discussed are implemented as the `resourceLocator` relation used to target requests and the `needed` and `notNeeded` object status values respectively. The `resourceLocator` relation limits where a domain will look for an object and corresponds to a name service in the modeled system. The previously discussed `needed` and `notNeeded` status values indicate whether a domain should seek an object. Thus, a domain will only attempt to access an object if it is `needed` and will send requests to domains specified by the `resourceLocator` relation. By limiting a domain to specific object requests from specific domains, the state space becomes quite manageable for systems involving nearly 20 communicating domains.

6.2. Model Checking Results

With approximations in place to control state space explosion, analysis of actual system models begins. We examined four systems that included 5, 7, 15, and 19 domains each with knowledge of 13 resources. The size of each system model corresponds with a configuration representing an important boot or run-time subsystem identified by system designers. Analysis of all theorems was performed with the SAL deadlock checker (`sal-deadlock-checker`) and the SAL symbolic model checker (`sal-smc`) with slicing

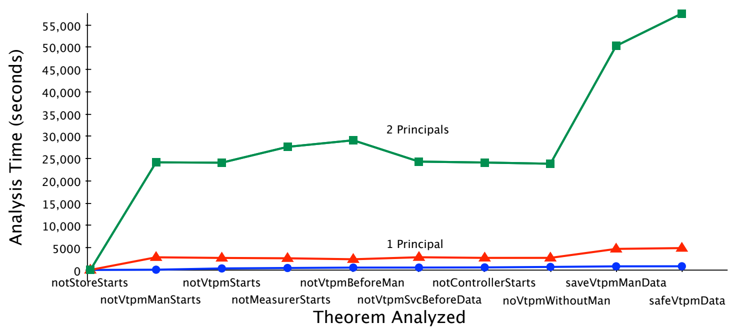


Fig. 11. Impact of allowing a single domain to attempt acquisition of all resources on analysis time. Circles indicate a baseline with no principals requesting all resources. Triangles and squares indicate run times of one and two principals requesting all resources.

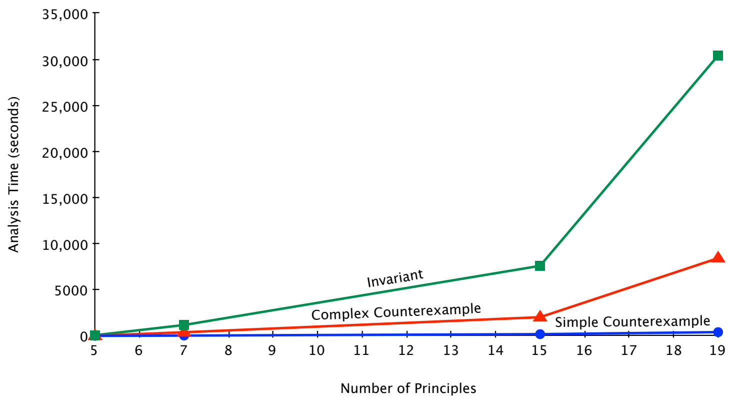


Fig. 12. Run times for models generating counterexamples of increasing complexity.

enabled. The execution platform for each theorem was a dual-core 2.7Ghz Linux workstation.

Figure 12 shows the run times of three typical theorems over models of size 5, 7, 15 and 19 domains. The three theorems represent finding a simple counterexample, a complex counterexample, and fully verifying an invariant. Both counterexamples are approximations of liveness conditions as discussed previously. The simple counterexample case occurs when the model checker finds a counterexample relatively early in the state space search. This specific example is checking to see if a domain obtains boot resources. For this simple case the domain obtains its resources after sending a single message. As expected, search times increase moderately from 10 to 194 CPU seconds.

The complex counterexample is similar to the simple counterexample, except that the domain being checked must obtain numerous resources that are not all immediately available early in boot. Again, a predicate asserting that it does obtain resources is negated and a counterexample is discovered. Note that for the complex counterexample, analysis times increase from 27 to 2034 CPU seconds as the number of principals increases from 7 to 15 and to almost 8500 CPU seconds when analyzing 19 principals.

Finally, the invariant checks a safety property that requires checking all system states. In this case the property is checking to determine that an encrypted object and its key are never held by the same domain other than the owner of the object. As to be

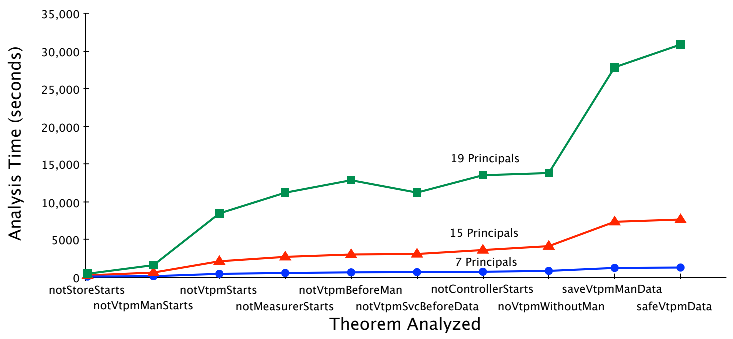


Fig. 13. Time required for verifying all theorems in systems with 7 principals (circles), 15 principals (triangles) and 19 principals (squares).

expected, the increase in resources from 78 to just over 30,000 seconds is significant. However not as significant as one might assume. The invariant takes approximately 3 times as long as the complex counterexample in the model with 5 nodes. The invariant takes approximately 3 times as long for 15 nodes as well—both increasing by two orders of magnitude. The sharp inflection at 15 as the number of principals increases is due to thrashing caused by increasing data structure size.

For completeness, figure 13 shows complexity results for all theorems that are checked in models of size 7, 15 and 19. These results again show complexity increasing with search depth as expected.

For each of our system models and access control policy sets, we performed 4 analyses with respect to baseline theorems from table I plus specialized theorems for each model. First we configured the system correctly and evaluated all allowed boot cases to determine the minimal correctness of the policies and establish our systems will reach booted states. We then repeated each analysis allowing each domain to individually seek all resources emulating the behavior of a rogue domain. Next we repeated the rogue domain analysis allowing bad domains to seek resources wherever they chose to. Finally, we repeated the analysis allowing pairs of domains to see all resources, emulating the behavior of pairs of rogue domains. For our 19 principal model, some models ran for as long as two days during the last analysis.

We were able to discover several minor errors in the access control policies tested. Interestingly, these issues were almost all discovered in the smaller models and not the larger models. We believe this is due to the nature of the system examined where core domains are the most complex in terms of their resource needs. Our larger models are consistently supersets of smaller models confirming an incremental approach to development and analysis taken over the course of the project.

The rationale behind analyzing models with multiple rogue domains is that two domains together can acquire resources and services that one alone cannot. As an example, consider a domain that can acquire a key and another that can acquire data encrypted with that key. If both are rogue domains, they can access unencrypted data together while it is impossible separately. Interestingly, our analysis revealed no such problems in the policies we analyzed as long as the hypervisor's policy remains sound limiting communication among principles. While we were able to simulate such problems by weakening the hypervisor policy, we did not see such problems in the presence of the good hypervisor policy. This was a surprising result suggesting that the two tiered approach modeled—hypervisor policy with domain policies—has significant benefit.

6.3. PVS Type Checking

One advantage to SAL is its implementation of a rich dependent type system in its specification language. Because the SAL type system is dependent and allows predicate subtypes—the ability to specify types as subsets of other types—the SAL type system is not decidable. Using predicate subtypes it is quite possible to define types that are empty and then declare a construct to be of that type. The result is an inconsistency that potentially invalidates an entire model.

Because the SAL well-formedness checker that performs rudimentary type checking does not have a theorem proving capability, it cannot check type judgments generated by its dependent type system. As a specific example, SAL cannot detect when a predicate subtype is uninhabited. Even if it did implement a theorem prover, the process cannot be fully automated. The SAL type system shares a semantics with PVS, also developed by SRI. The PVS type system includes dependent types implemented using predicate subtypes in the same manner as SAL. SAL users are thus encouraged to move type declarations into PVS for semi-automated checking.

To perform type checking, declaration sections from SAL models are manually transformed into a PVS theory with little modification—only array declarations require changes due to small, syntactic differences. PVS is then used to perform type checking. Any type check conditions (TCCs) that cannot be automatically discharged may then be verified with the PVS prover. In all our example cases, TCCs generated by PVS are discharged with the single, built-in PVS proof tactic.

7. RELATED WORK

Jaeger et al. [2003] state that access control policy analysis is a relatively new area of investigation. Since then, a number of researchers have developed techniques for specifying, analyzing and synthesizing access control policies. As Flask-style mandatory access control policy [Spencer et al. 1999] is used exclusively in this work, we focus on related work with similar goals.

Researchers at MITRE [Guttman et al. 2004] provide a semantics and formal system for checking SELinux configuration files using nuSMV [Cimatti et al. 2002]. By providing a denotation from Flask configuration primitives to logical representations, MITRE is able to directly analyze Flask policies. Although we utilize a similar semantics for understanding Flask policies, we abstract away the specifics of Flask implementation. It is far easier for an engineer to specify their access control goals abstractly than using the Flask language directly. The trade-off is the translation between Flask configuration files and our models must be verified when moving Platform MAC into a Flask form. Flask policies have not be implemented for the system we examine, further supporting the use of abstract policy descriptions.

Margrave [Nelson et al. 2010; Fisler et al. 2005; Dougherty et al. 2006] uses BDD analysis originally and subsequently Alloy to verify dynamic access control policies specified using XACML [Moses 2003]. In addition to automatically transforming policies into BDD representations, Margrave provides a unique capability for comparing two policies. A designer may design and maintain policies incrementally, understanding the immediate impacts of a design decision. XACML differs from the abstraction used here in that XACML is an access control specification language implying that Margrave analyzes actually policies rather than models. However, examination of Margrave's working examples reveals a similar approach ensuring both sufficient and complete conditions for policies. We do not provide an automated transformation from policies to SAL models, opting instead to provide a reusable, parameterized model for policy exploration.

Guttman et al. [2004], Margrave [Fisler et al. 2005; Dougherty et al. 2006], and our work all use a BDD-based model checker to verify policy properties. Some notable differences remain. While Margrave uses a custom translation to an underlying checker, Guttman et al. and our work use off-the-shelf model checking systems with their own high-level specification languages. Both Margrave and Guttman et al. work directly on XACML and Flask configuration files while our work focuses on higher-level abstractions. Given the abstraction differences between models analyzed in these systems, it is difficult to make significant comparisons among results.

Archer et al. [2003] describe using TAME [Archer 2000] to analyze SELinux security policies like those analyzed in this work. TAME is a customization of PVS [Owre et al. 1992] that checks properties of IO automaton [Lynch and Tuttle 1989]. Like MITRE's work, TAME analysis is performed at the abstraction level of the Flask configuration file. The TAME work differs substantially from our work as well as the MITRE and Margrave work in that it uses a proof checker (PVS) rather than a model checker for system analysis. By avoiding state-space explosion issues, TAME has the potential to explore larger policies than state-space exploration techniques. The trade-off is the lack of full automation during analysis. Although we use PVS in the work reported here, it is only used to verify static type properties of specifications.

We noted earlier our adaptation of semantics defined by Hicks et al. [2007] to define access control policies. Although we are not using Flask configuration files, policies we analyze must be implemented as Flask rules making this semantics appropriate for our use. Note that work by Hicks et al. [2007] is part of a larger effort verifying access control policies in virtualized systems similar to ours [Hicks et al. 2007; Rueda et al. 2009].

Jaeger et al. [2003] present an early example of Flask policy analysis targeting integrity protection. They developed a custom analysis tool, Goyko, that identifies conflicts between a collection of integrity goals and an SELinux policy. This work is noteworthy as it analyzes the example policy for SELinux, a significant real policy. Furthermore, their focus on integrity properties is similar to our exploration of soundness conditions, but does not capture completeness conditions.

Narain et al. [2008] present ConfigAssure, a general purpose tool for generating system configurations from formal specifications. In related work, they propose automatically synthesizing component configurations from first-order logic constraints using a combination of Alloy [Jackson 2011] and SAT solving techniques. Specifications for access control constraints along with sets of components to be configured are used to generate and solve a SAT problem. Similarly, Schaad and Moffett [2002] use Alloy for analyzing role-based access control configurations. While their target is role-based access control rather than mandatory access control, this distinction is minor. Specifically, Schaad and Moffett use Alloy to specify the RBAC96 model, defining separation properties and using Alloy to analyze for conflicts.

Zhang et al. [2005]; Guelev et al. [2004] have developed support for automatically verifying RW specifications using a model checking approach then automatically synthesizing XACML specifications from the same specifications [Zhang et al. 2004]. Using this work flow engineers write RW specifications defining read and write permissions on resources, verify their models, and translate them into XACML. This work flow is substantially similar to ours, except we use native SAL relations for representing MAC policies and do not automatically synthesize Flask configuration files. Given that our target is a system under design, it is not useful to synthesize Flask at this time.

Zanin and Mancini [2004] present a formal model, SELAC, for analyzing security policy configuration files specifically for SELinux. The SELAC model formally defines a model similar to that described earlier from Hicks et al. [2007]. They formally define specification constructs used in SELinux policy files and operations over those constructs to make accessibility determinations. They show how users define accessibility for an example system, but do not demonstrate automated analysis capabilities present in other systems described here.

8. CONCLUSIONS

We have described a scenario where formal analysis of distributed access control policies informed the design of a virtualized platform. Specifically, we defined a semantics for access control policies, defined policies for a specific experimental system, and examined their system-wide behavior to be implemented in Xen domains and the Xen hypervisor. To achieve this, we defined soundness and completeness properties describing confidentiality and run-time resource access requirements respectively. We used SAL to implement and analyze three classes of models representing boot failures, nominal operational behavior, and rogue attacks with respect to properties describing soundness and completeness properties. The results of our analysis supported system designers by providing evidence of trustworthiness claims used to inform design decisions. By developing a highly reconfigurable model and a modest computing cluster we were able to provide feedback in a timely manner as needed by design engineers. Both the approach and the specific models proved scalable and reusable across numerous design alternatives.

The models continue to be expanded to include more operational detail. Specifically, we are currently extending the model to include examination of locality in the vTPM implementation and access control implemented over system measurement functions. Both require additional access control policies and finer grained representation. While this work is ongoing, we can safely say the original models continue to be useful and are scaling to these new domains.

The MAC policies that we check are evolving models resulting from an ongoing design process. They are not MAC implementations and represent policies at a much higher abstraction level. Although we could model MAC policy implementations, even small policies would overwhelm SAL. Should we choose to address MAC policy implementation, the best approach would be synthesis of implementations from our high-level policies.

We have not attempted application to problems outside MAC analysis or using other access control mechanisms. An obvious next step would be extending the approach to discretionary access control where principals can delegate permissions. Because policies are specified as first-class in the model, it is possible to model delegation by changing policies during execution. Specifying when delegation occurs would require additional work, but could be done. Property-based access control models would prove more difficult due to a need to check parameterized properties. We feel this could be achieved, but not without significant work.

REFERENCES

M. Archer. TAME: Using PVS Strategies for Special Purpose Theorem Proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181, 2000.

M. Archer, E. Leonard, and M. Pradella. Modeling Security-Enhanced Linux Policy Specifications for Analysis. In *Proceedings of the DARPA Information Survivability Conferences and Exhibition (DISCEX'00)*, 2003.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pragg, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)*, Boldon Landing, NY, USA, 2003.

S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueb, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An Overview of SAL. In C. M. Holloway, editor, *Fifth NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

S. Berger, R. Caceres, K. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the Trusted Platform Module. IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA, 2006. URL http://www.kiskeya.net/ramon/work/pubs/security06.pdf.

J. Cihula. Intel's Xen Security Update. Presentation at Xen Summit, January 17-18 2006. URL http://www.xen.org/files/xs0106_intel_xen_security.pdf.

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV'02)*, 2002.

G. Coker. Xen security modules (XSM). Presentation at Xen Summit, April 2007. URL http://www.xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf.

G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of Remote Attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

G. S. Coker, J. D. Guttman, P. A. Loscocco, J. Sheehy, and B. T. Sniffen. Attestation: Evidence and trust. In *Proceedings of the International Conference on Information and Communications Security*, volume LNCS 5308, 2008.

D. Dieckman, P. Alexander, and P. A. Wilsey. ActiveSPEC: A Framework for the Specification and Verification of Active Network Services and Security Policies. In *Proceedings of Formal Methods in Security Protocols*, Indianapolis, IN, June 1998.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, March 1983. ISSN 0018-9448. .

D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and Reasoning about Dynamic Access Control Policies. In *Proceedings of The International Joint Conference on Automated Reasoning (IJCAR'06)*, August 2006.

K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of The International Conference on Software Engineering (ICSE'05)*, May 2005.

P. Frey, R. Radhakrishnan, H. Carter, P. Wilsey, and P. Alexander. A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation. *IEEE Transactions on Software Engineering*, 28(1):58–78, January 2002.

D. P. Guelev, M. D. Ryan, and P.-Y. Schobbens. Model-checking Access Control Policies. In *Proceedings of the 7^{th} Information Security Conference (ISC'04)*, Lecture Notes in Computer Science. Springer–Verlag, 2004.

J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security*, 13:2005, 2004.

V. Haldar, D. Chandra, and M. Franz. Semantic Remote Attestation – A Virtual Machine directed approach to Trusted Computing. In *Proceedings of the Third Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, SACMAT '07, pages 91–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-745-2. . URL http://doi.acm.org/10.1145/1266840.1266854.

D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2011.

T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium*

- *Volume 12*, SSYM'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

C. Kong and P. Alexander. Formal Modeling of Active Network Nodes using PVS. In *Proceedings of Formal Methods in Software Practice (FMSP'00)*, Portland, OR, October 2000.

N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automaton. *CWI Quarterly*, 2(3):219–246, September 1989.

F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example*. Prentice Hall, 2007.

T. Moses. eXtensible Access Control Markup Language version 1.0. Technical report, OASIS, February 2003.

S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and Systems Managment*, 16(3):235–258, Sept. 2008. ISSN 1064-7570. .

T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of Large Installation System Administration*, 2010.

S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proc. of 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer–Verlag.

S. Rueda, H. Vijayakumar, and T. Jaeger. Analysis of Virtual Machine System Policies. In *Proceedings of the Symposium on Access Control Models and Technologies (SACMAT'09)*, Stresa, Italy, June 2009.

A. Schaad and D. Moffett, Jonathan. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2002. ACM. ISBN 1-58113-496-7. .

R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.

Trusted Computing Group. *TCG TPM Specification*. Trusted Computing Group, 3885 SW 153rd Drive, Beaverton, OR 97006, version 1.2 revision 103 edition, July 2007. URL https://www.trustedcomputinggroup.org/resources/tpm_main_specification/.

G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, SACMAT '04, pages 136–145, New York, NY, USA, 2004. ACM. ISBN 1-58113-872-5. . URL http://doi.acm.org/10.1145/990036.990059.

N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 56–65. ACM, 2004.

N. Zhang, M. D. Ryan, and D. P. Guelev. Evaluating Access Control Policies Through Model Checking. In *Proceedings of the 8th Information Security Conference (ISC'05)*, Lecture Notes in Computer Science. Springer–Verlag, 2005.