T H E U N I V E R S I T Y O F T U L S A

THE GRADUATE SCHOOL

HYBRID ATTACK GRAPHS FOR MODELING CYBER PHYSICAL SYSTEMS SECURITY

by
George Robert Louthan IV

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science
in the Discipline of Computer Science

The Graduate School
The University of Tulsa

2011

T H E U N I V E R S I T Y O F T U L S A

THE GRADUATE SCHOOL

HYBRID ATTACK GRAPHS FOR MODELING CYBER PHYSICAL SYSTEMS SECURITY

by
George Robert Louthan IV

A THESIS

APPROVED FOR THE DISCIPLINE OF

COMPUTER SCIENCE

By Thesis Committee

_____, Chairperson
John C. Hale

_____
Mauricio Papa?

_____
Peter Hawrylak?

ABSTRACT

George Robert Louthan IV (Master of Science in Computer Science)

Hybrid Attack Graphs For Modeling Cyber Physical Systems Security

Directed by John C. Hale

56 pp., Chapter 1: Conclusions

(75 words)

As computer systems' interactions with the physical world become more pervasive, largely in safety critical domains, the need for tools to model and study the security of these so-called cyber physical systems is growing. This thesis presents extensions to the attack graph modeling framework to permit the modeling of continuous, in addition to discrete, system elements and their interactions, to provide a comprehensive formal modeling framework for describing cyber physical systems and their security properties.

ACKNOWLEDGEMENTS

Acknowledgments go here.

The work of Chris Hartney and Matt Young in relation to the National Vulnerability Database and exploit pattern terminology and Carsten Müller with respect to the Common Platform Enumeration is gratefully acknowledged, as is the role of Aleks Kissenger in the whiteboard conversation that resulted in the concept of hybrid link automata.

This material is based on research sponsored by DARPA under agreement number FA8750-09-1-0208. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, or DARPA or the U.S. Government.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Introduction

As computer systems become pervasive across a variety of domains, not only are their interactions with people becoming more frequent; computer systems are also increasingly interacting with the physical world and with each other.

Systems that include both continuous and discrete components are termed *hybrid systems*. When linked together with a significant network component, these systems are sometimes called *cyber physical systems*. They have been targeted as a key area of research by the National Science Foundation because they are becoming pervasive in safety-critical domains such as medical, critical infrastructure, and automotive equipment. This thesis is concerned with modeling the security of these systems and their interactions with each other and the physical world.

1.2 Modeling Frameworks

An excellent argument for the need for new research in modeling cyber physical systems is due to Lee in a 2006 position paper in the National Science Foundation Workshop on Cyber-Physical Systems, a prelude to the NSF's research initiative on cyber physical systems:

> Cyber-Physical Systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect compu- tations and vice versa. In the physical world, the

> passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions. [20]

Existing frameworks for modeling and analysis of purely discrete computer networks are inappropriate for use in these systems because of their inability to capture the continuous domain; they also lack a robust, let alone "inexorable" notion of time. Likewise, existing methods for studying hybrid systems fall short when it comes to modeling the complex distributed networks that are often the hallmarks of cyber physical systems.

1.3 Scope

This thesis presents an extension of the attack graph modeling framework, a discrete domain formalism for studying network security, into the continuous domain to enable it to model cyber physical systems. The goal is to combine aspects of both hybrid systems modeling frameworks, particularly hybrid automata, which best describe systems in relative isolation; and computer network security modeling frameworks, particularly attack graphs, which excel at capturing the complex interrelationships and interdependencies among assets and attacks.

The remainder of this thesis is structured as follows. Chapter 2 provides background in hybrid systems and their modeling methods, introduces past work in attack graphs, and presents a set of case studies in both the hybrid and discrete domains to be used throughout this work. Chapter 3 contributes a specification for the generation process. Chapter 4 introduces in detail the improved attack graph framework to be used as the basis for the hybrid extensions. Chapter 5 introduces the extensions themselves. Chapter 6 delivers some results from this modeling methodology, and Chapter 7 draws conclusions and suggests further work.

CHAPTER 2

BACKGROUND

2.1 Introduction

This chapter provides background and context for the concepts, modeling frameworks, and example domains involved in this thesis. First, hybrid systems and their newer counterpart, cyber physical systems, are addressed. Existing modeling frameworks for hybrid systems, particularly hybrid automata, are defined. Second, an introduction to attack graphs is provided along with a survey of their areas of research.

Next, section 2.4 presents at length the basic attack graph model developed at the University of Tulsa and the components thereof that do not represent a contribution of this thesis. Finally, section 2.5 introduces the specific domains of the two examples used throughout this thesis.

2.2 Cyber Physical Systems

2.2.1 Hybrid Systems

A system with both continuous (frequently physical) components and discrete (frequently digital) components is said to be a *hybrid system*, named for its characteristic blending of the two domains. Examples of hybrid computer systems abound in industrial controls, for example, although hybrid systems may also be fully physical (e.g., a bouncing ball that experiences continuous behavior when rising and falling and discrete behavior when colliding with a surface).

The term hybrid system is an older one that was coined as researchers began to study the newly pervasive reactive systems that arose as programmed control of the physical world became widespread [3]. For several reasons it does not suffice to describe precisely the types of systems with which this work is concerned: a subset of hybrid systems that incorporate a significant computer and networking component.

Nevertheless, the modeling of hybrid systems is well studied and provides a sufficient body of relevent work from which to draw to warrant its inclusion. This chapter includes background on a particularly relevent modeling framework for hybrid systems called the hybrid automaton, which is used in this thesis as the standard benchmark against which to compare hybrid modeling techniques.

2.2.2 Definition

A newer, better term for the systems investigated in this thesis is *cyber physical systems* (CPS). Put simply, a cyber physical system is a networked hybrid system: a networked computer system that is tightly coupled to the physical world.

2.2.3 Challenges

According to the 2008 Report of the Cyber-Physical Systems Summit, "The principal barrier to developing CPS is the lack of a theory that comprehends cyber and physical resources in a single unified framework." [1]

The summit further identified as part of the necessary scientific and technological foundations of cyber physical systems both (1) new modeling frameworks that "explicitly address new observables" and (2) studies of privacy, trust, and security including "theories of cyber-physical inter-dependence" [1], a major theme of this work.

Crenshaw and Beyer recently enumerated four principal challenges in cyber physical systems testing that are equally apt for security: their concentration in safety critical domains, their frequent integration of third-party or otherwise

unrelated systems, their dependence upon unreliable data collection, and their pervasiveness [11].

2.2.4 Hybrid Automata

Definition: A valuable formalism for modeling hybrid systems in isolation and with limited composition is the hybrid automaton of Alur, et al. [3]. This section introduces the version of the formalism described in 1996 by Henzinger [15], to which a reader interested in more than a superficial understanding is referred.

Formally, a hybrid automaton H is made up of a set of real-valued state variables, their first derivatives, a set of operational modes and switches between the modes, and predicates attached to those modes and switches describing the operation of the system in those modes and the discrete transitions between them. One can think of a hybrid automaton as a pairing of a finite state machine whose states (called modes) and transitions (called switches) denote the discrete-domain behavior of the hybrid system, with a set of differential equations attached to each mode, which govern its continuous-domain behavior. Switches may also be labeled in order to permit synchronization across composed hybrid automata.

Modes may be decorated with invariant conditions (which state whether the system is allowed to be in that mode), flow conditions (which state how the continuous domain state variables are permitted to evolve while in that mode), and initial conditions (which state under which, if any, conditions the automaton may begin its operation with that mode). Switches are decorated with jump conditions, which serve as guards on the switch determining both (1) when the switch is allowed to be taken, and (2) the discrete changes in state variables due to that switch's activation.

A simple example of a hybrid automaton is given in Fig. 2.1, which models a simple heater thermostat [15]. The vertices in the automaton represent its operating

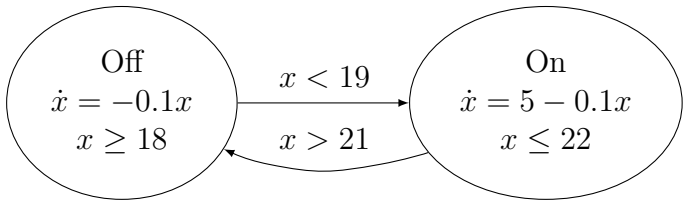

Figure 2.1: Thermostat hybrid automaton

modes, and the edges represent its control switches. In the "Off" mode, the temperature (given by x) must be greater than or equal to 18, and its first derivative with respect to time (denoted $\dot{x}$) is $-0.1x$, which represents a cooling of the environment. When the temperature is strictly less than 19, the switch from off to on is available (but not mandatory until the off mode's invariant condition $x \geq 18$ ceases to be satisfied.) The switch from on to off behaves similarly.

The hybrid automaton model is sufficiently rich to capture many hybrid systems.

Shortcomings: There are some problems with the hybrid automaton model. A hybrid automaton is not guaranteed to have a valid execution, and computing whether it does or not is non-trivial [23]. Model checking has been developed for only some subclasses of automata [16] [14], and many desirable properties of them are undecidable [17].

However, there are even more nagging problems when considering hybrid automata or their variants for the study of cyber physical systems. One of the hallmarks of cyber physical systems is a distributed and highly networked nature. While they provide a natural model for the discrete-continuous boundary, hybrid automata have only a rudimentary notion of communication, no clear means for specifying message passing, and when used in large topologies have significant scaling problems, both computationally and cognitively.

Alternatives: Some attempts have been made to solve the problem of the hybrid automaton's unsatisfactory capability for modeling networks and communication. Particularly, the designation of shared actions and shared variables as "input" or "output" is a popular tactic, used in the powerful hybrid I/O automaton [25] [24], its descendent the timed I/O automaton [19], and also in the PHAVer model checker [14].

The work of this thesis is also something of an outgrowth from an instance of this strategy in which prototypical "hybrid link automata" were developed to model explicit communication channels. An example of the cognitive scalability issues inherent with this design is given in Fig. 2.2, a considered "hybrid link automaton" prototype modeling a link on which messages may be dropped, injected, or delayed, and on which rudimentary mutual exclusion of messages is enforced. This strategy may have a place in modeling some systems but falls short of the goal of modeling complex, interdependent networks of hybrid systems with more conventional computer networks.

Other alternatives may exist in the realm of hybrid systems research. Although the hybrid automata is the gold standard for modeling hybrid systems, other frameworks do exist such as hybrid process algebras [12] [5], an entirely symbolic system with many of the same properties and drawbacks as hybrid automata. Hybrid Petri nets are similar to hybrid automata in that they pair a standard discrete model (in this case, Petri nets instead of finite state automata) with differential equations to model the continuous side of the system or process [9]. Because Petri nets are designed to model distributed systems, hybrid Petri nets may hold some promise, though they are not the topic of this thesis.

2.3 Attack Graphs

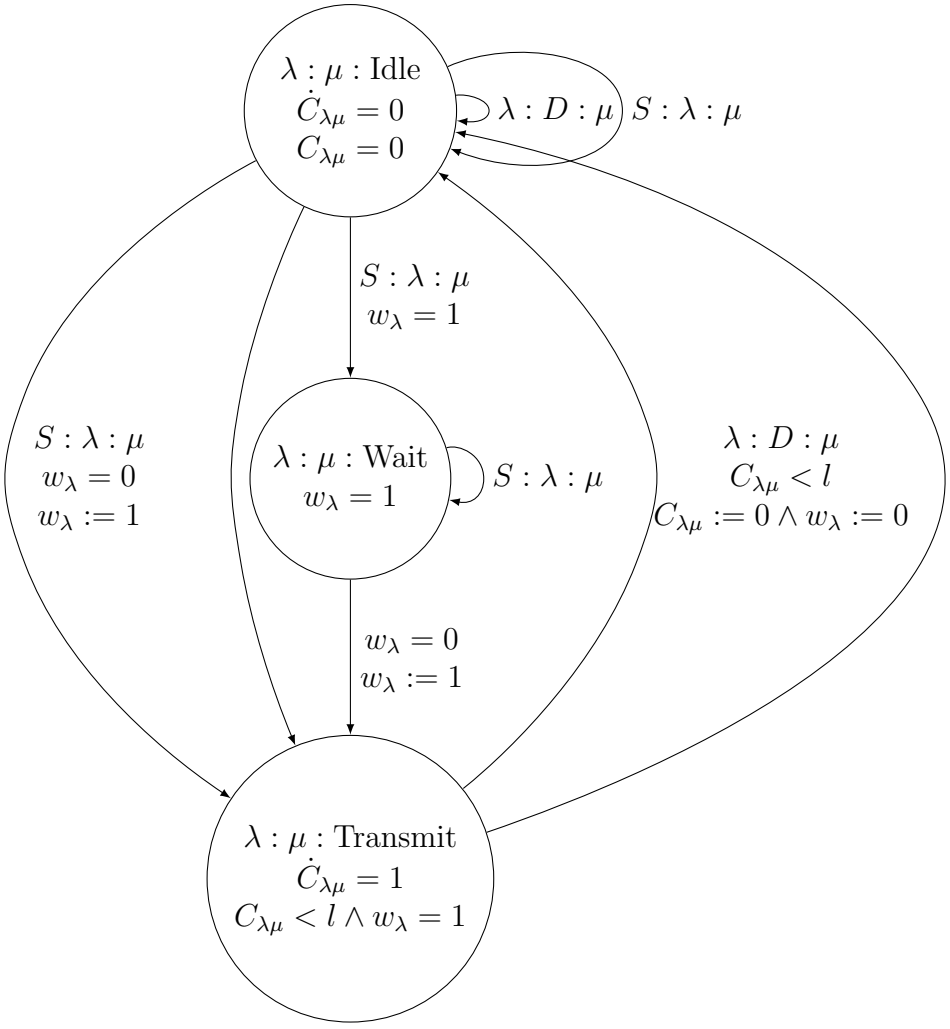

Figure 2.2: Example hybrid link automaton

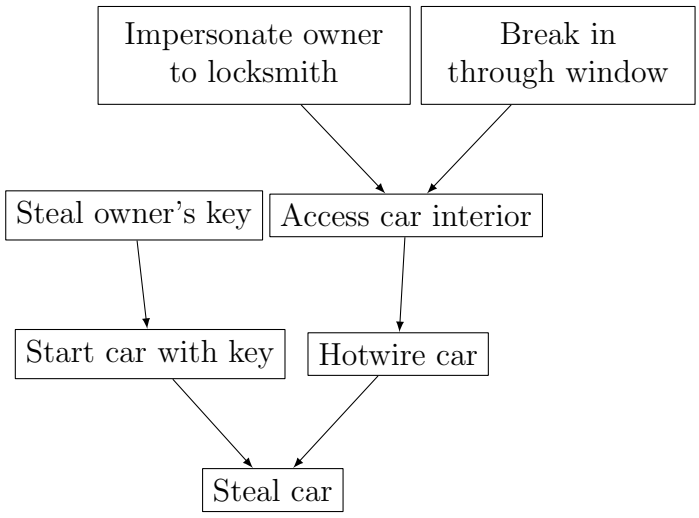

Figure 2.3: Simple car theft attack tree

2.3.1 Introduction

An attack graph is one of several related formalisms that utilize graph theory to model the state space of computer systems attacks. Perhaps they are best introduced when presented as an alternative to a similar model called an attack tree.

2.3.2 Attack Trees

An attack tree is a goal-oriented tree model of an abuse of a system [28]. The root of the tree represents the attacker's goal, and the children of any given node represent the prerequisite activities required to reach that node. For example, consider the goal of stealing a car, which is modeled in a simple attack tree in Fig. 2.3.

The attacker must start the car and drive away; this could be accomplished either by breaking in and hotwiring the car, or by stealing the owner's key and using it to subsequently steal the car. The root of the tree represents the final goal of the theft, with prerequisite goals flowing upward from the leaf nodes.

There are a few features of this modeling method to note. It is goal oriented, meaning that the consequences of the attack are known, and the goal is to ennumerate and and analyze the means by which those consequences could be reached. It is, as

an attack model, agnostic to the underlying system model which makes it difficult to generate automatically. Finally, and perhaps most significantly, it captures the ways in which an attacker's actions interact and depend upon each other.

This threat-centric model is not necessarily the most useful for system stakeholders. It requires that one work backward from the attack to the system state necessary to realize the attack. If, instead, an analyst desires to work from a system characterization and explore the attack space permitted by that system characterization, the attack tree framework must be in some sense turned upside down. Attack graphs do exactly that.

2.3.3 Attack Graphs

Introduction: In contrast to attack trees, attack graphs permit a topology-aware exploratory analysis of the state space of a system. It is a graph theoretic model in which vertices represent individual system states, and edges represent state transitions caused by an adversary. The concept as introduced in 1998 included notions of generalized attack patterns to be bound to state transitions; network elements and their individual configurations; network topology (three characteristics common to all current attack graph iterations); a notion of the attacker's capabilities, and edge weights representing likelihood [26]. A similar structure called a privilege graph was introduced in 1994 [13].

Most approaches to attack graph modeling represent exploits (attack patterns) using preconditions and postconditions [21] since this was suggested in about 2000 [30]. Exploits are chained together by matching preconditions in a state node's underlying system model and applying their postconditions to generate a successor state.

Model Types: The modeling substrates of attack graphs can be broadly separated into two schools of thought, separated by the philosophy that guides

the representation of the underlying network model over which network states and transitions are computer.

A specification of an underlying network model may be done with only very loose restrictions, allowing arbitrary keywords as named qualities and topoligies of network objects. This thesis employs this method. It is also favored in the work of George Mason University [4] [32]. It has the advantage of permitting more straightforward adaptation into the continuous domain, which is the reason it is favored by this work.

An alternate specification method is much more restricted, confining the modeler to certain sets of terms that may, for example, impose explicit computer networking comcepts onto the model [30]. This permits generation and analysis to take a more nuanced view of a network state, including reachability analysis to determine whether a given topology permits communication between two hosts [18]. This approach is favored in the work of MIT Lincoln Laboratory and the University of California, Davis.

Research Directions: Research in attack graphs is spread throughout a variety of pathways. These include evaluating a network's security [4], specification of formal languages to represent attack graphs [30], intrusion detection system integration [31], automatic generation of security recommendations [32], and reachability analysis between hosts in a single network state [18]. For a thorough literature review up to 2005 and more detailed discussion of popular research directions, refer to the work of Lippmann and Ingols [21].

Attack graph work can be considered to fall into four broad categories, referenced occasionally throughout this work by the following names.

Modeling Attack graph modeling concerns the development of the underlying representation and use of that representation to model systems. Terminology

belongs here, as do efforts to automatically generate network models from real networks and exploit patterns from vulnerability databases.

Generation Attack graph generation is the process of building a graph out of a model by closing the state space over its exploits. This is where most performance work is concentrated. A good portion of the work that enables a representation of time to be included in attack graphs belongs here as well. Constraints (such as monotonicity) on the way that state transitions are allowed to progres also fall under the generation category.

Analysis Analysis of attack graphs focuses on drawing conclusions about a system based upon the attack graph generated from its model. Work here includes integration with intrusion detection systems, automatic delivery of mitigation recommendations, and the identification of security consequences with particular states.

Visualization Visualization of attack graphs seeks to reduce or eliminate their known cognitive scalability issues; the goal is to deliver the results of the other three steps in a meaningful fashion.

Its goal being to introduce an architecture for modeling hybrid systems with attack graphs, this thesis is mainly concerned with the modeling stage. However, some amount of work is included in the generation stage, particularly dealing with the progression of time; consideration is also given to analysis, particularly on identifying failure states and on aggregating sufficiently similar states (which also touches visualization).

2.4 Attack Graph Generation and Modeling

This sections presents a version of the attack graph modeling framework specific to traditional information systems. The framework has a very permissive model

that includes notions of assets, qualities, topologies. Assets represent potentially attackable system components; qualities assign an arbitrary string value to a named property of an asset, and topologies bind pairs of assets together with a named connection between them. Exploit patterns with preconditions and postconditions attached to free variables that can be bound at generation time to assets serve to describe potential state transitions.

2.4.1 Intuitive Definition

For the purposes of this work, an attack graph is comprised of the following components.

Assets Assets are the subjects in the attack graph formalism, mainly representing attackable system components. For example, an asset may represent a host on a network, an attacker, or a critical document. Assets are specified with unique names, and they are decorated with *qualities* and *topologies*. Assets can also be used to model users and adversaries if it is necessary to give them explicit properties. A model's collection of assets is fixed at definition time and is not changed by state transitions.

Qualities Qualities represent properties of an asset, such as a software package or version that is installed, whether it is offline, online, or in sleep mode. Qualities are can be considered a key/value pair, where both the key and the value are string tokens. For example, the `host1` asset may have the quality `power` and the value `on`. Together with topologies, qualities make up the network state's collection of facts.

Topologies Topologies represent relationships between two assets. These can be physical such as denoting that a printer is plugged into a computer, logical such as denoting that one host is accessible from another on an adjacent network,

or more abstract such as a particular trust relationship or level of access that a subject has on another. Topologies are directed and named with string tokens. For example, `host1` might be accessible over the network via the web by `host2`, so a directed topology from `host2` to `host1` called `network_remote_web` might be used. Together with qualities, topologies make up the network state's collection of facts.

State A network or system state is comprised of all of the facts about the system's asset collection. A network model's state is fully described by the asset collection and the fact base; given the constant asset collection, a state is uniquely described by its fact base. The fact base is all the qualities and topologies that are valid for that state.

Exploit patterns Exploit patterns are generalized templates for how the actions of the attacker can alter the system state by inserting and removing qualities and topologies (but not assets). They are written as functions that take a number of parameters corresponding to assets, mapping a set of preconditions (facts about the free asset parameters) to a set of postconditions: insert and delete actions on qualities and topologies to update the fact base and therefore generate a new network state.

2.4.2 Formal Definition

Primitive Domains: To further clarify the data types in use by the attack graph formalism, this section provides a more formal definition of their domains. The primitive domains are the most basic domains that describe the atomic units of the formalism: assets, properties (qualities), values (qualities), relationships (topologies), vulnerabilities (exploit pattern identifiers), and parameters (free asset variables in the attack patterns), and operations (used in postconditions representing insert or

$\mathcal{A}$:assets
$\mathcal{P}$:properties (quality names)
$\mathcal{V}$:values (property values)
$\mathcal{R}$:relationships (topology names)
$\mathcal{W}$:vulnerabilities (exploit pattern names)
$\mathcal{I}$:parameters (free asset names)
$\mathrm{Op} = \{\mathrm{ins}, \mathrm{del}\}$

Figure 2.4: Attack graph primitive domains

delete actions). See Fig. 2.4.

Compound Domains: Compound domains comprise the fundamental concepts that are composed of combinations of members of the primitive domains. These form the level of abstraction that it is most convenient to discuss in the articulation of the execution model and in the preceding intuitive definition, for instance.

Qualities Qualities bind an asset to a property to a value; therefore their domain is the cartesian product of those domains. The n subscripts denote that these are bound qualities of a network state, rather than free qualities in exploit preconditions and postconditions:

$$\mathcal{Q}_n : \mathcal{A} \times \mathcal{P} \times \mathcal{V}$$

Topologies Topologies bind an asset to another asset through a relationship; therefore their domain is the cartesian product of those domains:

$$\mathcal{T}_n : \mathcal{A} \times \mathcal{A} \times \mathcal{R}$$

Network states A network state, then, is denoted by a collection of assets, and a fact base of qualities and topologies; the domain of a network state is the

cartesian product of the power sets of these domains:

$$\mathcal{N} : \mathbb{P}(\mathcal{A}) \times \mathbb{P}(\mathcal{Q}_n) \times \mathbb{P}(\mathcal{T}_n)$$

Exploit patterns Exploit patterns, taken from the domain $\mathcal{E}$ depend upon free versions of qualities and topologies, which are parameterized by members of $\mathcal{I}$ rather than $\mathcal{A}$, which are used in preconditions and, when combined with an operator, postconditions:

$$\mathcal{Q}_e : \mathcal{I} \times \mathcal{P} \times \mathcal{V}$$

$$\mathcal{T}_e : \mathcal{I} \times \mathcal{I} \times \mathcal{R}$$

$$\text{Preconditions } \mathcal{P}rc_e : \mathbb{P}(\mathcal{Q}_e) \times \mathbb{P}(\mathcal{T}_e)$$

$$\text{Postconditions } \mathcal{P}oc_e : \mathbb{P}((Op, \mathcal{Q}_e)) \times \mathbb{P}((Op, \mathcal{T}_e))$$

$$\mathcal{E} : \mathcal{W} \times \vec{\mathcal{I}} \times \mathcal{P}rc \times \mathcal{P}oc$$

Attacks An exploit pattern whose parameters have been bound to assets is referred to as an attack. It takes a similar appearance:

$$\text{Preconditions } \mathcal{P}rc_n : \mathbb{P}(\mathcal{Q}_n) \times \mathbb{P}(\mathcal{T}_n)$$

$$\text{Postconditions } \mathcal{P}oc_n : \mathbb{P}((Op, \mathcal{Q}_n)) \times \mathbb{P}((Op, \mathcal{T}_n))$$

$$\mathcal{X} : \mathcal{W} \times \vec{\mathcal{I}} \times \mathcal{P}rc \times \mathcal{P}oc$$

2.4.3 Network Model Specification

The attack graph execution model expanded upon by this thesis does not use this formal notation to represent system elements, instead favoring a more user friendly specification language. This section describes that specification language and the generation process. A network model specification consists of a list of assets

```
network model =
    assets :
    asset_1;
    asset_2;
    asset_3;

facts :
        quality:asset_1,quality_1,value_1;
        quality:asset_2,quality_1,value_2;
        topology:asset_1,asset_2,topology_1;
        topology:asset_2,asset_3,topology_2;
    topology:asset_3,asset_2,topology_2;
.
```

Figure 2.5: Example background network model specification

and an initial fact base (list of qualities and topologies). It begins with the phrase `network model`, followed by the `=` symbol.

The first component of the specification itself is an asset list. The asset list is a semicolon separated list of the names of network assets preceded by the word `assets` and a colon.

Following the asset list, the initial fact base is specified as a semicolon separated list of facts freceded by the word `facts` and a colon. Facts may occur in any order. Qualities are specified by the word `quality`, followed by a colon, followed by the asset in question, a comma, then the name of the quality, then a comma, then the value of the quality. Topologies are specified by the word `topology`, followed by a colon, followed by the names of the source asset, a comma, the name of the destination asset, then the name of the topology. The fact listing, and thus the network model specification, is concluded with a period. An example network model specification is found in Fig. 2.5

2.4.4 Exploit Specification

Exploits are specified using a related scheme. An exploit pattern resembles a

```
exploit exploit_1(asset_param_1,asset_param_2)=
    preconditions:
        quality:asset_param_1,quality_1,value_1;
        topology:asset_param_1,asset_param_2,topology_1;
    postconditions:
        delete topology:asset_param_1,asset_param_2,topology_1;
        insert quality:asset_param_1,quality_1,value_2;
.
```

Figure 2.6: Example background exploit pattern specification

function and contains four parts: a header or signature, preconditions, postconditions, and a terminating period symbol. An example may be found in Fig. 2.6.

An exploit specification begins with the word `exploit` followed by a unique identifier to name the exploit pattern, an opening parenthesis, a comma separated list of parameter names (which are to be bound to assets), a closing parenthesis, and the `=` symbol.

The preconditions list follows. It begins with the word `preconditions`, followed by a colon, followed by a semicolon separated sequence of facts, which adhere to the same grammar as the facts for network model specification.

The postconditions list follows this. It begins with the word `postconditions`, followed by a colon, followed by a semicolon separated sequence of operations. Operations consist of the word `insert` or `delete`, followed by a space, followed by a fact. The exploit pattern is terminated with a period character. Again, an example of this format is found in Fig. 2.6.

Before continuing, a few words on the semantics of exploit processing may reduce the chance of confusion. The `insert` operation inserts a new rule in the fact base, which has the effect of replacing any previous rule with which it conflicts. For example, the insertion of `value_2` as the value of the quality `quality_1` in the example exploit `exploit_1` would replace the existing value of `value_1` specified in the preconditions; no ambiguity is introduced. The same is true of topologies.

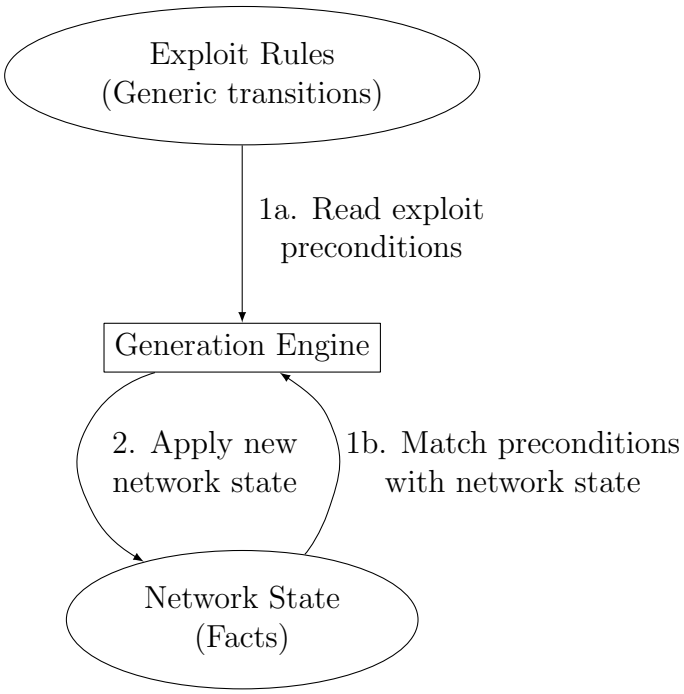

Figure 2.7: Attack graph generation process

2.4.5 Generation Process

This thesis is primarily concerned with modeling attack graphs, but it is worth giving some consideration to the attack graph generation process used in its reference implementation.

Attack graph generation is the process of chaining exploits to enumerate the attack space [8] [26] [29]. Methods for generating attack graphs share a common general architecture among the modern methods that use preconditions and post-conditions in exploit definitions, pictured in Fig. 2.7. The attack graph generation process combines network state and exploit patterns as input, applying exploit postconditions back onto the network state to generate its output of successor states.

This thesis employs this method, which proceeds depending upon a monotonicity assumption: the attacker never moves "backwards". That is, once an exploit is realized, even though the attacker may have the capability of undoing it, he does not do so. A maximum attack graph "depth" (really maximum permitted shortest

path length from the node representing the initial state) is selected before generation begins, and no self loops are permitted, though loops in general are allowed.

As this work's specific algorithm is a contribution of this thesis, it is given more detailed treatment in chapter 3.

2.5 Case Studies

Throughout this thesis, two examples of attacks are used to illustrate the models presented. The first is on a traditional information system, based upon an offensive educational exercise deployed at the University of Tulsa in 2008 involving several chained attacks. The second is the denial of service through battery exhaustion of a simple cyber-physical system of active radio frequency identification (RFID) tags and readers.

2.5.1 Blunderdome

The first case study is an attack on a simulated educational network deployed as part of a security engineering course in 2008. Dubbed the Blunderdome, it featured a firewalled network of two hosts available per attacker. See Table 2.1 for a listing of the stages and their preconditions and results. The attacker was required to log into a login server by cracking its weak SSH key (due to an operating system vulnerability), execute an elevation of privilege (due to a Linux kernel vulnerability), log into the web server, and execute a SQL injection attack to change a simulated grade. The architecture from the exercise is provided in Fig. 2.8 [22].

2.5.2 RFID Denial of Sleep

The second case study used throughout this work is a denial of service attack on the ISO 18000-7 RFID tag inventory system similar to those used by the United States Department of Defense for shipping tracking and the Department of Energy for tracking spent fuel containers [10]. The attack is similar to the ones described by

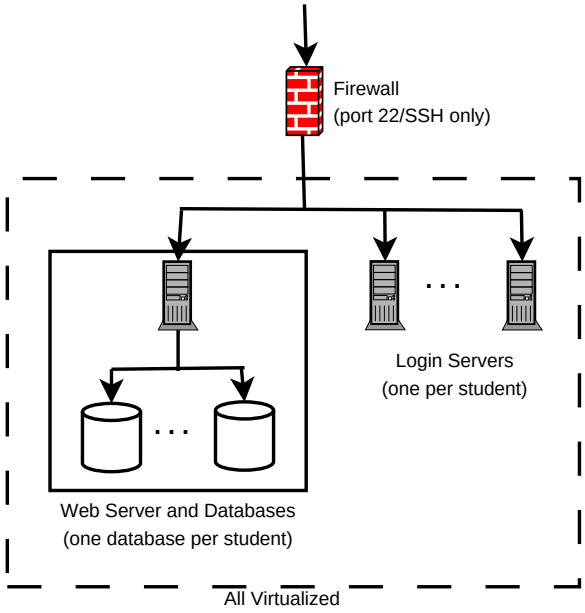

Figure 2.8: Blunderdome network architecture

Stage	Precondition	Attack	Postcondition
Gain remote user access	SSH public key available (given); weak public key	Break weak public key	User privileges on login server
Gain root access	User-level access	Execute `vmsplice` privilege escalation	Root privileges on login server; access to web server credentials
Change grade	Address and credentials for web service	Execute SQL injection	Altered grade in database

Table 2.1: Stages of the Blunderdome attack

Buennemeyer, *et al.* [7], and is of a newly distinguished class of attacks sometimes termed *denial of sleep attacks* [6] [27].

These ISO 18000-7 RFID tags are active and battery powered; they are used for inventory and shipment tracking. In particular, they are used by the Department of Energy to monitor the location and seal status of radioactive material containers, greatly reducing workers' radiation exposure. The batteries on the tags should last as long as possible in order to limit radiation exposure to maintenance workers, and the loss of power to these devices has severe safety and security consequences. An energy draining attack to deplete the tags' batteries could significantly speed this loss of power.

The ISO 18000-7 tags have two modes: an active mode, and a sleep mode in which their power consumption is significantly reduced. The active mode has a 30 second timeout, which will cause them to sleep unless the timer is reset by the receipt of a valid command from the reader or a wake-up signal. In sleep mode, the tags will only respond to a wake-up command, which causes them to enter active mode. A denial of sleep attack occurs when the tag is not permitted to enter sleep mode or is awoken more frequently than normal.

This attack can be realized in two ways. The first is that a second, rogue RFID reader is placed by the attacker within range of some or all of the active tags. The second is for the attacker to compromise an existing reader by hacking into a computer system connected to it via a network. This presupposes that the reader is connected to the Internet or some private network into which the attacker can intrude.

Hybrid automata serve to represent the behavior of the devices themselves quite well. Fig. 2.9 represents the reader (legitimate or rogue), which does nothing but transmit commands and wake-up signals, which can come at any time with no restrictions. Under ordinary operating conditions this might be a few times per day

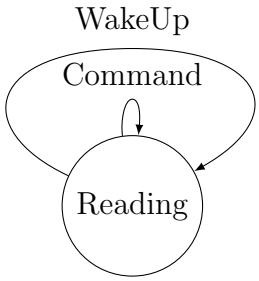

Figure 2.9: Hybrid automaton model of the RFID reader

over the course of several years before the batteries in the tags are drained [10].

Fig. 2.10 depicts a model of the tags. For simplicity, they are shown as starting in the active mode. It has two state variables: c, which represents the active mode timeout clock, and B represents the capacity of the battery. In active mode, the battery drains at a rate of -50 per second, a rate chosen arbitrarily for illustrative purposes only. In sleep mode, the battery drains at a rate of -1 per second. No restrictions are placed on the starting condition of the battery. The two automata are composed using two shared actions: *WakeUp* and *Command*, which synchronize the switches they decorate between the automata.

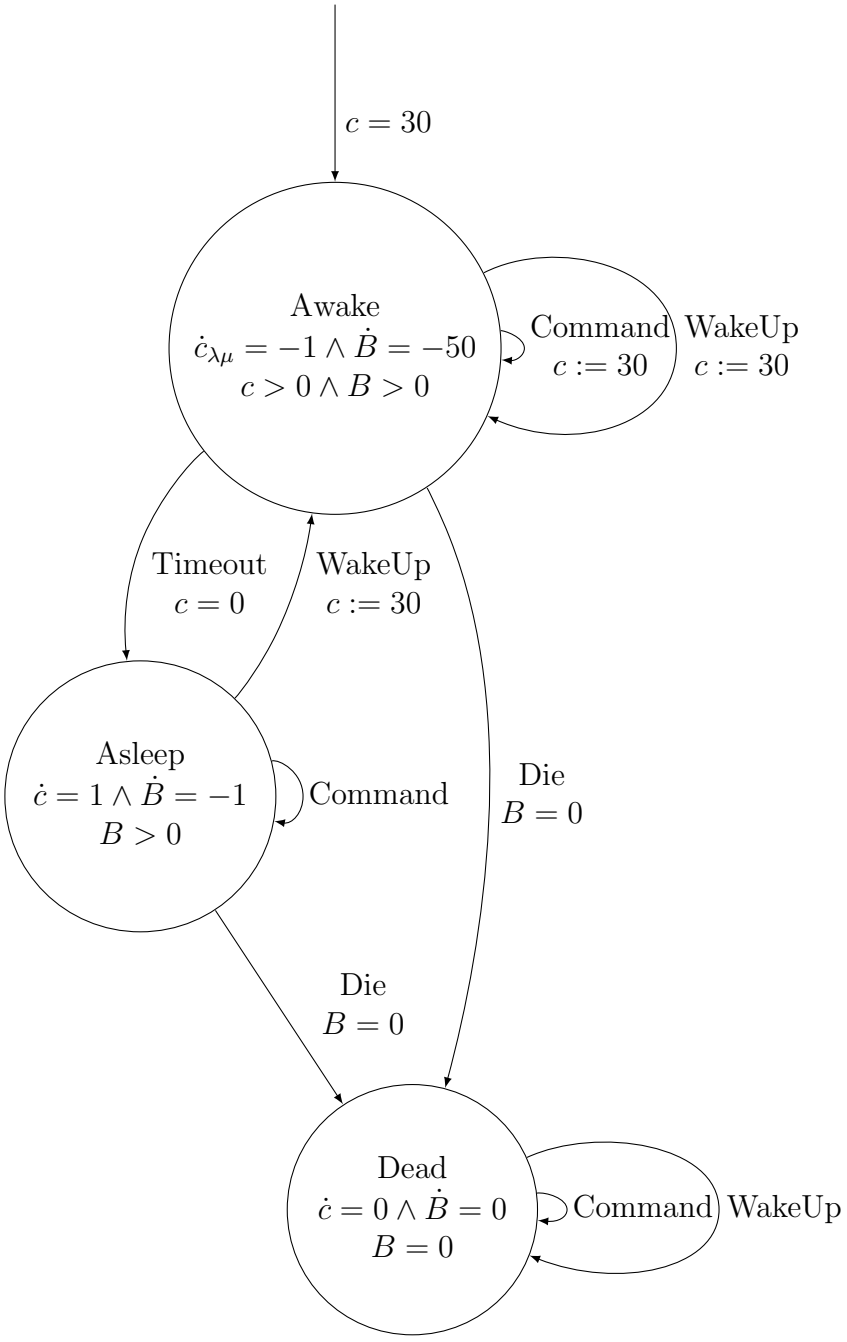

Figure 2.10: Hybrid automaton model of the case study active RFID tags

CHAPTER 3

ATTACK GRAPH GENERATION

3.1 Introduction

Attack graph generation requires a variety of tradeoffs in terms of performance, storage, time, expressivity, and comprehensiveness of output. Development and improvement of the generation methodology is a research area all to itself, one that is not the goal of this thesis. This work is concerned with generation only in terms of building an effective research platform from which hybrid systems modeling can proceed.

Nevertheless, there is a dearth of straightforward presentations of attack graph generation algorithms, optimized for performance or not, in the literature. As a result, this chapter contributes a detailed treatment of the generation algorithm used in this work. Of course, as a main goal of this thesis is to expand on the formalism, the process evolves throughout this work. As expansions are made, effort is given to note the changes to the generation algorithm. Furthermore, the algorithm presented here lags behind the state of the art in attack graph generation performance and represents a contribution only in its explicitness.

3.2 Generation Algorithm and Pseudocode

3.2.1 Description

Attack graph generation proceeds in the following stages. The input to this process is a network model specification (assets and initial fact base), exploit pattern

```
fact-tuple = ('quality', asset, name, value) or
           = ('topology', source, dest, name)

type network_state:
    assets : set of strings;
    factbase : set of fact-tuples;
```

Figure 3.1: Network state datatype pseudocode

specification, and maximum allowed depth (actually maximum allowed shortest path from starting state) of the constructed attack graph.

In sections where it is applicable, pseudocode is provided. As the reference implementation is written in Python, this pseudocode uses some common Python idioms, such as a heavy reliance upon maps (dictionaries).

3.2.2 Network model parsing

The specification of the network model in the attack graph language is parsed into a set of assets and a set of facts (the fact base).

For the purposes of this section, consider network state facts to be represented as ordered tuples: for qualities, `('quality', asset, quality name, quality value)`; and for topologies, `('topology', source asset, destination asset, topology name)`. These are stored in per-state unordered collections without duplicates (sets) usually denoted `factbase`.

An example network model (also called network state) data model is shown in Fig. 3.1.

3.2.3 Exploit parsing

The exploit pattern specification is parsed into a set of exploits, each containing a set of precondition facts and a set of postcondition operations.

An example exploit data model is shown in Fig. 3.2.

```
type exploit:
    name : string;
    params : ordered tuple of strings;
    preconditions : set of fact-tuples;
    postconditions : set of postconditions;

type postcondition:
    operation : 'insert' or 'delete';
    fact : fact-tuple
```

Figure 3.2: Exploit datatype pseudocode

```
def get_attack_bindings(assets, exploits):
 attacks = []
 for exploit in exploits:
  param_perms = (permutations of assets with
                    length len(exploit.params))
  for params in param_perms:
   param_bindings = map with keys=exploit.params,
                             values=params
   attacks.append( (exploit, param_bindings) )
 return attacks
```

Figure 3.3: Attack binding computation pseudocode

3.2.4 Attack binding computation

Next, the set of all possible attack bindings (recall that an attack is the bound version of an exploit pattern) is computed and stored in memory. This represents a list of all exploit patterns, with their parameters bound to every possible asset permutation. That is, for each exploit pattern, a binding must be generated for every possible asset sequence of length n, where n is the arity of the exploit pattern, without repetition. Pseudocode for this stage is provided in Fig. 3.3.

3.2.5 Attack validation

Attack validation is a repeated process for selecting which of the exhaustively produced attack bindings may be applied to a given network state. That is, it comprises attack precondition processing. Two functions are described here. One

```
def get_attacks(network_state, attack_bindings):
 valid_attacks = []

 for attack in attack_bindings:
  if validate_attack(network_state.factbase, attack):
   valid_attacks.append(attack)
 return valid_attacks

def validate_attack(factbase, attack):
 # Recall: attack is of the form (exploit, binding map)
 exploit = attack[0]
 binding_dict = attack[1]

 for precondition in exploit.preconditions:
  if precondition.type == 'quality':
   if ('quality', binding_dict[precondition.asset],
           precondition.name,
           precondition.value) not in factbase:
       return False
  else if precondition.type == 'topology':
   if ('topology', binding_dict[precondition.source],
           binding_dict[precondition.dest],
           precondition.name) not in factbase:
    return False
 return True
```

Figure 3.4: Attack validation pseudocode

gets all valid attacks for a network state; given a network state and a set of attack bindings, it returns the subset of those attack bindings that may be applied to the provided network state. The second validates a single attack binding against a network state, performing parameter binding and checking simple set membership of the generated fact in the network state's fact base.

Pseudocode for these two functions is provided in Fig. 3.4.

3.2.6 Successor state computation

Successor state computation is the second component of attack application; it comprises postcondition processing. Its functionality is straightforward. Given a

network state and an attack binding, it first copies the network state's component pieces. Next, it loops through the attack's postconditions, binds parameters to assets, and removes or adds new facts in accordance with the postcondition operation. Lastly, the network state's components are used to construct a new network state model, the successor state.

Pseudocode for successor state computation is provided in Fig. 3.5. Note that this uses an unspecified function, `get_quality_value`, whose implementation should be straightforward with a variety of strategies. This thesis's reference implementation duplicates network state data in a per-state map that is used only for quality lookups.

3.2.7 Attack graph generation

Here the attack graph generation process begins in earnest. A recursive process, the generation function operates on a collection of analysis states, a remaining allowed depth, and an attack graph. Initially, the analysis state list contains only the initial state, the remaining allowed depth is the maximum depth provided at program invocation, and the attack graph is an empty graph.

Execution halts, and the attack graph is returned, if either the analysis state list is empty, or the remaining allowed depth reaches zero.

For each state in the analysis state collection, the entire list of attacks is checked for compatibility (by checking each of its precondition facts for membership in the analysis state's fact base). For each suitable attack, the analysis state is copied to a new, so-called successor state, and the operations (insertions of facts or deletions of facts) in the attack's postconditions are applied sequentially to the successor state. If the successor state does not exist as a node in the attack graph, it is added. In either case, an edge is added from the analysis state to the successor state. A running list of all new successor states generated (that is, those that were

```
def get_successor_state(network_state, attack):

 successor_assets = deep copy of network_state.assets
 successor_facts = deep copy of network_state.factbase

 # Recall: attack is of the form (exploit, binding map)
 exploit = attack[0]
 binding_dict = attack[1]

 for postcondition in exploit.postconditions:
  if postcondition.operation == 'insert':
   if postcondition.type == 'topology':
    successor_facts.insert(('topology',
        binding_dict[postcondition.source],
        binding_dict[postcondition.dest],
        postcondition.name,
        value=postcondition.value)
   else if postcondition.type == 'quality':
    old_value = get_quality_value(binding_dict[postcondition.asset],
                                  postcondition.name)
    successor_facts.remove(('quality',
                            binding_dict[postcondition.asset],
                            postcondition.name,
                            old_value))
    successor_facts.insert(('quality',
                            binding_dict[postcondition.asset],
                            postcondition.name,
                            postcondition.value))
  elif postcondition.operation == 'delete':
   if postcondition.type == 'topology':
    successor_facts.insert(('topology',
                            binding_dict[postcondition.source],
                            binding_dict[postcondition.dest],
                            postcondition.name))
   elif postcondition.type == 'quality':
    old_value = get_quality_value(binding_dict[postcondition.asset],
                                  postcondition.name)
    successor_facts.remove(('quality',
                            binding_dict[postcondition.asset],
                            postcondition.name,
                            old_value))
 return new network_state(assets=successor_assets,
                          factbase=successor_facts)
```

Figure 3.5: Successor state generation pseudocode

```
def generate_attack_graph(analysis_states, depth, attack_bindings,
                          attack_graph):
 if len(analysis_states) == 0 or depth == 0:
  return attack_graph

 # This will hold the new states added in this iteration:
 successor_states = []

 # For each state to be processed for successors:
 for analysis_state in analysis_states:
  if analysis_state not in attack_graph:
   attack_graph.add_node(analysis_state)

  # For each valid attack in that state:
  for attack in get_attacks(analysis_state, attack_bindings):
   successor_state = get_successor_state(analysis_state, attack,
                                         exploit_dict)
   if successor_state == analysis_state:
    continue

   if successor_state not in attack_graph:
    successor_states.append(successor_state)
    attack_graph.add_node(successor_state)

   attack_graph.add_edge(analysis_state, successor_state)

 return generate_attack_graph(successor_states, depth-1, attack_binding
                              attack_graph)
```

Figure 3.6: Attack graph generation pseudocode

newly added to the attack graph) is maintained. When all possible attacks on all analysis states have been applied, the function recurses, with the successor states becoming the new analysis states and the remaining permitted depth decremented by one.

Pseudocode for this process is provided in Fig. 3.6.

3.3 Example

In order to aid understanding the generation process, this section demonstrates

```
network model =
  assets :
    asset_1;
    asset_2;
    asset_3;

  facts :
    quality:asset_1,quality_1,value_1;
    quality:asset_2,quality_1,value_2;
    quality:asset_3,quality_1,value_1;
    topology:asset_1,asset_2,topology_1;
    topology:asset_3,asset_2,topology_1;
.
```

Figure 3.7: Illustrative example network model

a sample attack graph generation process using an example. The reader is warned to avoid searching for meaning or design in the selection of these assets, qualities, and topologies; they are intended to be illustrative only, and any relation to real world networks, problems, attacks, or situations is purely coincidental. This section will use the network model specification in Fig. 3.7 and the exploit patterns in Fig. 3.8.

Fig. 3.9 represents the initial network state (denoted State 0) specified in this example. Note that Fig. 3.9 is *not* an attack graph, merely a convenient graph based representation of the example network in use here. Each node represents an asset in the network state; it is labeled first with the state number (in this case 0) and the asset name, then with a listing of its qualities (in this case, there is only one). Likewise, the edges that represent topologies are labeled with the topology name they represent.

Execution of the generation process begins by specifying a maximum "depth" of generation, which will be 2 for the purposes of this exercise, and by creating an initial list of states for analysis, which contains only State 0.

Generation proceeds by examining each analysis state, in this case only State 0. First, the generation function creates a list of all valid attacks on State 0.

```
exploit exploit_1(asset_param_1,asset_param_2)=
  preconditions:
    quality:asset_param_1,quality_1,value_1;
    topology:asset_param_1,asset_param_2,topology_1;
  postconditions:
    delete topology:asset_param_1,asset_param_2,topology_1;
    insert topology:asset_param_2,asset_param_1,topology_1;
.

exploit exploit_2(asset_param_1,asset_param_2)=
  preconditions:
    quality:asset_param_1,quality_1,value_2;
    topology:asset_param_1,asset_param_2,topology_1;
  postconditions:
    insert quality:asset_param_2,quality_1,value_2;
.
```

Figure 3.8: Illustrative example exploit patterns

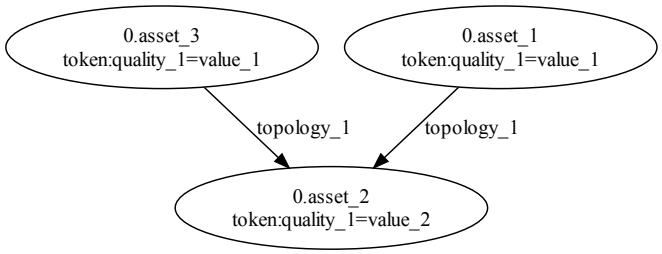

Figure 3.9: State 0 of the illustrative discrete example

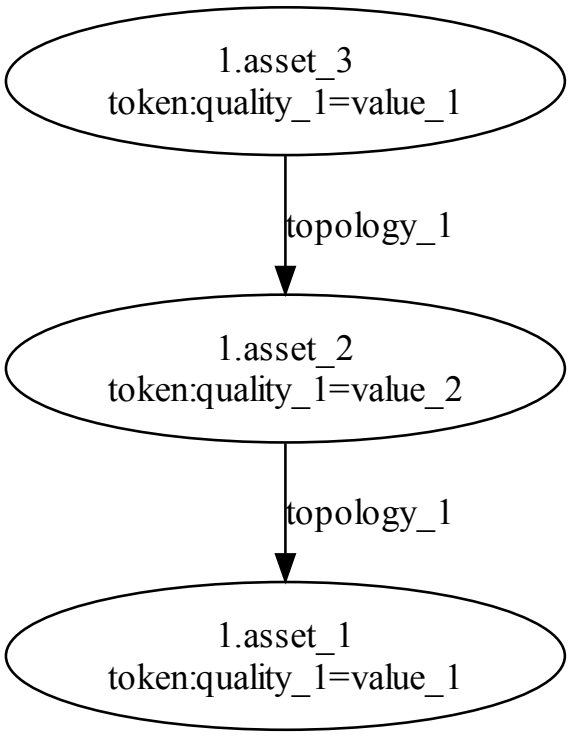

Figure 3.10: State 1 of the illustrative discrete example

Two such bindings are permitted: `exploit_1 (asset_1, asset_2)`, and `exploit_1 (asset_3, asset_2)`. Both bindings result in new states: the first in State 1 (Fig. 3.10), and the second in State 2 (Fig. 3.11). Edges are added from State 0 to each as they are generated. The current state of the attack graph is illustrated in Fig. 3.12. Both of these states are added to the successor state list.

Remaining depth is reduced to 1, the successor state list becomes the analysis state list, and generation proceeds again. In this case, the analysis states are State 1 and State 2. Analysis begins with State 1. Two attacks are possible: `exploit_1 (asset_3, asset_2)` and `exploit_2 (asset_2, asset_1)`. Both create new states, with the former generating State 3 (Fig. 3.13) and the latter generating State 4

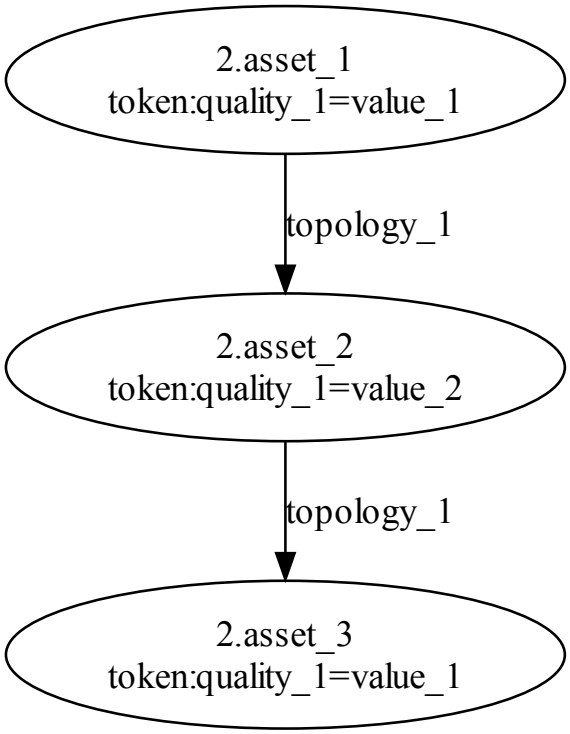

Figure 3.11: State 2 of the illustrative discrete example

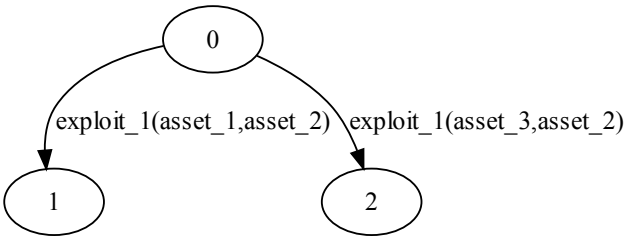

Figure 3.12: Attack graph after first iteration

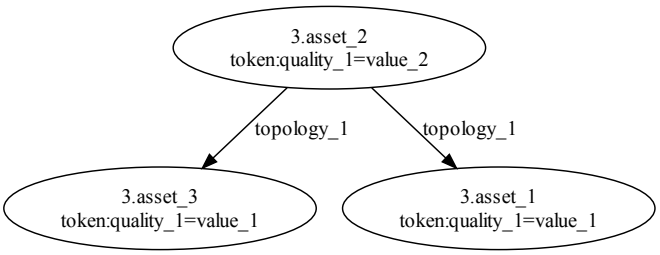

Figure 3.13: State 3 of the illustrative discrete example

(Fig. 3.14). Both of these are new states, and they are added to the attack graph with the appropriate edges, as well as to the successor state list.

Generation continues by addressing State 2. In State 2, two possible attacks are returned: `exploit_1 (asset_1, asset_2)` and `exploit_2 (asset_2, asset_3)`. The first attack generates State 3, an existing state. Since that state is already in the attack graph, it is not added to the successor state list or to the attack graph again; instead, only an edge is drawn. The second attack generates State 5 (Fig. 3.15), which is new and is added to the attack graph and the successor state list.

For the next invocation, the successor state list becomes the analysis state list, and the depth is reduced to 0. Although there are more successor states, the depth limit has been reached. Therefore, generation halts. The final product attack graph is illustrated in Fig. 3.16. Incidentally, if execution had been allowed to continue until there were no more successor states, the generated attack graph would be the one in Fig. 3.17.

This concludes the description of the basic attack graph modeling framework employed at the University of Tulsa. The chapters following this one are devoted to this framework's expansion and analysis.

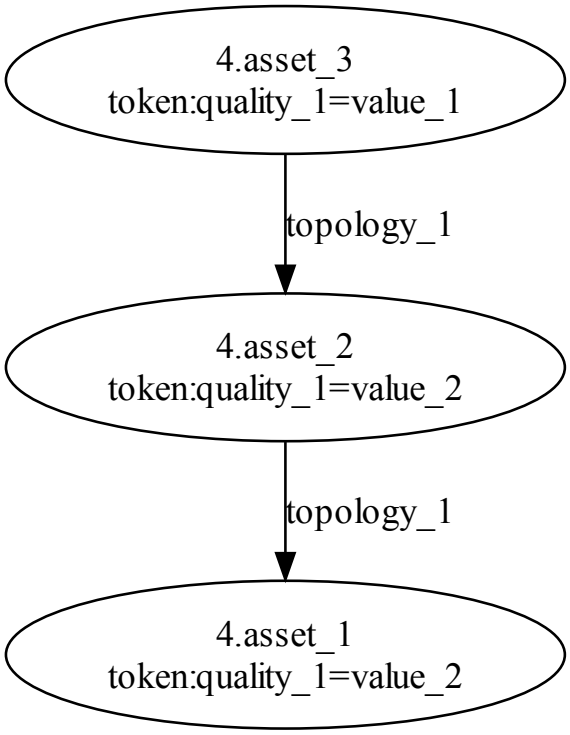

Figure 3.14: State 4 of the illustrative discrete example

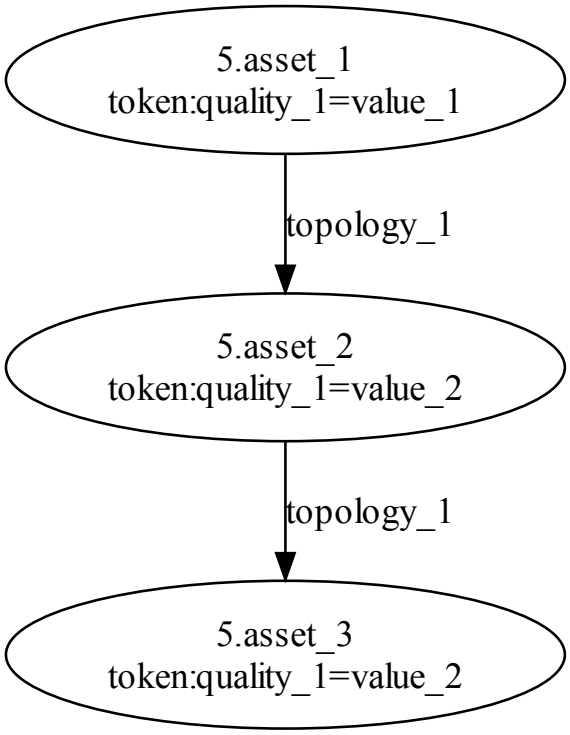

Figure 3.15: State 5 of the illustrative discrete example

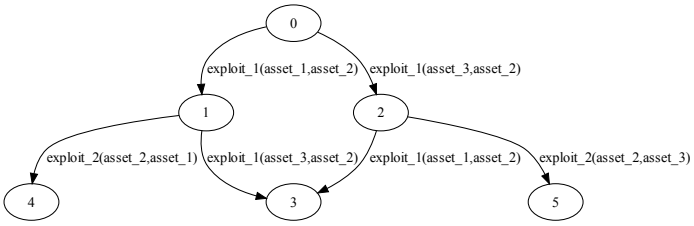

Figure 3.16: Attack graph after first iteration

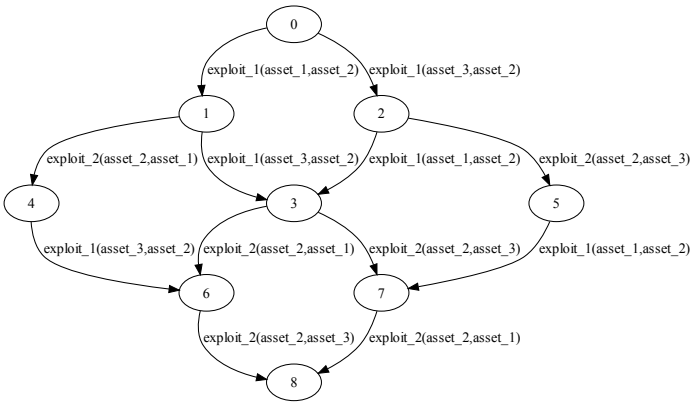

Figure 3.17: Complete illustrative attack graph

CHAPTER 4

DISCRETE EXTENSIONS

4.1 Introduction

For various reasons, the adaptation of the attack graph formalism described in the preceding chapter for hybrid purposes depends upon the addition of a number of entirely discrete elements. Their inclusion produces a transitional attack graph formalism that is not yet appropriate for hybrid modeling but that contains a number of enhancements suitable for discrete system modeling.

This chapter introduces these enhancements, which include a working lexicon of standard terms for topologies and qualities based upon the Common Vulnerability Enumeration and National Vulnerability Database; some syntactic changes to ease the modeling of information systems; a scheme for integrating with the Common Platform Enumeration; and some state predicate analysis tools.

4.2 Working Lexicon

4.2.1 Introduction

Because of the unrestricted nature of the terms available to a modeler in this version of the attack graph framework, in order to proceed systematically some conventions must be established in the use of terms. This thesis recommends a set of conventions designed to ease the goal of automated exploit pattern extraction from the National Vulnerability Database (NVD) maintained by the National Institute of Standards and Technology (NIST), which among other roles indexes MITRE's

Common Vulnerability Enumeration (CVE). "NVD includes databases of security checklists, security related software flaws, misconfigurations, product names, and impact metrics" [2].

4.2.2 National Vulnerability Database

Several concepts are used across the NVD's index of vulnerabilities; they strongly inform the working lexicon employed in this thesis and include a vulnerability's access vector and impact type.

Access vector: The access vector of a vulnerability on NVD refers to the logical location from which the attacker may launch the attack. The possible attack vectors according to the National Vulnerability Database are as follows.

Local The vulnerability is exploitable through physical or local account access to the device on which the vulnerability resides.

Adjacent Network The vulnerability is exploitable through access to a network that is adjacent to the vulnerable host; that is, the attacker must be in the same broadcast domain or collision domain (e.g. the same network segment or VLAN).

Network No local or adjacent access is required to exploit the vulnerability; in other words, it is exploitable over the Internet.

Impact Type: The vulnerability's impact type on NVD places the effects of the vulnerability's exploitation on the target into one of the following categories. , and another for some other type of privileged access.

Confidentiality Confidentiality impacts allow the unauthorized disclosure of information (corresponding to STRIDE's "information disclosure")

Integrity Allows modification of data (corresponding to STRIDE's "tampering")

Availability Availability impacts correspond to disruptions of service (STRIDE's "denial of service")

Security Protection The security protection category refers to the effects of exploits that provide unauthorized access to the target. This may be either general system access or application access. The security protection category corresponds roughly to STRIDE's "elevation of privileges" and also partly encompasses "spoofing identity". It has three subcategories:

User access This subcategory refers to the attacker's gaining user level access to the operating system.

Administrative access This subcategory refers to the attacker's gaining root level access to the operating system.

Other access This subcategory refers to any other type of privileged access on the target.

4.2.3 Topologies

Introduction of actual terms begins with topologies, but first a note about the modeling of adversaries is needed.

Two approaches are possible for modeling the attacker. The first is to design the network model so that it is, in a way, from the attacker's perspective. In this scheme, henceforth the *first person* strategy, the attacker is treated as implicit, and its access and connections are considered properties of the system itself. The adversary's access level to a server, for instance, would be modeled as a quality of that server asset.

In the second approach, henceforth the *third person* strategy, the attacker is modeled as a first class part of the system: as an asset. Its properties, connections,

and access levels are modeled as qualities and topologies of the adversary asset. This strategy, which is the one employed primarily in this thesis, can model multiple adversaries. Aside from that difference, they are roughly equivalent and a matter of taste on behalf of the modeler and analyst. This section's terminology, however, assumes the third person model.

Two types of topology terms are introduced in this section: connection and access. These correspond directly to the two NVD concepts introduced in the previous section.

Connection Topologies: Connection topologies refer to how two assets are connected, over the network or otherwise. They may be local, adjacent, or network connected. These connection topologies begin with the word `connected`, followed by an underscore, then one of the three connection types: `local`, `adjacent`, or `network`. For local and adjacent connections, this is all that is necessary.

For network connections it is necessary to specify the available protocols individually. These are done by appending another underscore, then the lower case version of the standard abbreviation for the protocol (e.g. `connected_network_http` for web or `connected_network_ssh` for secure shell).

Connections are one-way: the source of the connection topology is considered to be the "client" in the relationship, and the destination of the connection is considered the "server", where such distinctions are meaningful.

Access Topologies: Access topologies refer to trust relationships and distinguish what kind of access one asset (possibly a program, individual, attacker, user, or other security principal) has to another. Much like the subcategories of the Security Protection impact type, there are three basic types of access topology (plus the lack of a topology, which signifies a lack of any noteworthy trust or access relationship): user access, root access, and other access.

They are named using the word `access`, followed by an underscore, followed by the lowercase name of the access type, then, if the access type is `other`, an optional underscore delimited description of the access type (perhaps the name of the application whose access level is in question). Examples include `access_user`, `access_root`, `access_other_apache`, or `access_other_vsftpd`.

4.2.4 Qualities

Although qualities are used in a more *ad hoc* fashion to describe entity-specific asset properties, there is still a role for a few standard quality structures in the lexicon, namely a simple "status" type of property to be used in determining whether an asset is enabled or disabled.

Status: The status property is simple but powerful. Each asset that represents a host has a quality named `status`. It may take the value `up` or `down`. This allows exploits against or involving the host in question to require that it be online as a precondition, and it provides a simple mechanism of action for denial of service attacks to be modeled – they simply have the postcondition of a `status = down` quality fact.

Later in this chapter, a simple preprocessor combined with a dash of syntactic sugar is introduced to automate most of the status modeling process across both exploits and the network model.

4.3 Syntactic Sugar

4.3.1 Directional Topologies

This section introduces a new notation for specification of topologies. As it is common to model bidirectional topologies, an additional convenience syntax is now permitted to specify the directionality of a topology. As before, topologies are

specified by the word `topology`, followed by a colon, followed by the names of the assets in question; however, in the new syntax, they are separated not by a comma but by a directionality symbol: `->` to represent a one-way topology from the left asset to the right asset, or `<->` to represent a two-way topology. To promote readability, there is no `<-` directionality permitted.

Furthermore, the model has no innate distinction between one-way and two-way topologies except the `<->` shorthand for specifying symmetric topologies. That is to say, a bidirectional topology fact is actually implemented as two unidirectional topology facts. Therefore, for example, if a bidirectional topology exists in the fact base, one of its component directional topologies may be removed by the realization of an exploit without affecting its reverse.

4.3.2 Value Assignment

In preparation for the introduction of real-valued facts, the existing discrete type of qualities, henceforth called "token valued", are given a special assignment operator. Simply put, token values are assigned with the `=` operator and tested with the `=` and `!=` operators. This enables a robust distinction between assignment operators and relational operators.

Additionally, `delete` operations in postconditions take only a quality name, with no value required (eliminating the confusing possibility of a command to `delete quality:a,q!=5`, one of several deletion commands that has little meaning).

4.3.3 Host Declaration and Status Proprocessing

To support the use of status properties of host assets, a set of shorthand syntax is now provided for denoting which assets represent hosts. In practice, this has been observed to be the majority of modeled assets, so the shorthand is highly convenient for modelers. Host denotation and status processing is done in two places: in the declaration of assets, and in the parameter list of exploits.

In the asset list declaration, up hosts are denoted by prefixing their name with the symbol `@`, a signal to the host preprocessor to automatically add the `quality:hostname,status=up` fact; and down hosts are denoted with the `!@` symbol, automatically adding the down host quality fact.

Similarly, in exploit parameter lists, asset parameters that must have the up status precondition are declared by prefixing their names with the up host symbol `@`; likewise with the down symbol `!@`. If no host status preconditions are required, no symbol is needed. The host status preprocessor automatically adds the required preconditions; if a host status postcondition is necessary, for example in the case of a denial of service attack bringing a host offline, the status property must be manually specified; this is not a significant burden, as this case arises considerably less frequently.

4.3.4 Global and grouped exploits

For a variety of modeling tasks, it is convenient to cause an exploit to be fired on all possible bindings simultaneously or to synchronize two exploits' firing with one another. Two optional keywords are now permitted at the beginning of the exploit header for this purpose: `global` and `group()`. The `group` keyword takes a single argument.

Their semantic behavior is as follows. When a global exploit is fired on one binding of assets, it will also be fired on all other assets for which a valid binding exists, and the corresponding state transition will be considered a single edge on the attack graph. When a group exploit is fired on a binding of assets, all other exploits denoted with the group keyword and the same group argument will also attempt to fire on the same asset binding. Obviously grouped exploits require the same number of parameters.

4.3.5 Platform Properties

Platform facts are new types of facts introduced by this work. They use the specification used by MITRE's Common Platform Enumeration (CPE), a component of the Security Content Automation Protocol (SCAP). (TODO: CITE) They can specify operating systems, applications, and hardware.

A CPE platform fact simply a valid CPE URI tied to an asset. The CPE URI is comprised of the word `cpe`, a colon, then a slash, then the part type (`o` for operating system, `a` for application, or `h` for hardware), then a colon, then the CPE vendor abbreviation, then a colon, then the CPE product abbreviation, a colon, the version, a colon, the update, a colon, the edition, a colon, then the language. Version is required, but the rest is optional.

For example, to specify that a host is running Adobe Reader version 8.1, a platform fact named `cpe:/a:adobe:reader:8.1` would be used. Platforms are matched according to the algorithms provided in the CPE specification, permitting blank (wildcard) fields and the prefix property.

4.4 Generation process changes

4.4.1 Bidirectional topologies

4.4.2 Platform properties

4.4.3 Precondition matching

4.5 Examples

4.5.1 Illustrative

4.5.2 Blunderdome

4.6 Analysis

CHAPTER 5

HYBRID EXTENSIONS

5.1 Introduction

An objective of this thesis is to make the first known foray into the realm of attack graphs with real-valued components. This chapter introduces the additional modeling syntax and generation semantics required to permit real values for qualities and topologies. Furthermore, some initial attempts at representing the passage of time are described.

Some of the requirements that this new hybrid scheme places on the modeler have turned out to be more onerous than desired. However, since the use of attack graphs on hybrid systems with real valued properties is quite nascent, being introduced in this thesis, there is a significant contribution to their study even in introducing less than ideal solutions to their inherent problems.

5.2 Definition of New Syntax

5.2.1 The update operator

5.2.2 Terminology

5.2.3 Topology values

5.2.4 Operators

5.2.5 Restrictions

5.3 Implementation challenges

5.3.1 Topologies

5.3.2 Matching

5.3.3 Updating

5.3.4 Variable updates

5.4 Time

5.4.1 Rate qualities

5.4.2 Time exploits

5.5 Time State Aggregation

5.6 Modeling challenges

5.7 Examples

5.7.1 Blunderdome

Added Capability:

5.7.2 Denial of Sleep

CHAPTER 6

RESULTS

6.1 Performance

6.2 Hybrid Capabilities

6.3 Analysis Capabilities

6.4 Toward Visualization

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

7.1.1 Summary

7.1.2 Shortcomings

7.2 Related and Future Work

7.2.1 Network Model

7.2.2 STRIDE and DREAD

7.2.3 Exploit Model

7.2.4 Exploit Pattern Enhancement

7.2.5 Hybrid System Equivalence

7.2.6 Model Checking

7.2.7 Analysis and Visualization

7.2.8 Generation

7.2.9 Analysis

BIBLIOGRAPHY

[1] Report: Cyber-physical systems summit. Technical report, National Science Foundation, 2008.

[2] National vulnerability database version 2.2, 2011. Web page. Retrieved April 13, 2011.

[3] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid systems*, pages 209–229, 1993.

[4] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM, 2002.

[5] J.A. Bergstra and CA Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2-3):215–280, 2005.

[6] M. Brownfield, Y. Gupta, and N. Davis. Wireless sensor network denial of sleep attack. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 356–364. IEEE, 2005.

[7] T.K. Buennemeyer, G.A. Jacoby, W.G. Chiang, R.C. Marchany, and J.G. Tront. Battery-sensing intrusion protection system. In *Information Assurance Workshop, 2006 IEEE*, pages 176–183. IEEE.

[8] C. Campbell, J. Dawkins, B. Pollet, K. Fitch, J. Hale, and M. Papa. On Modeling Computer Networks for Vulnerability Analysis. *DBSec*, pages 233–244, 2002.

[9] R. Champagnat, P. Esteban, H. Pingaud, and R. Valette. Petri net based modeling of hybrid systems. *Computers in industry*, 36(1-2):139–146, 1998.

[10] K. Chen, H. Tsai, Y. Liu, and J. Shuler. A Radiofrequency Identification (RFID) Temperature-Monitoring System for Extended Maintenance of Nuclear Materials Packaging. In *Proceedings of 2009 ASME Pressure Vessels and Piping Division Conference, Prague, Czech Republic*, 2009.

[11] T.L. Crenshaw and S. Beyer. UPBOT: a testbed for cyber-physical systems. In *Proceedings of the 3rd international conference on Cyber security experimentation and test*, pages 1–8. USENIX Association, 2010.

[12] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005.

[13] M. Dacier and Y. Deswarte. Privilege graph: an extension to the typed access matrix model. *Computer Security ESORICS 94*, pages 319–334, 1994.

[14] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *Hybrid Systems: Computation and Control*, pages 258–273, 2005.

[15] T.A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE, 1996.

[16] T.A. Henzinger, P.H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):110–122, 1997.

[17] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[18] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *2009 Annual Computer Security Applications Conference*, pages 117–126. IEEE, 2009.

[19] D.K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. *Synthesis Lectures on Distributed Computing Theory*, 1(1):1–137, 2010.

[20] E.A. Lee. Cyber-physical systems-are computing foundations adequate. In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*. Citeseer, 2006.

[21] R.P. Lippmann, K.W. Ingols, and MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB. *An annotated review of past papers on attack graphs*. Massachusetts Institute of Technology, Lincoln Laboratory, 2005.

[22] G. Louthan, W. Roberts, M. Butler, and J. Hale. The blunderdome: an offensive exercise for building network, systems, and web security awareness. In *Proceedings of the 3rd international conference on Cyber security experimentation and test*, pages 1–7. USENIX Association, 2010.

[23] J. Lygeros, K.H. Johansson, S. Sastry, and M. Egerstedt. On the existence of executions of hybrid automata. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, volume 3, pages 2249–2254. IEEE, 1999.

[24] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata revisited. *Hybrid Systems: Computation and Control*, pages 403–417, 2001.

[25] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid I/O automata. *Hybrid Systems III*, pages 496–510, 1996.

[26] C. Phillips and L.P. Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM, 1998.

[27] D.R. Raymond, R.C. Marchany, M.I. Brownfield, and S.F. Midkiff. Effects of denial-of-sleep attacks on wireless sensor network MAC protocols. *Vehicular Technology, IEEE Transactions on*, 58(1):367–380, 2009.

[28] B. Schneier. Modeling security threats. *Dr. Dobb's journal*, 24(12), 1999.

[29] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. 2002.

[30] S.J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM, 2001.

[31] T. Tidwell, R. Larson, K. Fitch, and J. Hale. Modeling internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and security*, volume 59, 2001.

[32] L. Wang, S. Noel, and S. Jajodia. Minimum-cost network hardening using attack graphs. *Computer Communications*, 29(18):3812–3824, 2006.