Model Checking to Verify Computer Security Policies

Robert N. M. Watson
robert@fledge.watson.org

Department of Philosophy,
Carnegie Mellon University,
Pittsburgh, PA

April 26, 1999

Abstract

Model checking is a method of formally verifying properties of finite state machines. By describing operating system structure and system authorization policies using finite state machines, model checking may be used to verify useful properties of policies, improving the chances of developing a secure system. The technique is demonstrated on authorization systems from an Active Network, and from a simplified UNIX-like environment.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal methods of verifying software and algorithms play an important role in computer security. The ability to rigorously demonstrate that a program acts as intended, no matter what user interaction might take place, is a useful building block from which to construct complex but secure systems reliably.

Model checking is a method of formal verification that has only recently been applied to issues of secure computing. It has largely been applied in the area of network protocols for security and electronic commerce to discover flaws or verify the lack thereof [CJM96]. Model checking allows us to formally examine the properties of existing and novel security architectures through relatively simple methods that allow a high degree of automation. To demonstrate the usefulness of this technique, we analyze a security policy and mechanism associated with a simplified UNIX operating system, as well as an *active network*, a distributed computing environment with a sophisticated security requirements. The benefits and limitations of this method are reviewed, as well as directions for continued research. The application of model checking to additional areas in computing, including operating system policy verification, appears to be a promising area for future exploration.

Chapter 2

Background

The application of model checking to computer and network security policies builds on much existing work in both the areas of formal verification and computer security, as well as attempts thus far to apply formal verification to computer security problems.

2.1 Formal Verification

Formal verification has traditionally attempted to follow the model of mathematical proof: an argument is constructed that tracks the algorithm's behavior, showing that if certain initial assumptions hold, the conclusions drawn by the proof must be true. Computer scientists make use of the concept of an *invariant*, a property that is shown to hold throughout a portion of an algorithm, and required for correctness of the overall proof. The task of proving is reduced to the task of maintaining that the invariant is not broken by any of the algorithm's actions, and that the necessary and desirable preconditions and postconditions must hold.

The premise of model checking is a simplification of the objects about which properties are to be proven. While traditional formal program verification targets a Turing-complete universe, model checking limits itself to finite state machines[CGL94a]. This limitation allows for the exhaustive exploration of a system's behavior given all possible inputs and sets of non-deterministic choices which would be prohibited in a Turing-complete system due to the Halting Problem. Exhaustive exploration allows the extraction of dependencies between events, and the verification of properties at all possible points of system execution. Model checking has found greatest use in the area of hardware verification: digital computing lends itself to representation as a finite state machine, as modern computers are effectively large finite state machines. Model checking has been used to verify the correctness of simple logic circuits, as well as complex bus arbitration protocols and cache coherency protocols [CGH+93].

The model checking process involves generating a formal specification of the system as a finite state machine. The model may be generated automatically from an existing electronic-form design. Desirable properties are expressed in temporal logic, and then automatically verified. If one is incorrect, an example sequence of state transitions will be returned that violates the specification. This process, applied iteratively by developers against successive versions of the system, can be used to discover and remove bugs in the system. The proper specification of properties is still left to the human, but the high degree of automation involved in the process is a significant advantage over traditional proof mechanisms that require human guidance to explore the proof space.

In this work, we make use of NuSMV [CCGR99] [CCGRed], a model checker developed by the Instituto per la Ricerca Scientifica e Technologica (IRST) based on Carnegie Mellon University's Symbolic Model Verifier (SMV) [CGL94b]. NuSMV accepts a set of finite state machines in a special-purpose description language, as well as a set of properties to verify, expressed in temporal logic. It applies the model checking algorithm to confirm properties or return explicit

examples of failure. Due to limitations in the model checking software, only the computation tree logic (CTL) and linear-time logic (LTL) subsets of the temporal logic may be verified. This places some limits on the expressions that may be evaluated.

2.2 Computer Security

Computer security becomes relevant with the desire to restrict access to data or services maintained by a computer system. This is exemplified by the move towards multi-user machines and remote access to services through a computer network. The desire to differentiate service offered to different users can be decomposed into a desire to *authenticate* users and services, and to *authorize* requests. Authentication refers to the task of determining who has requested a particular service, or whether the appropriate service is being contacted in behalf of a user. Authorization refers to the task of determining if a request (optionally authenticated) should be carried out. The three major concerns of security, *confidentiality*, *integrity*, and *privacy*, may often be reduced to these two tasks. [Pfl96]

The implementation of authorization may be decomposed into two components: *authorization mechanism*, the scope of restrictions that a particular system offers, and *authorization policy*, the configuration for a mechanism that determines how it should be applied. Policies may apply broadly (system-wide), or have a narrow scope (many objects with many policies). They may also be statically configured over the course of a system run, or be modified dynamically at run-time. A more general and flexible authorization mechanism will support more flexible policies capable of being adapted to a variety of environments. But increasing the degree of flexibility of the system increases the complexity of the system.

2.3 Formal Verification and Computer Security

Increased demands on growingly complex security system inevitably increase the chances of an unintended consequence in the design or implementation of a system. For this reason, formal verification is required for the A* security levels defined in the Orange Book specification [oD85], the highest certifiable levels as described by the US National Security Agency. This requirement suggests that a strong, formal argument of correctness is fundamental to the design of a secure operating system. Traditional proof has been used in this area with varying degrees of success.

The *BAN Logic* has also been used to reason about the correctness of network security issues [BAN89]. BAN is a belief-based logic that allows reasoning about states of belief, along with a set of primitives adapted for use with cryptographic protocols for authentication. While BAN has been used successfully to discover holes and redundancies in existing protocols, its applicability is weakened by the strength of its fundamental assumptions. In its application to public-private key algorithms, weaknesses have been discovered in a supposedly proven algorithm [Low95]. The mechanism used to find the subtle weaknesses was model checking, in the form of the NRL Protocol Analyzer, a special purpose model checker for use with network protocols [Mea96].

While traditional proof of algorithms and implementation has been applied to secure operating systems to verify correctness, the flexibility and simplicity of the model checking technique offers many advantages, including greater automation and the ability generate examples of failure.

Chapter 3

Policy Modeling and Representation

For model checking to be applied to an authorization policy, the policy must first be represented as one or more finite state machines. However, as a policy is essentially a syntactic description and we are interested in semantic behavior, it is necessary to also model other components of the overall authorization system. We must model at least the following components: one or more user processes that make arbitrary requests against the system, the authorization policy itself, and the authorization enforcement mechanism binding the policy to the user process along with any authorization state. The authorization state consists of attributes of the process maintained by the system: transitions in this state are directed by the policy. For the purposes of policy verification, assertions are made about the condition of the authorization state, user process requests, and policy approval responses, all of which are represented as finite state machines. It is desirable to model the system in such a way that appropriate properties may be described, and that the policy may be easily modified so that incremental improvements may be made to improve its verifiable properties.

For the purposes of this discussion, all models will be in the NuSMV mod-

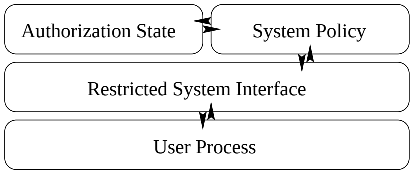

Figure 3.1: Relationship between finite state machines composed to form a policy modeling environment

eling language, which may be used to describe combinations of finite state machines as well as temporal logic assertions on state to be verified [CCGR99] [CCGRed].

3.1 The User Process

The user process is permitted to make a limited set of restricted system interface calls to the system, or in the operating system parlance, "system calls". Because we do not know how any particular process will behave, we model the user process as a finite state machine capable of invoking arbitrary sequences of system calls against the authorization mechanism. This allows the user process model to effectively attempt all desirable and undesirable combinations of requests. In general, system calls may affect the authorization state only indirectly, as mediated by the authorization policy.

In more complex environments, concurrent user processes acting in parallel against one or more sets of authorization state and authorization policies might interact in complicated ways. The model checking paradigm allows us to model this effectively by replicating components of the model (user process, authorization state, restricted system interface) and extending the policy mechanism to

allow interaction.

3.2 Restricted System Interface

A restricted system interface allows us to limit the points of interaction between a system and the user processes. It is traditionally a well-defined API, such as the UNIX POSIX interface [IEE90]. The interface is often broad in scope incorporating a variety of security-related services. Appropriate incorporation of a call's arguments into a model is important so that the model may effectively cover the needs of policy verification, as the arguments almost always significantly influence the nature of a request.

All restricted system interface modules share the characteristic that they accept operation requests and arguments from the user process, and make them available to the policy module. In the demonstration policy models, a stall state may be imposed on the user process by the restricted system interface so that the operation and argument processed by the policy module are not those stored in the user process state. This facilitates the correct modeling of user processes supporting threads where the arguments could be changed during the call, and more accurately reflects the behavior of operating systems such as UNIX.

3.3 Authorization State

The authorization state consists of state necessary to support the set of desirable system policies. In the case of many existing systems, this is predefined and fairly static. For example, regardless of the policies imposed on individual calls in the restricted system interface, the UNIX kernel maintains a numeric user ID and set of numeric group IDs for each active user process. Selecting the breaking point between policy and authorization state is not clear cut, and will depend on the system. Expanding the authorization state expands the scope of policies applicable in the system.

Authorization state may be represented as variables within the authorization mechanism, but separate from the policy. For more complex policies, some component of the authorization state might include the state of the policy finite state machine. For our active network example, the authorization state is stored entirely within the authorization policy finite state machine. In the UNIX example, the state is stored by the syscall handler, and the policy machine is effectively stateless, responding only to changes in the syscall handler state.

3.4 Policy Representation

The system policy is driven by input from the user process via restricted system interface calls, and maintains and observes authorization state. The policy may return true or false to the restricted system interface to indicate approval or rejection of the request being made by the user process, and may maintain long-term state so as to implement more complex policies.

In the models we implement, the approval or rejection is returned immediately based on a NuSMV definition rather than the result of a finite state machine transition. However, finite state machines in the policy module may advance as a result of the request and its arguments, and the current condition of the finite state machine may be used to influence the returned approval if desired.

3.5 Automated Policy Conversion

While the NuSMV modeling language accurately and concisely describes finite state automata and temporal logic conditions, it does not make a great human readable format for real-world use. For the purposes of an Active Network implementation, a more task-specific policy language was defined, albeit still in the form of a finite state machine model. Any equally powerful policy description language may be converted to the NuSMV modeling language and thus could benefit from model checking. Finite state machines present a relatively powerful computation model and are a promising format for policy storage and manipulation. It is easy to envision an alternative storage language that could be converted to a finite state machine representation for policy verification by an automated tool, removing the need for direct interaction with the model checker at a low level.

Chapter 4

Policy Verification

Policy verification consists of running the model checker to verify that certain conditions hold: that is, that the authorization state never leaves certain acceptable sets of states describable via temporal logic expressions, the effective limitations on state transition are the actual policy implemented. The specifications reflect the desired policy, and the model checker will reveal any inconsistencies between them.

4.1 Expression of Properties

The scope of expressible properties is limited by the representation of the modeled system. Properties are expressed in temporal logic, and describe possible conditions of the model and relationships between those conditions. A *state formula* is a boolean expression describing the condition of the model at one snapshot in time. Simple true and false values describing the state variables in the machine may then be combined using the logical operators listed in Table 4.1.

Assertions may be composed to describe relevant features of a particular desired snapshot. If the snapshot matches, the expression will return true;

$!p$	logical not of boolean expression p
$p\&q$	logical and of boolean expressions p, q
$p\|q$	logical inclusive or of boolean expressions p, q
$p- > q$	logical implication of boolean expressions p, q
$p < - > q$	logical equivalence of boolean expressions p, q

Table 4.1: Temporal logic boolean operators

otherwise, false. For example,

```
(authstate = privileged) & (syscall.op = shutdown) ->
        (policy.approval = 1)
```

is interpreted as meaning, "True if the authstate finite state machine is in state privileged, and the syscall.op finite state machine is in state shutdown; otherwise, false."

Linear-time operators add the ability to describe the relationship between two states in time over the course of a single path through the finite state machine. By combining state formulas with linear-time operators, the range of describable situations becomes quite broad. For example,

```
G(authstate = privileged)
```

describes the case where, along a particular path, "The authstate finite state machine is always in the privileged state." The available linear-time operators are listed in Table 4.2.

Np	boolean expression p holds *next* time
Fp	boolean expression p holds sometime in the *future*
Gp	boolean expression p holds *globally* in the future
pUq	boolean expression p holds *until* boolean expression q holds

Table 4.2: Temporal logic linear-time operators

Path quantifiers prefix arbitrary combinations of state formulas composed with linear-time operators. These allow the creation of evaluatable expressions describing sets of possible paths. For example, the

```
EG(authstate = privileged)
```

expression describes the assertion, "There exists a path along which the authstate finite state machine is always in the privileged state." The available path quantifiers are listed in Table 4.3

Ax	x is true for every path
Ex	there exists a path where x holds

Table 4.3: Temporal logic path quantifiers

This temporal logic, CTL*, allows the expression of a broad range of properties of sets of finite state machines. The available model checking software is limited to verifying expressions in one of two subsets: computation tree logic (CTL) or linear temporal logic (LTL) forms of temporal logic. Each places restrictions on the types of linear-time operators used in a particular expression. This limits the expressive power, as there exist temporal logic formulas that cannot be expressed in either of the two subsets.

4.2 Properties of Interest

Verifiable properties are those formulas expressible in the NuSMV temporal logic, describing sets of states globally or conditionally reachable in the state graph. Only a restricted subset of these possible assertions about a finite state machine are actually interesting to system designers. The model checking technique cannot help decide which properties are interesting to investigate, leaving that task to the designer. Several types of properties proved useful in the exploration of sample policies, largely consisting of assertions that certain combinations of events either could never happen or must always occur. For example, requests for privileged operations in the active network example should only be approved if the user process is privileged.

Some useful properties refer only to the state of one module in the system. For example, a property describing the possibility of certain types of authorization being attained at the same time, or in a certain order might be interesting. In this case, temporal logic formulas describing these possibilities in terms of only the authorization state might be used. Similarly, a property might pose that the policy mechanism itself only enters certain states, or passes through them in a particular order.

Other properties involve the interaction between different components of the model. Many properties refer to the policy approval or rejection of particular calls made by the user process against the restricted system interface. For example, it might be asserted that a certain call by the user process would always result in approval, or that a successful call to the UNIX syscall *setuid* would result in later calls to *setuid* failing. Similarly, assertions that the authorization state would always meet certain requirements based on calls from the user process are useful.

Ultimately, the calls of interest are specific to the model defined for verification. Without understanding the semantics and context of a system, the model checker can only return true or false to any policy verification request.

4.3 Verification Results

As limiting as a simple true or false answer may be, the ability to ask useful questions via model checking is still a powerful tool. The "true" result implies that the authorization policy and authorization mechanism satisfy some assertion. If "false" is returned, the model checker provides a concise trace of the behavior leading up to the violation of the assertion.

In either case, an inaccurate representation of the authorization mechanism or authorization policy could result in a less useful, albeit correct answer to the question posed of the model checker. Providing a comprehensive set of assertions about how the system should work is one way to forestall an incorrectly represented policy or mechanism. For example, not only should assertions about events that are never supposed happen be verified, but also assertions about what should be allowed to happen, and that it is possible for these events to occur. In this way, an incorrect representation can be detected.

Chapter 5

Applications of Policy Verification

To explore the effectiveness of model checking as a tool for policy verification, we implemented a simplified active network security interface model, as well as a simplified POSIX/UNIX *setuid* syscall model. The goal of these implementations was to determine whether a finite state machine description of the systems would be adequate to prove useful properties about the systems. The needs of both systems are discussed, and annotations of the finite state machine representations provided.

5.1 Active Networks

Active networks address the need of a rapidly evolving network world: vast quantities of legacy network hardware exist in the field. As bug fixes and protocol improvements become available, the need to rapidly upgrade infrastructure is made difficult by the inflexible static routing architecture. To perform a router software upgrade is a non-trivial administrative duty, often resulting in significant down-time. Active networks provide an alternate model for network routing: just as object oriented programming associated executable code with data structures, active networks transform traditional network packets from data passively routed around the network to a data–code combination that route themselves through the network. For each protocol, an object exists on each router to direct data through the network; new code may be easily deployed to describe and implement new protocols.

While this development promises to dramatically change the face of networking, it imposes serious security concerns. Where traditionally router executable code has been carefully monitored, and changes tightly restricted, the new model suggests a vast quantity of uncertified, possibly anonymous code executing on routers. Complicating factors include the authentication issue, dynamic code generation and execution, and the need for a distributed policy implementing often obscure requirements. At the same time, performance remains a driving factor behind the router market: rapid execution has traditionally been a stronger selling point than correctness (witness the lack of TCP/IP checksum verification on most router platforms).

Providing an adequately efficient security architecture for the active network environment is challenging, especially as the policies applied to "capsules" needs to follow them across the network: the behavioral history of a capsule should be a strong predictor of its future security restrictions. Attaching traces of

past behavior to each capsule is clearly not an adequate solution: the linear (or worse) growth of packet size is simply unacceptable, despite the need for the data in complex policies typical of large distributed computing environments.

One possible mechanism by which to address this problem is to express the active network authorization policy for a particular administrative domain as a finite state machine. This is appealing from the efficiency sense: if the machine describing the policy is static, it may be distributed independently of capsules restricted by the policy. All that need be attached to each capsule is that state information uniquely identifying a location in the state machine (or locations in a set of state machines). For example, if authenticable role is considered to be a state machine, only a light-weight certification of that role needs to be carried with the packet. Persistent recollection of authentication at an entry-point to a security domain rather than reauthentication of code is one clear advantage of this approach, but as shortcuts are used to improve efficiency, the risk of an incorrect policy with unintended consequences increase. The risks also increase as the complexity of a policy expands to describe a large distributed environment with high degrees of functionality provided by routers.

The finite state machine description of authorization policy is implementable within the system. It also has the advantage that it is in a form where the model checking technique could be applied to verify desirable properties.

The TIS extensions to the ANTS active network prototype is modeled for the purposes of this work: based on Java, the security architecture prototype provides a privileged Node object that sets up secured execution environments for mobile code, preventing access the resources of the node in unmediated forms, as well as unmediated communication with other capsules passing through the node.[Wet99]

5.1.1 Active Network Model Modules

Three component finite state machine modules representing the user process, restricted system interface, and policy mechanism, in the active network model are linked together by a *main* module.

```
MODULE main
VAR
        capsule : user_engine(syscall.stall);
        syscall : syscall_engine(capsule.op, capsule.oparg,
                                 policy.approval);
        policy : policy_engine(syscall.op, syscall.oparg);
```

Figure 5.1: Active network model "main" module

5.1.2 Active Network Capsule Representation

In the Active Network parlance, the user process is a "capsule", a combination of mobile code and the data associated with a particular instantiation. The code and associated data are prepared for execution in a restricted environment and allowed to make a limited set of API calls against the system, or "node". For the purpose of this model, only a subset of those calls are represented to demonstrate the technique.

This set of API calls against the node is modeled as the restricted system interface. The calls accepted by the interface are listed in Table 5.1.

Only the *authenticate* call accepts an argument, indicating which identity it is to authenticate against. Because the goal of the model is to demonstrate an authorization framework and not the correctness of the authentication, it is assumed that authentication always succeeds, although we allow for an unauthenticated state.

The *op* argument is allowed to non-deterministically take on any of the

authenticate	which accepts an argument indicating the identity to authenticate against.
unauthenticate	which returns the process to an unauthenticated state
migrate	an argument-free form of the *sendto* call provided in the ANTS framework
privreq	a generic argument-free privileged request; in a real prototype, this might be the request to shutdown the router or modify privileged system state.
exit	the request to terminate the current capsule execution
no-op	a request representing the lack of a request; comparable to non-API CPU calls in a real execution environment.

Table 5.1: Active network restricted system interface calls

```
MODULE user_engine(stall)
VAR
        op : {authenticate, unauthenticate, migrate, privreq,
              exit, no-op};
        oparg : {nil, robert, shafeeq, privileged};
ASSIGN
        init(op) := no-op;
        next(op) :=
                case
                        stall : op;
                        1 : {authenticate, unauthenticate,
                             migrate, privreq, exit, no-op};
                esac
        init(oparg) := {robert, shafeeq, privileged, nil};
        next(oparg) :=
                case
                        stall : oparg;
                        1 : {robert, shafeeq, privileged, nil}
                esac
```

Figure 5.2: Active network user process model

possible restricted system interface requests. In the event that the user process selects the *authenticate* request, it also selects an identity to authenticate to (*oparg*). The *stall* argument to the user process causes the user process to wait while the syscall handler copies in the arguments and passes them to the policy mechanism for approval, and prevents the arguments from being modified.

5.1.3 Active Network Node Representation

In the active network model, the restricted system interface is reflected as a set of potential states for the *op* variable shared between the policy module and the user process. In the restricted API implemented, only the *authenticate* call accepts an argument, in this case the identity to authenticate to.

```
MODULE syscall_engine(userop, useroparg, polapproval)
VAR
        op : {authenticate, unauthenticate, migrate, privreq,
              exit, no-op};
        oparg : {nil, robert, shafeeq, privileged};
        stall : boolean;

ASSIGN

        -- Stall the user process while we inspect
        -- arguments and policy
        init(stall) := 1;
        next(stall) := !stall;

        -- Copy in operation, argument
        init(op) := userop;
        next(op) := userop;
        init(oparg) := useroparg;
        init(oparg) := useroparg;
```

Figure 5.3: Active network restricted system interface model

The stall is introduced to allow state a chance to settle. Copying out user operations and arguments and stalling are not necessary in the model, but more accurately represents the organization of code in the original active network prototype.

5.1.4 Active Network System Policy

The active network policy manager is charged with the task of maintaining authentication information about the process it is supervising, and authorizing the user process to perform various activities. Because the entire policy state must be transfered whenever the capsule migrates to another node, the authorization state is stored entirely within the policy module, rather than being maintained as a separate finite state machine, or in the syscall handler. The authorization state consists of two components: the current authenticated identity (*authstate*), and a policy phase (*processphase*) that tracks the lifetime of the user process with regards to the *migrate* syscall. The policy is intended to only allow a call to *migrate* once during the process's execution. To implement this, the initial phase is set to *phase1*. To successfully call the *migrate* call, the process must currently be in *phase1*. On a call to *migrate*, the state is transitioned to *phase2* which can no longer authorize a call to *migrate*. While this is a simple policy, it servers the purpose of demonstrating the flexibility of an authorization policy stored as a finite state machine.

```
MODULE policy_engine(op, oparg)
VAR
        processphase    : {phase1, phase2};
        authstate       : {robert,shafeeq,privileged,nil};

DEFINE
        approval :=
                case
                        op = no-op : 1;
                        op = unauthenticate : 1;
                        op = authenticate : 1;

                        (authstate = privileged) &
                                (op = privreq) : 1;

                        (processphase = phase1) &
                                (op = migrate) : 1;

                        1 : 0;
                esac;

ASSIGN
        init(authstate) := nil;
        next(authstate) :=
                case
                        op = unauthenticate : nil;
                        op = authenticate : oparg;
                        1 : authstate;
                esac;

        init(processphase) := phase1;
        next(processphase) :=
                case
                        (processphase = phase1) &
                                (op = migrate) : phase2;

                        1 : processphase;
                esac;
```

Figure 5.4: Active network system policy model

5.1.5 Active Network Properties

For the active network, we assume a single security domain, over which we would like to verify that certain authorization properties always hold. For example, it is desirable that certain privileged operations (*privreq*) never succeed unless a privileged authorization state is acquired. This property is expressed in the NuSMV temporal logic as

```
SPEC
        AG(syscall.op = privreq) &
                (policy.authstate = privileged) ->
                (policy.approval = 1)
```

which expresses that the privreq will succeed when privileged, and also

```
SPEC
        AG(syscall.op = privreq) &
                !(policy.authstate = privileged) ->
                (policy.approval = 0)
```

which asserts that the unprivileged process will fail. For completeness, we would also like to verify that the syscall can actually be made by a user process:

```
SPEC
        EF(syscall.op = privreq)
```

NuSMV reports that all of these properties are true (Figure 5.5).

```
-- specification AG syscall.op = privreq & policy.authstate =
   privileged -> policy.approval = 1 is true
-- specification AG syscall.op = privreq & !policy.authstate =
   privileged -> policy.approval = 0 is true
-- specification EF syscall.op = privreq is true
```

Figure 5.5: NuSMV results for active network specification (1)

Similarly, we would like to verify that the process phase migration limit works correctly. First, we ask whether a request to migrate in *phase1* always succeeds:

```
SPEC
        AG (syscall.op = migrate) &
                (policy.processphase = phase1) ->
                (policy.approval = 1)
```

Then we wish to inquire whether such a request in *phase2* will always fail:

```
SPEC
        AG (syscall.op = migrate) &
                (policy.processphase = phase2) ->
                (policy.approval = 0)
```

To complete our investigation, we would like to know that it is possible for the user process to request migration, and that it is possible for the user process to enter phase2.

```
SPEC
        EF (syscall.op = migrate)

SPEC
        EF (policy.processphase = phase2)
```

NuSMV confirms these properties, also, in Figure 5.6.

Through model checking, we are able to formally verify a number of useful properties about our active network model. By extending the active network model and continuing the exploration of the model using temporal logic properties, we can gain some degree of assurance that the model is correct, and that the policy mechanism works as we desire.


```
-- specification AG syscall.op = migrate &
        policy.processphase = phase2 -> policy.approval = 0
        is true
-- specification AG syscall.op = migrate &
        policy.processphase = phase1 -> policy.approval = 1
        is true
-- specification EF syscall.op = migrate is true
-- specification EF policy.processphase = phase2 is true
```

Figure 5.6: NuSMV results for active network specification (2)

5.2 UNIX Authorization Model

UNIX-like operating systems typically have a fairly static authorization policy. That is, leaving aside a small set of authentication state associated with a process, the capabilities of the process do not change much as a result of actions they have performed in the past. The one exception to this is the file system, where permissions on objects may be changed dynamically, and capabilities to modify objects are granted as a result of file *open* calls. We describe a simplified UNIX authorization model excluding file systems, in an attempt to demonstrate properties of a properly implemented *setuid* syscall.

In a manner similar to the active network model, three modules are combined using a *main* module to link their input and output. As with the active network capsule module, we impose a stall state on the UNIX user process to allow for finite state machine advancement in the policy module and syscall mechanism.

```
MODULE main
VAR
        uproc : user_engine(syscall.stall);
        syscall : syscall_engine(uproc.op, uproc.oparg,
                                 policy.approval);
        policy : policy_engine(syscall.op, syscall.oparg,
                               syscall.authstate);
```

Figure 5.7: UNIX model “main” module

5.2.1 Unix User Processes

The UNIX process model is similar to the active network capsule model: the user process is treated as a non-deterministic finite state machine making arbitrary combinations of requests and arguments against the syscall mechanism.


```
MODULE user_engine(stall)
VAR
        op : {setuid, gettime, privbind, exit, no-op};
        oparg : {root, robert, shafeeq};

ASSIGN
        init(op) := no-op;
        next(op) :=
                case
                        stall : op;
                        1 : {setuid, gettime, privbind,
                             exit, no-op};
                esac;

        init(oparg) := {root, robert, shafeeq};
        next(oparg) :=
                case
                        stall : oparg;
                        1 : {root, robert, shafeeq};
                esac;
```

Figure 5.8: UNIX user process model

5.2.2 Syscall Interface

Only a limited set of UNIX syscalls are implemented in the restricted system interface, or “syscall handler.”

setuid	switch the userid to that of the argument (requires appropriate privilege)
gettime	return the current time of day
privbind	bind a privileged network port (required appropriate privilege)
exit	terminate the process
no-op	a place-holder for internal computation not involving the syscall mechanism

Table 5.2: UNIX restricted system interface calls

The *setuid* syscall requires appropriate privilege, or root access in UNIX terminology, to successfully execute. The operation argument is the user id that the process wishes to switch to. In the event that it is called by any user other than root, it should be rejected.

The *gettime* and *exit* syscalls are unprivileged operations that are not intended to affect the authorization state in any way. Normally, these calls would return the current time of day or exit the current process, respectively.

The *privbind* call is intended to reflect an attempt to bind a privileged UNIX port, but it is effectively a syscall that requires privilege but should not affect the process authorization state.

As with the active network model, the *no-op* syscall is intended to reflect internal computation within the process, but it does not affect the policy in any way.

Unlike in the active network model, the authorization state is stored entirely within the restricted system interface, or syscall module. This reflects the hard-coded nature of the UNIX userid. While individual calls may have their own policy decisions, the process authorization state is typically stored independently in the per-process information structure, and is typically not extended (with the exception of file descriptor capabilities). In the model, this information is stored in the *authstate* variable. As with the active network model, the arguments of a syscall are copied out and made available to the policy module during the user process stall state.


```
MODULE syscall_engine(userop, useroparg, polapproval)
VAR
        authstate : {root, robert, shafeeq};

        op : {setuid, gettime, privbind, exit, no-op};
        oparg : {root, robert, shafeeq};
        stall : boolean;

ASSIGN
        init(authstate) := root;
        next(authstate) :=
                case
                        polapproval & op = setuid : oparg;
                        1 : authstate;
                esac;

        init(stall) := 1;
        next(stall) := !stall;

        init(op) := userop;
        next(op) := userop;

        init(oparg) := useroparg;
        next(oparg) := useroparg;
```

Figure 5.9: UNIX restricted system interface model

5.2.3 UNIX Authorization Policy

The UNIX authorization policy is quite simple: privileged operations require the userid, or authorization state to be equal to root. All other operations succeed regardless of the environmental factors. No state beyond *authstate* is required to represent this policy, so the policy is described entirely with a NuSMV stateless define. As a twist, we allow the *setuid* syscall to succeed regardless of the current userid if the argument is the current userid. That is, a user may ask for their identity to be switched to their own, and get a successful return.

```
MODULE policy_engine(op, oparg, authstate)
VAR

DEFINE
        approval :=
                case
                        op = no-op : 1;
                        op = gettime : 1;
                        op = exit : 1;

                        -- can setuid to oneself
                        op = setuid & oparg = authstate : 1;

                        -- root can setuid to anyone
                        authstate = root & op = setuid : 1;

                        -- root can bind any port
                        authstate = root & op = privbind : 1;

                        -- reject everything else
                        1 : 0;
                esac;
```

Figure 5.10: UNIX system policy model

5.2.4 UNIX Properties

While this is a simple model that does not reflect the full complexity of the UNIX operating system, it does allow us to verify some simple properties that are useful in a real-world system.

Using model checking, we can explore the capabilities of a root user. For example, we can determine whether at any point in time, a call made by a user process with *authstate* of root will always be approved by the authorization policy. This is expressed as

```
SPEC
        AG(syscall.authstate = root) -> (policy.approval)
```

At the same time, it is important to us that syscalls made by other users have the potential to fail, which can be checked as,

```
SPEC
        EF(!syscall.authstate = root) -> !(policy.approval)
```

The *setuid* behavior described in the specification would seem to imply that once root access has been given up, it can never be regained using the normal *setuid* call. We can verify this in our model by checking the assertion:

```
SPEC
        AG((!(syscall.authstate = root)) ->
                AG(!(syscall.authstate = root)))
```

We introduce a sample user, shafeeq, who we want to verify properties about. For example, we would like to know that he can make the normal set of syscalls, but be prevented from doing anything questionable. We can verify this behavior by look at a number of expressions that describe desirable behavior. The assertion,

```
SPEC
        AG(((syscall.authstate = shafeeq) &
                (syscall.op = privbind)) -> (!policy.approval) )
```

allows us to verify that a *privbind* call by shafeeq will never succeed. We can also verify that shafeeq cannot successfully *setuid* to root:

```
SPEC
        AG( ((syscall.authstate = shafeeq) &
                (syscall.op = setuid) &
                (syscall.oparg = root)) ->
                (!policy.approval) )
```

NuSMV confirms all of our suspicions about the policy in Figure 5.11.

```
-- specification AG (syscall.authstate = root ->
        policy.approval) is true
-- specification EF (!syscall.authstate = root ->
        !policy.approval) is true
-- specification AG (syscall.authstate = shafeeq \&
        syscall.op = privbind -> !policy.approval) is true
-- specification AG (syscall.authstate = shafeeq \&
        syscall.op = setuid & syscall.oparg = root ->
        !policy.approval) is true
-- specification AG (syscall.authstate = shafeeq \&
        syscall.op = gettime -> policy.approval) is true
-- specification AG (!syscall.authstate = root ->
        AG (!syscall.authstate = root)) is true
```

Figure 5.11: NuSMV results for UNIX specification

It is important to remember that these guarantees make strong assumptions about the underlying system, and talk about the privilege of processes to request certain actions, and not the privileges of users, or the results of programs that are incorrectly written. A guarantee that a user cannot call *setuid* does not mean they cannot reach a state in which they effectively have root access. However, the ability to verify claims such as the ones above is useful: we know that

the internal policy mechanism is consistent with our assumptions, and that the policy we formulated matches the assertions we made.

A more complex modeling of the UNIX system would take into account execution of new programs in an existing process, files and IPC between processes of differing privilege. Even the behavior of buggy software could be modeled by relaxing assumptions of correctness on particular processes, and then observing the effects on assertions that could and could not be verified.

Chapter 6

Limitations to the Model Checking Approach

The model checking approach provides a useful framework for the formal verification of security policies and implementations. However, it suffers from a number of limitations, partially imposed by the structure of this approach, and in part by the underlying model checking paradigm.

6.1 Limits Imposed by Abstraction

As with the BAN logic, an abstraction phase may exist in the verification process. In the UNIX example, the model is an abstraction of an existing UNIX authorization mechanism. While it is clear that differences exist between the simplified model presented here and real-world UNIX implementations, more complex models that attempt to more accurately describe UNIX may also differ from the real system in subtle ways. Showing that the model and the real object have the same properties is difficult, and an inherent limitation to any verification based on a human-derived abstraction.

In part, this may be addressed by involving a machine translation in the abstraction process. For example, an automated mechanism could be used to

derive either the finite state machine model from the real system, or the model could be used to generate the corresponding components of the real system. In the case of the active network example, it is suggested that a finite state machine might actually be the appropriate policy representation language for that application, in which case the same representation can be used both in the application and the verification. If the model is used in the design process, then it becomes a question of whether the implementation meets the specification, and not one of an inaccurate modeling. It is, in a sense, a software engineering issue that is faced today: what to do if your programmers don't implement your specifications. No formal verification on a model can make up for poorly implemented software, as any human translation phase can introduce errors, be it from software to model or model to software.

6.2 Expressiveness of Finite State Machines

Model checking is limited to the checking of properties described using temporal logic over finite state machines. Some policies may not be easily expressible (or expressible at all) as finite state machines. Those that are expressible may suffer from the limitations of model checking in terms of performance: as the number of states grows, so does the space required to verify the model. As the complexity of the properties and model grow, so does the time taken to verify the properties.

Some policies are particularly subject to this degradation of performance: any policy that contains limitations of the form "a certain action is permittable only after x iterations", or any other math or counting-related limitation will suffer from a fair degree of state explosion.

It is easy to observe, however, that many common and useful policies will not suffer from this limitation. Most policies consist of "an action x is not permitted unless prerequisite y is met", where prerequisite y is almost always the possession of some appropriate authorization state. Similarly, it can be observed that all existing computing devices are already effectively finite state machines, as they merely approximate Turing computability given finite resources. An abstraction of a complex finite state machine is likely to be a simpler one, and we suspect that the chances are high that many interesting policies are easily expressible in a finite state machine form.

6.3 Expressiveness of Temporal Logic

Existing model checkers are limited to verifying expressions in limited subsets of temporal logic. This places limits on the security properties that may be expressed. While temporal logic makes it easy to express some types of concepts, the limitation that LTL and CTL logic cannot be used in the same property already places limitations on the properties that may be verified. For example, an expression cannot currently contain both an Up assertion and a EFp expression, and there are CTL* concepts that cannot be expressed in either sub-logic.

It is also not clear that all possible desirable properties of an authorization policy may be described using temporal logic.

6.4 Limits on Parallelism

Model checking is able to deal with parallelism in two ways. First, the finite state machines making up modules may be replicated, with appropriate adaptation. For example, multiple user process modules could be linked to a single policy module in the model. However, this only works for explicitly tested degrees

of replication. The other approach is the use of a form of induction over the degree of replication. The use of this approach is more complex and not currently easy to automate, but it guarantees that an assertion is true for any number of processes. As the techniques become more accessible, it may be that only a combination of traditional proof and model checking can be used to verify statements like "The assertion x is true for any number of user processes."

This conclusion places limits on what may be verified in the most general case. However, as finite limits are placed on most system resources in individual systems, we are able to verify cases with high degrees of replication (given adequate computational power).

6.5 Limits on Dynamic Policies

Model checking is limited to dealing with static finite state machines: that is, the finite state machine to be verified is fixed at the beginning of the verification process. In the real world, authorization policies may change over time, and in arbitrary ways. A simple example of this is user-controlled access control lists on file system objects: ACLs may be changed by the users at run time, and new objects may be created. For simple policies like ACLs, a finite state machine representation may be substituted with little difficulty. But if the policies attached to objects are to be dynamically provided by non-deterministic user processes, then the model checking approach cannot easily be used to verify the results. While limitations on the scope of the represented policies could be made, the increase in complexity and nested nature of the computational models suggests that problems might be encountered in extending this approach. Whether or not useful properties could be shown at all about completely dynamic policy environments is an interesting and under-addressed research question.

Chapter 7

Benefits to this Approach

While this mechanism has limitations, a number of key concepts to making further use of this approach are described.

7.1 Benefits of Model Checking

The inheritance of the positive features of model checking provides a number of benefits to this method. The high degree of automaticity in verification, the generation of examples of violations of a specification, the ability to scale the components, and optionally, automatic model generation, all contribute to the potential success of this method. The benefits of model checking are wide and well documented, especially within the hardware community where it is being used to verify existing specifications and find errors in them.

7.2 Component Isolation

Isolation of components reflects both real-world implementation (user processes must be isolated from the policy mechanism), and the hopes for real-world implementation (the policy should be separable from the mechanism so that it

may be improved). This technique has long been used in software engineering and computer security to improve fault tolerance. We propose that it is an important design goal from both the reliability and security aspects.

7.3 Non-determinism

Support of non-determinism in model checking allows for the representation of an unpredictable user process. This assumption accurately reflects the flexible multi-programmed systems commonly available today. Through the exhaustive analysis provided by model checking, behavior violating the specification may be discovered.

Chapter 8

Future Directions for Research

Further directions for research in this area abound, in extensions to the models presented here, as well as extensions of the model checking approach itself.

8.1 Interacting Policies

In a world with ever-increasing quantities of mobile code, the chances that mobile code will pass through multiple security domains over its lifetime of migration, such as in an active network, are growing. Even non-mobile code may be under the dominion of multiple parties, each of which will want to impose its own security policy on the code. For example, a Java applet running in a web browser on a UNIX machine goes through two layers of policy (at least) before actually gaining access to the resources it requests. Modeling these and other interacting and nested security policies might prove to be an interesting area of further research.

8.2 Parallelism

As discussed in section 6.4, the model checking approach provides various ways to approach the issue of parallelism. Determining the correct approach and exploring the possibilities would help determine the limits on the use of model checking in this field.

8.3 Dynamic Policies

Section 6.5 suggests that the model checking approach may not scale well to systems with dynamic policies. Exploring the best way to represent such systems, and how model checking can address the needs of such systems would prove interesting. One example of such an environment might be in a common router on an active network: a piece of mobile code leaving a copy of itself behind to interface with additional code and data following it would want to protect itself against interference by other parties, and might provide an appropriate policy to the router for protecting the code.

8.4 Automated Policy Translation

Section 3.5 notes that the NuSMV representation format for finite state machines and specifications is accurate but not necessarily the best format in which to present security issues for human consumption. Areas of research might include translation of existing policy formats into finite state machines, as well as ways in which specifications can be more easily constructed.

Related to representation is the issue of common use: if system vendors provide formal models of their security systems, users could both be assured of the soundness of the system, and be able to check their policy against a set of vendor-recommended specifications that should be met to assure security, once they adapt the policy to the local environment.

8.5 Additional Environments

The two examples presented here are quite limited, and serve best to demonstrate the validity of the technique rather than the security of those systems. More complete models of both the active network and UNIX environment would be useful from a design perspective, and also increase the chances of discovering any flaws in those systems.

also for the possibility of discovering flaws in those systems.

There are also many additional systems that might benefit from the application of model checking to their security architecture. Two examples are discussed here: a Multi-Level Security (MLS) system and electronic commerce applications.

8.5.1 MLS Model

Traditional multi-level security systems define a set of security levels, and follow a fixed set of rules for classifying data and preventing unintended sharing of data in inappropriate ways. Model checking would provide an excellent tool to describe and explore the behavior of an MLS system, as such systems offer a very explicit security model.

8.5.2 Electronic Commerce Applications

Model checking has been effectively applied to verify the correctness of on-the-wire electronic commerce protocols. An interesting follow-up question to that

of protocol verification is whether the underlying e-commerce systems behave correctly. For example, to determine whether goods are ever delivered without a completed transaction, or whether access to data is provided without appropriate authorization. These are questions which could be addressed via a policy representation and appropriate authorization mechanism expression.

Chapter 9

Conclusion

While information release and authentication have traditionally been addressed by strong verification mechanisms, system policy design involves a high degree of complexity, making it difficult to apply traditional algorithm and program proof mechanisms. Model checking helps alleviate some of these problems by providing an automated verification mechanism for a model of the security system, allowing the easy verification of assertions expressed in temporal logic.

This technique might prove useful for both the original developer of a system and the end user. For the developer, there are guarantees of correctness in the system as it ships. For the consumer, there is now the ability to verify that the model provided by the vendor is sound, and to check that the consumer-produced policy in a flexible policy scheme does not violate underlying security assumptions of the system. The use of this powerful tool for the verification of security systems appears promising, and is an area open for additional research.

Bibliography

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, 1989.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV a new Symbolic Model Verifier. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the eleventh International Conference on Computer Aided Verification CAV*, Lecture Notes in Computer Science, Trento, Italy, July 1999. Springer Verlag. To appear.

[CCGRed] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software and Tools for Technology Transfer*, To be announced. Invited paper, to appear.

[CGH+93] E. M. Clarke, O. Grumber, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, April 1993.

[CGL94a] E. Clarke, O. Grumber, and D. Long. Verification tools for finite-state concurrent systems. In *Lecture Notes in Computer Science: A Decade of Concurrency: Reflections and Perspectives*, volume 803. Springer-Verlag, 1994.

[CGL94b] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, 1994.

[CJM96] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. 1996.

[IEE90] IEEE. *Information technology–Portable operating system interface (POSIX). Part 1, System application program interface (API) : C* *language*. Institute of Electrical and Electronics Engineers, inc., December 1990. IEEE Std 1003.1-1990; revision of IEEE Std 1003.1-1988.

[Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, Oct 1995.

[Mea96] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[oD85] United States. Dept of Defense. *Department of Defense trusted computer system evaluation criteria*. Dept. of Defense, December 1985. Supersedes CSC-STD-001-83, dtd 15 Aug 83. Library no. S225,711.

[Pfl96] Charles P. Pfleeger. *Security in Computing*. Prentice Hall PTR, second edition, 1996.

[Wet99] David J. Wetherall. *Service Introduction in an Active Network*. PhD thesis, Massachussetts Institute of Technology, February 1999.

Index