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

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

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

Directed by John C. Hale

96 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 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

For their continued support, I thank my family and Evan Mackay.

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 Kissinger in the whiteboard conversation that resulted in the concept of hybrid link automata. My further thanks to Chris Hartney, Zach Harbort, and Jordan Sanderson for their collaboration on a variety of attack graph related research and development.

I would also like to thank my advisor, Dr. John Hale, for his patience and guidance, as well as my other committee members..

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, of DARPA or the U.S. Government.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Introduction

As computer systems become pervasive, not only are their interactions with people becoming more frequent; they are also increasingly interacting with the physical world and with each other. These systems sometimes bridge the divide between the computational and the physical, blending discrete and continuous elements into a single system.

Such systems are termed *hybrid systems*. When linked together with a significant network component, these systems are sometimes called *cyber-physical systems*. They are pervasive in safety-critical applications such as medical, critical infrastructure, and automotive equipment. This thesis is concerned with modeling these systems, their security, and the manner in which their components interact 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:

> 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 computations 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 [22].

This still holds true today. Existing frameworks—abstractions—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, modeling methods from the world of isolated control systems cannot model the complex distributed networks that are the hallmark of cyber-physical systems.

1.3 Scope

This thesis presents an extension of the attack graph into the continuous domain. The goal is to incorporate aspects of hybrid systems modeling (e.g. hybrid automata), which best describe systems in relative isolation into attack graphs, which excel at capturing complex interrelationships and interdependencies among systems and attacks.

The remainder of this thesis is structured as follows. Chapter 2 provides background in hybrid systems and their modeling, introduces past work in attack graphs, and presents a set of case studies in both the hybrid and discrete domains used throughout this work. Chapter 3 contributes a specification for the generation process. Chapter 4 introduces in detail an expanded attack graph model to be used as the basis for the hybrid extensions. Chapter 5 introduces those 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, models, and examples involved in this thesis, including some insofar unpublished work that does not necessarily comprise a contribution of this thesis.

First, hybrid systems and their newer counterparts, cyber-physical systems, are addressed. Existing modeling frameworks for hybrid systems, particularly hybrid automata, are defined, and an introduction to attack graphs is provided along with a survey of their areas of research.

Next, section 2.4 presents at length the version of the attack graph model developed at the University of Tulsa and the components thereof. 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 abound in industrial controls, 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]. It does not suffice to describe precisely the types of systems with which this work is concerned: the subset of hybrid systems that incorporate significant computer and networking components.

Nevertheless, the modeling of hybrid systems is well studied and provides a sufficient body of relevant work from which to draw to warrant its inclusion. This chapter includes background on a particularly relevant 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 term for the systems investigated in this thesis is *cyber physical systems* (CPS). Simply, a cyber physical system is a networked hybrid system: that is, 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: their concentration in safety critical domains, their integration of unrelated systems, their dependence upon unreliable data collection, and their pervasiveness [12].

2.2.4 Hybrid Automata

Definition: A valuable formalism for modeling hybrid systems in isolation and with limited interconnection is the hybrid automaton (HA) of Alur, *et al.* [3]. This section introduces the version of the formalism described in 1996 by Henzinger [16], to which a reader interested in more than a superficial understanding is referred.

Formally, a hybrid automaton H is made up of a set X of real-valued state variables, a set $\dot{X}$ of their first derivatives, a set of operational modes and guarded switches between the modes, and predicates describing continuous evolution over time.

One can think of a hybrid automaton as a pairing of a finite state machine—whose states ("modes") and transitions ("switches") denote the discrete-domain behavior of the system—with differential equations that govern its continuous-domain behavior.

Modes may be decorated with *invariant* conditions (when the system may be in that mode), *flow* conditions (how the continuous state variables may evolve while in that mode), and *initial* conditions (when and if the automaton may start in that mode). Switches are decorated with *jump* conditions, which determine (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, given in Fig. 2.1, is a heater thermostat [16]. The vertices in the automaton represent its operating 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 $x < 18$). The switch from on to off behaves similarly.

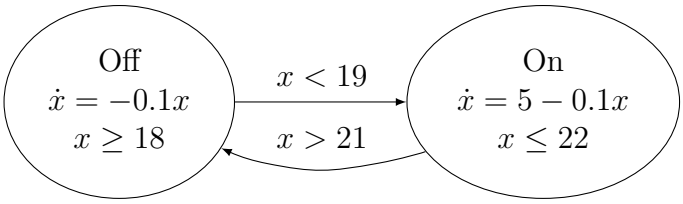

Figure 2.1: Thermostat hybrid automaton

Shortcomings: A hybrid automaton is not guaranteed to have a valid execution, and computing whether it does or not is non-trivial [26]. Model checking has been developed for only some subclasses of automata [17, 15], and many useful properties of them are undecidable [18].

However, there are further issues when considering hybrid automata for the study of cyber-physical systems. One of the hallmarks of cyber-physical systems is a distributed and highly networked nature. While HAs provide a natural model for the discrete-continuous boundary, they have only a rudimentary notion of communication, and when used in large topologies have significant scaling problems, both computationally and cognitively.

Alternatives: Attempts have been made to improve the hybrid automaton's network modeling capability. The designation of shared actions and shared variables as "input" or "output" is a popular tactic, as in the powerful hybrid I/O automaton [28, 27], its descendant the timed I/O automaton [21], and in the PHAVer model checker [15].

Similarly, the work of this thesis is partly an outgrowth of this strategy. An object example of that preliminary work's inherent issues is given in Fig. 2.2, a considered "hybrid link automaton" prototype. This models a link between two unpictured hybrid automata called S (the message source) and D (the message destination). The link is named λ, and the message is denoted with μ. The intent was that the automaton be parameterized, with many instances of it and similar link machines be used in the model of a single system. The link exhibits the following

behaviors:

Messaging The messages in the automaton are encoded as either "source : link : message" for a message incoming to the link from S, or "link : destination : message" for outgoing to D. These are actually hybrid automata events, which are used with its composition facility to model this messaging.

Unreliable delivery The unguarded switch from the transmit to idle mode permits transition back to idle without the actual emission of a message.

Message injection The unguarded switch from Idle to Transmit permits a message to transmit without being received.

Latency The clock ($C_{\lambda\mu}$, which increments at a rate $\dot{C}_{\lambda\mu}$) permits a certain waiting period between entry to the Transmit mode and actual transmission of the message.

Rudimentary mutual exclusion The Wait mode permits only a single message to be on the wire at a time.

This strategy may have a place in modeling some systems but falls short of the goal of modeling complex hybrid systems with more conventional computer networks—it is simply too low level.

Although the hybrid automaton is the gold standard for modeling hybrid systems, other frameworks do exist such as hybrid process algebras [13, 5], entirely symbolic systems 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 [10].

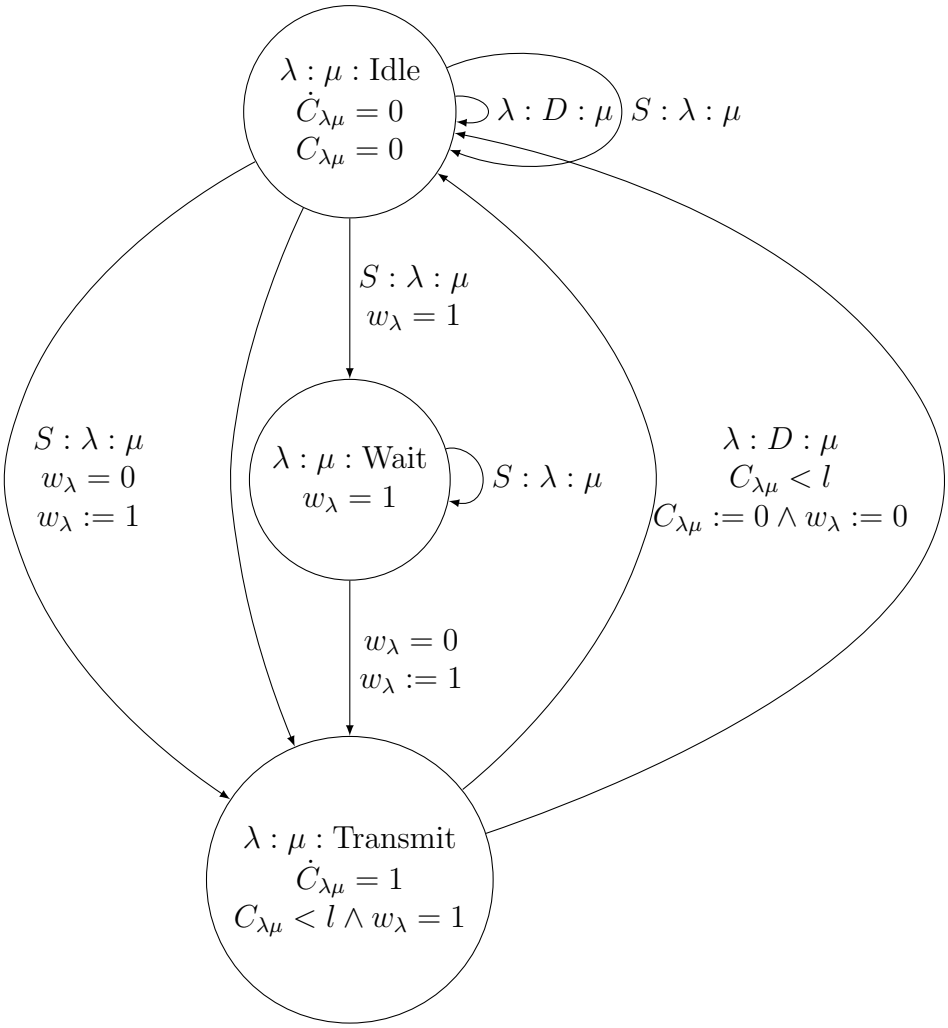

Figure 2.2: Example hybrid link automaton

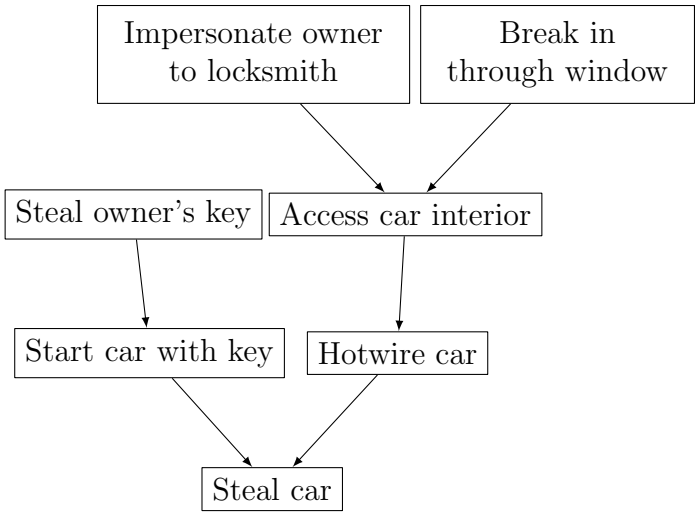

Figure 2.3: Simple car theft attack tree

2.3 Attack Trees and Graphs

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 with interacting elements. 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 model of an abuse of a system [32]. The root of the tree represents the attacker's goal, and the children of any given node represent its prerequisites. 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 hot wiring 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.

Attack trees are goal oriented: the consequences of the attack are known, and

the generation process is the enumeration of the means by which those consequences could be reached. It is, as an attack model, agnostic to the underlying system. This 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 approach 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 exploration of the state space of a system. They engage a graph theoretic model in which vertices represent individual system states, and edges represent state transitions caused by an adversary. The concept as originally introduced shares with all modern incarnations notions of attack patterns to be bound to state transitions; network elements and their individual configurations; and network topology. Also originally present but rarer in modern versions are representations of the attacker's capabilities and edge weights representing likelihood [30]. A similar structure called a privilege graph was introduced around the same time [14].

Most approaches to attack graph modeling represent exploits (attack patterns) using preconditions and postconditions [23, 35]. 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 stateful 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 computed.

Specification of the underlying network model may be done with only very loose restrictions, allowing arbitrary keywords as named qualities and topologies of network objects, as is favored for example in the work of George Mason University [4, 37]. This thesis employs this method, permitting more straightforward adaptation into the continuous domain.

The alternative is domain-driven, confining the modeler to terms that impose explicit computer networking concepts onto the model [35]. 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 [19].

Another variant of the attack graph, representing a fairly significant departure from the original formalism, is the exploit dependency graph (or attack dependency graph), which encodes similar information in a graph of conditions and exploits, rather than of states [20, 29, 24].

Research Directions: Research in attack graphs is spread throughout a variety of pathways. These include merely evaluating a network's security [4], specification of formal languages to represent attack graphs [35], intrusion detection system integration [36], automatic generation of security recommendations [37], and reachability analysis between hosts in a single network state [19]. For a thorough literature review up to 2005 and more detailed discussion of popular research directions, refer to the work of Lippmann and Ingols [23].

Attack graph work can be considered to fall into four broad categories, refer-

enced 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. The work that enables a representation of time is shared between generation and modeling. Constraints (such as monotonicity) on the progression of state transitions also fall under the generation category.

Analysis Analysis of attack graphs draws conclusions from attack graphs. Work here includes integration with intrusion detection systems, automatic delivery of mitigation recommendations, and the identification of states' security consequences.

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.

This thesis is mainly concerned with the modeling stage. Some amount of work is included in the generation stage, particularly dealing with the progression of time.

2.4 Attack Graph Generation and Modeling

This section presents a version of the attack graph modeling framework specific to traditional information systems. The framework has a very permissive modeling language 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 connect pairs of assets. Exploit patterns

have preconditions and postconditions on parameterized assets that can be bound at generation time to assets to create state transitions.

2.4.1 Intuitive Definition

An attack graph is comprised of the following components.

Assets Assets are the security principals in the attack graph formalism. For example, an asset may represent a host, an attacker, or a document. Assets are specified with unique names, and they are decorated with *qualities* and *topologies*.

Qualities Qualities represent properties of an asset, such as a software package or version, or whether it is offline, online, or in sleep mode. Qualities are key/value pairs, where both the key and the value are strings.

Topologies Topologies represent relationships between two assets. In addition to network connections, these include abstract relationships such as trust or access. Topologies are directed and named with strings. Together with qualities, topologies make up the network state's collection of facts, also called the "fact base".

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 general 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 parameters corresponding to assets, are guarded by preconditions, and specify a set of

$\mathcal{A}$: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)
$\text{Op} = \{\text{ins}, \text{del}\}$

Figure 2.4: Attack graph primitive domains

postconditions: insert and delete actions on qualities and topologies to update the fact base and therefore generate new network states.

2.4.2 Formal Definition

Primitive Domains: To clarify the attack graph formalism, this section provides a more formal definition of its components' domains. The primitive domains describe the atomic units of the formalism: assets, properties (qualities), values (qualities), relationships (topologies), vulnerabilities (exploit pattern identifiers), parameters (free asset variables in the attack patterns), and operations (used in postconditions representing insert or delete actions). See Fig. 2.4 for these elements' formal notation.

Compound Domains: Compound domains are composed of combinations of members of the primitive domains. These form the level of abstraction that it is most convenient to discuss intuitively.

Qualities Qualities bind an asset to a property to a value; therefore their domain is their Cartesian product. 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 is 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, members of $\mathcal{I}$ rather than $\mathcal{A}$. These are used in preconditions and, with operators, postconditions:

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

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

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

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

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

```
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

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 Specification Language

The attack graph generator does not use this formal notation to represent system elements, instead favoring a more user friendly specification language.

Network Model Specification: The first component of the specification itself is an asset list. Following the asset list, the initial fact base is specified as a semicolon separated list of qualities and topologies. An example network model specification is found in Fig. 2.5.

2.4.4 Exploit Specification

Exploit patterns resemble functions whose bodies are lists of preconditions,

```
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

which are identical in form to the facts in the network model (except that their assets are free rather than bound) and postconditions, which are simply deletions or insertions (with overwrite semantics) of facts. An example may be found in Fig. 2.6.

2.4.5 Generation Process

Attack graph generation is the process of chaining exploits to enumerate the attack space [9, 30, 33]. Modern methods for generating attack graphs share a common general architecture, 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.

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, it is given more detailed treatment in chapter 3.

2.5 Case Studies

Among the examples appearing in this work, two are based on specific real world scenarios. The first is an attack on a traditional information system, based upon an offensive educational exercise deployed at the University of Tulsa. The second is the denial of service through battery exhaustion of a simple cyber-physical system of

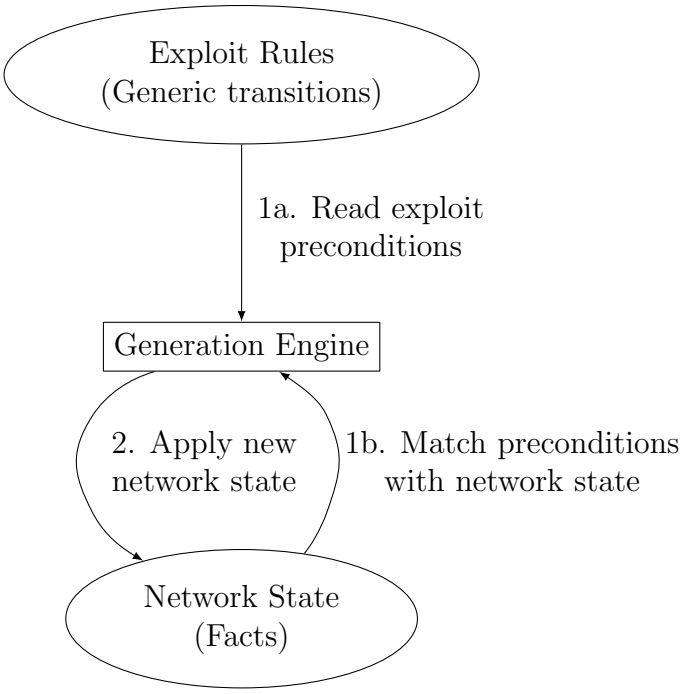

Figure 2.7: Attack graph generation process

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. 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 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, published in [25].

2.5.2 RFID Denial of Sleep

The second case study 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

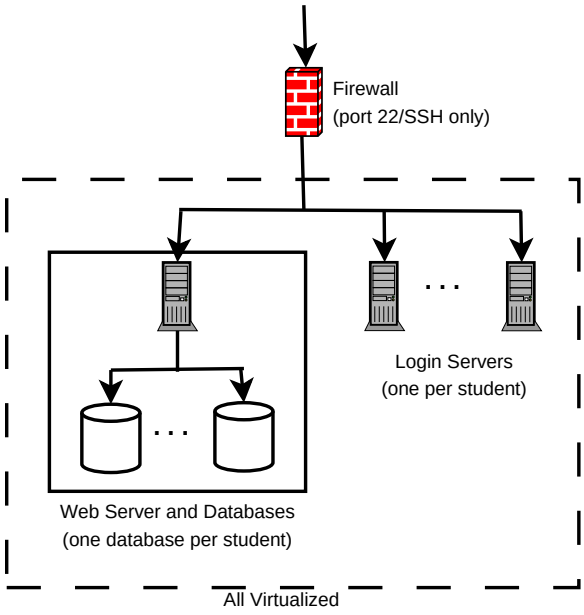

Figure 2.8: Blunderdome network architecture

Stage	Precondition	Attack	Postcondition
1	Given weak SSH public key	Find private key	Login server user access
2	User-level access	`vmsplice` privilege escalation	Root on login server; web server credentials
3	Web application access	SQL injection	Altered grade

Table 2.1: Stages of the Blunderdome attack

freight tracking and the Department of Energy for tracking spent fuel containers [11]. The attack is similar to the ones described by Buennemeyer, *et al.* [7], and is of a newly distinguished class of attacks sometimes termed *denial of sleep attacks* [6] [31].

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 safety 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 causes them to sleep unless the timer is reset by a valid command from the reader or a wake-up signal. In sleep mode, the tags can only be awoken by a wake-up command. 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 for a second, rogue RFID reader to be placed by the attacker within range of 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.

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. Under ordinary conditions this might take place over the course of several years before the batteries in the tags are drained [11].

Fig. 2.10 depicts a model of a tag. It has two state variables: c, which represents the active mode timeout clock, and B, which represents battery capacity. When active,

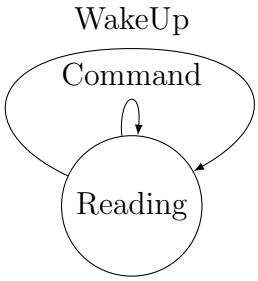

Figure 2.9: Hybrid automaton model of the RFID reader

the battery drains at the arbitrarily chosen rate of -50 units per second. 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.

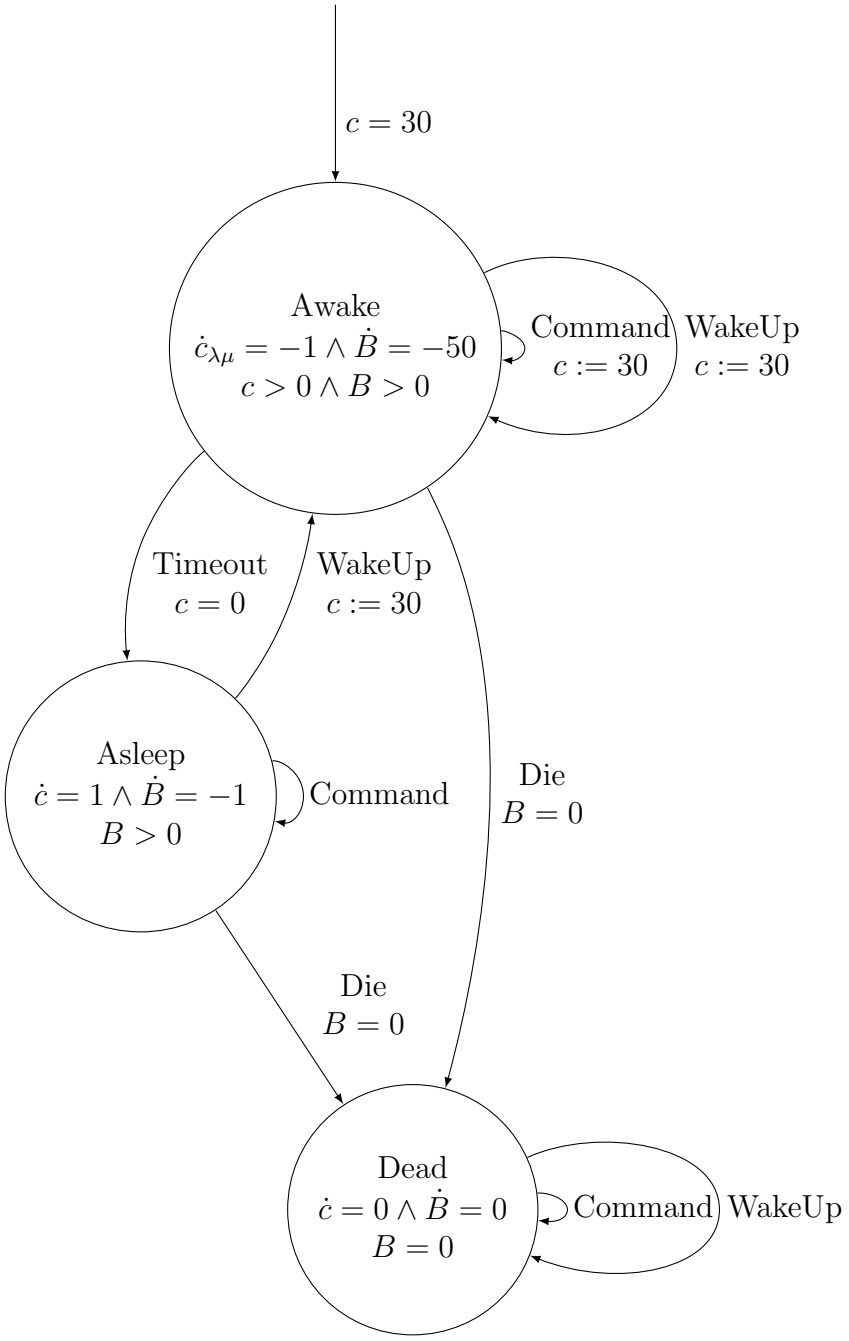

Figure 2.10: Hybrid automaton model of the case study active RFID tags

CHAPTER 3

ATTACK GRAPH GENERATION

3.1 Introduction

Attack graph generation is a research area all to itself, involving trade offs in performance, storage, time, and expressiveness. 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; the contribution lies in articulating such an algorithm explicitly.

3.2 Generation Algorithm and Pseudocode

3.2.1 Description

The input to this process is a network model specification (assets and initial fact base), exploit pattern specification, and maximum allowed depth (actually maximum allowed shortest path from starting state) of the constructed attack graph.

Attack graph generation proceeds in the following stages, separated by section below. In sections where it is applicable, pseudocode is provided. As the reference

```
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

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 structure 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 type is shown in Fig. 3.2.

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

```
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

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 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,

```
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

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 assets and facts. 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 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.

```
def get_successor_state(network_state, attack):
 successor_assets = deep copy of network_state.assets
 successor_facts = deep copy of network_state.factbase

 exploit = attack[0] # (exploit, binding map)
 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,
        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_bindings,
                              attack_graph)
```

Figure 3.6: Attack graph generation pseudocode

```
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

3.3 Example

In order to aid understanding the generation process, this section demonstrates 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 uses 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 is 2 for the purposes of this exercise, and by creating an initial list of states for analysis, which contains only 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

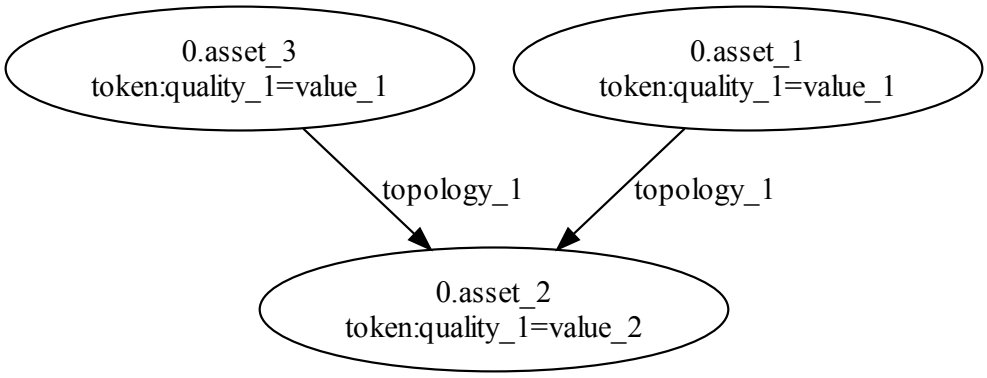

Figure 3.9: State 0 of the illustrative example

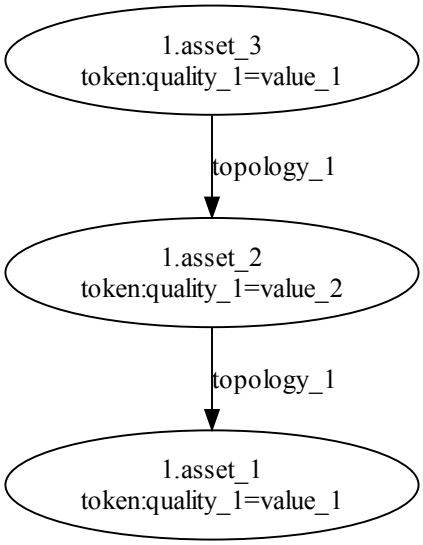

Figure 3.10: State 1 of the illustrative example

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. 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 (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.

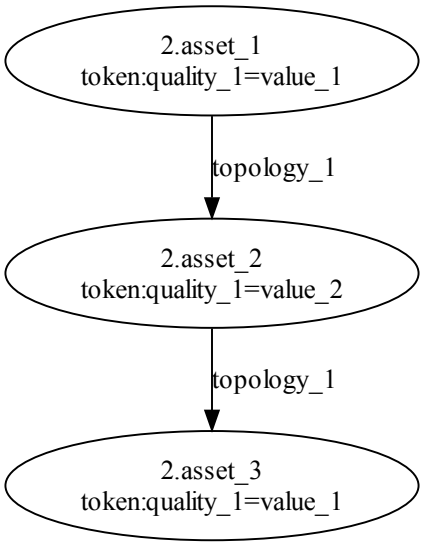

Figure 3.11: State 2 of the illustrative example

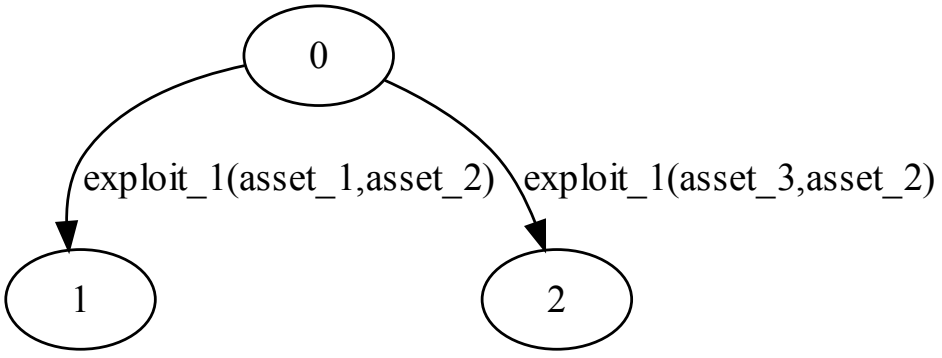

Figure 3.12: Attack graph after first iteration

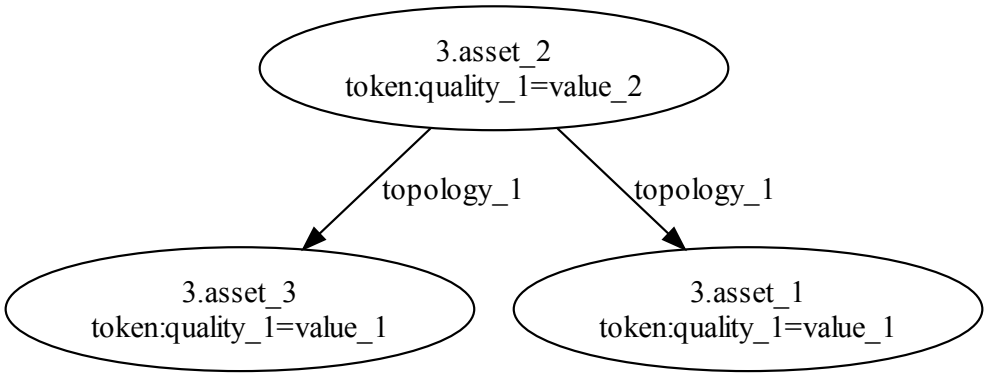

Figure 3.13: State 3 of the illustrative example

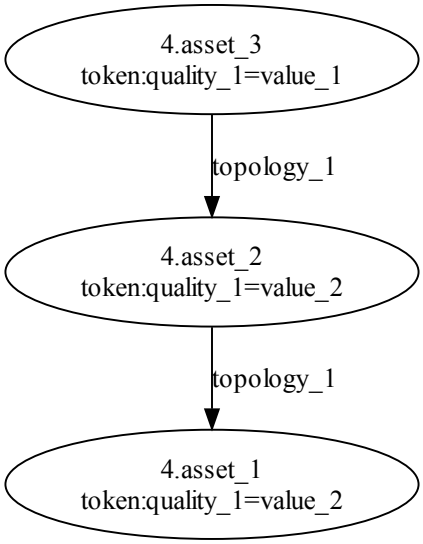

Figure 3.14: State 4 of the illustrative example

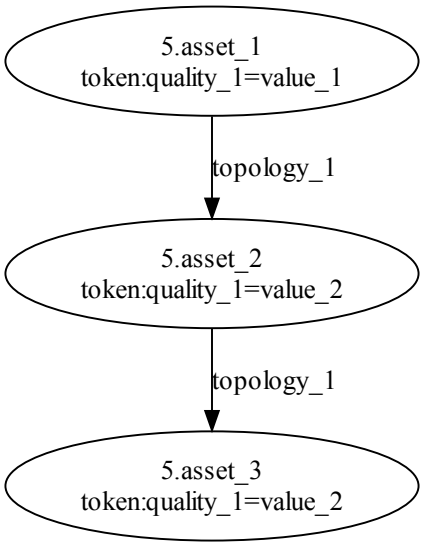

Figure 3.15: State 5 of the illustrative example

Generation continues by addressing State 2. In State 2, two possible attacks are returned: `exploit_1` on `(asset_1, asset_2)` and `exploit_2` on `(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 contents of the successor state list become the new analysis states, 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 model. The chapters that follow are devoted to this framework's expansion and analysis.

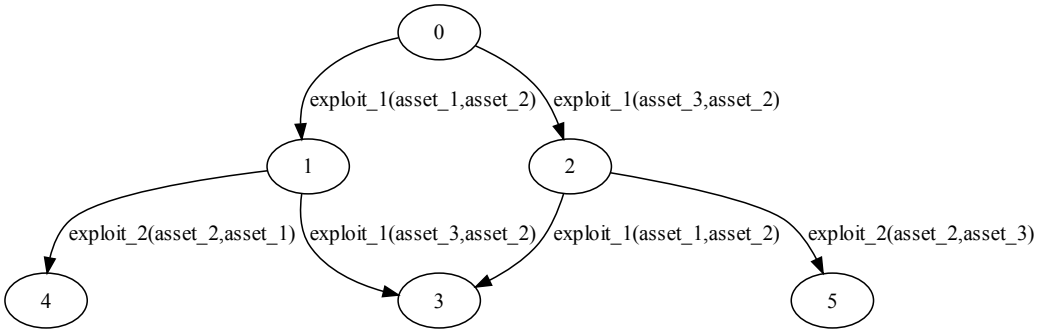

Figure 3.16: Attack graph after first iteration

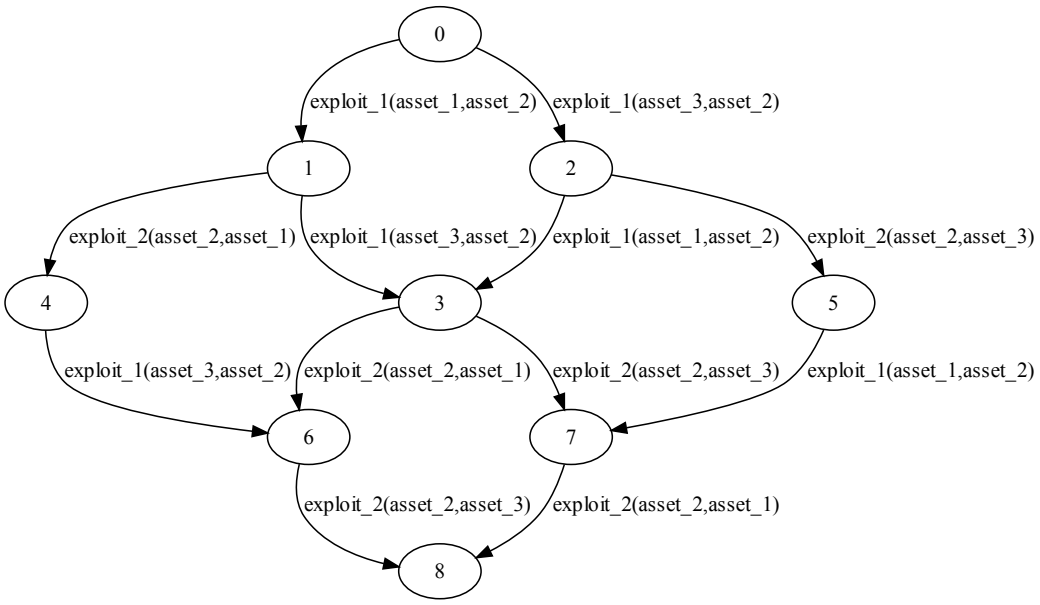

Figure 3.17: Complete illustrative attack graph

CHAPTER 4

DISCRETE EXTENSIONS

4.1 Introduction

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 to 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 [2]; some syntactic changes to ease the modeling of information systems; mechanisms for grouping attacks; and a scheme for integrating with the Common Platform Enumeration [8].

4.2 Working Lexicon

4.2.1 Introduction

Because of the unrestricted nature of the attack graph specification language, some conventions must be established in order to proceed systematically. This thesis recommends a set of conventions designed to ease 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).

4.2.2 National Vulnerability Database

Two vulnerability properties catalogued by the NVD inform this lexicon: *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.

Confidentiality Confidentiality impacts allow the unauthorized disclosure of information

Integrity Allows modification of data

Availability Availability impacts correspond to disruptions 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. It has three subcategories:

User access The attacker gains user level access to the operating system.

Administrative access The attacker gains root level access to the operating system.

Other access The attacker gains any other type of access (e.g. to a service account).

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 model the network 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 qualities. 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 an element of the system. Its properties, connections, and access levels are modeled as qualities and topologies of the adversary asset. This strategy, which is employed in this thesis and lexicon, can model multiple adversaries with varying capabilities in a single model. Aside from that difference, they are roughly equivalent, and their selection is a matter of taste.

This section introduces two types of topologies: connection and access, corresponding respectively to access vector and security protection impact subtype.

Connection Topologies: Connection topologies refer to how two assets are connected, over the network or otherwise. As with NVD access vectors, 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, protocols and services are specified explicitly and individually 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 the "client", and the destination is the "server", where such distinctions are meaningful.

Access Topologies: Access topologies refer to trust relationships and distinguish what kind of access one asset has to another. Much like the subcategories of the Security Protection impact type, there are three basic types of access topology: user access, root access, and other access.

They are named using the word `access`, an underscore, and the lowercase name of the access type. If the access type is `other`, an optional underscore delimited description of the access type (e.g. the name of the service). Examples include `access_user`, `access_root`, `access_other_apache`, or `access_other_vsftp`.

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 standard quality structure 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` with the value `up` or `down`. This allows exploits to require that an asset be online as a precondition or to implement denial of service attacks with `status = down` quality postconditions.

4.3 Syntactic Sugar

4.3.1 Directional Topologies

This section introduces new notation for 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`, a colon, and asset names; however, in the new syntax, they are separated by a directionality symbol: `->` to represent a one-way topology or `<->` to represent a two-way topology. To promote readability, `<-` is not recognized.

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. If a bidirectional topology exists in the fact base, one of its component directional topologies may be removed without affecting its reverse.

4.3.2 Value Assignment

In preparation for the introduction of real-valued facts, the existing discrete qualities, henceforth called "token valued", are given a special assignment operator. Simply put, token values are both assigned and tested for equality using the `=` operator; they may also be tested for inequality with the `!=` operator.

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 Preprocessing

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 down with the 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.

4.4 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 multiple exploits. 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 string argument, the group name.

Their 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; naming the parameters the same is highly recommended but not strictly required.

Two *caveats* are required with global and grouped exploits. The first is that a group may not mix global and nonglobal exploits. If mixing of nonglobals and globals were permitted within a single group, the result would be behavior that is compatible with neither the concept of a group or the concept of a global (a group requires that each single attack binding be made to all exploits within the group, whereas a global requires every possible attack binding be made simultaneously).

The second warning is that no consistency checking is specified. That is, preconditions are checked against a predecessor state, then postconditions are applied in arbitrary order. This means that if multiple attacks in a group modify the same facts on the same assets, indeterminate behavior or race conditions may occur. For this reason, it is advised to group only exploits that whose preconditions guard against this possibility.

4.5 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) [8], a component of the Security Content Automation Protocol (SCAP). They can specify operating systems, applications, and hardware.

A CPE platform fact is simply a valid CPE URI tied to an asset. The CPE URI takes the following form, all of which is optional.

```
cpe:/part:vendor:product:version:update:edition:language
```

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 a prefix property.

4.6 Generation process changes

These new capabilities require some adjustments to the semantic behavior of

the attack graph generation generation process. Happily, the attack graph generation algorithm itself is unchanged, although more sophisticated fact handling, both for precondition processing and postcondition application, is required. This section discusses these changes.

4.6.1 Bidirectional topologies

The addition of bidirectional topologies introduces a new requirement for parsing or precondition/postcondition analysis: that single bidirectional topology facts be decomposed into pairs of unidirectional topologies. In the reference implementation, this is done not in the initial parse itself but rather in the portion of fact handling that converts the raw parsed data structure into the generator's internal fact data structure.

4.6.2 Platform properties

The new platform properties introduce another set of conditional logic to precondition processing. Since there are no operators, precondition matching is relatively simple syntactically; however, because of the prefix property of its encoding and the ability to have blank "wildcard" fields in the matched CPE, a matching algorithm must be used. The matching algorithm in the reference implementation is based on the one provided in the CPE specification [8]. The pseudocode for this thesis's platform fact matching functionality is provided in Fig. 4.1

4.6.3 Precondition matching

On the other hand, quality precondition matching must become somewhat more complicated with the addition of the not-equal relational operator. Merely building a fact and checking for its membership in the fact base data structure is no longer sufficient.

Although the not-equal relational operator *could* be implemented similarly to

```
def has_platform(factbase_platformlist, platform):
 # Platform is a tuple
 for cpe in factbase_platformlist:
  if len(cpe) >= len(platform):
      ret = False
      for components in zip(cpe, platform):
       if components[0] == components[1] or platform == '':
        ret = True
       else:
        ret = False
        break
       if ret:
        return cpe # Return the matched platform
 return False
```

Figure 4.1: Platform fact matching pseudocode

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

type network_state:
    assets : set of strings;
    factbase : set of fact-tuples;
    qualities : map (keys=assets, vals=map(keys=quality_name,
                                           vals=quality_value))
```

Figure 4.2: Updated discrete network state data type pseudocode

the equality relational operator, by building a fact from the given name and value and checking for its exclusion from (instead of inclusion in) the fact base, a more advanced fact checking method is provided in order to ease the transition to permitting real values for qualities.

The fact base data structure is updated to include maps from quality names to quality values in order to facilitate lookup. Although this redundancy results in an increased storage requirement, it prevents costly lookups in the existing fact base. Fact checking, then, is done using these simple hash table lookups and comparisons. A new specification for the network state data structure reflecting these changes is provided in Fig. 4.2.

4.6.4 Global and grouped exploits

The simultaneous application of global and grouped exploits is also fairly straightforward. Postcondition processing requires only the minor change of permitting a list of postconditions to be processed sequentially (which really only requires the union of their postcondition facts) using the existing postcondition application method.

Precondition processing is more complex insofar as it causes the selection and grouping of attacks that should be fed into the generation function, each producing its own state transition. To permit a reader to follow along, pseudocode is provided in Fig. 4.3.

The reference implementation uses a two-pass algorithm over the set of valid attacks (selected using the existing precondition processing method). First, a trio of mappings are instantiated to store the attack data structures. The "grouped global" map is keyed on group names and has as values a list of every valid binding to all the attacks in the group. The "non-global group" mapping is also keyed on group names and has as values more maps, which are themselves keyed to tuples of assets (representing bindings values) and valued with specific attack bindings. The "non-grouped global" mapping is keyed on exploit name and also valued with attack bindings.

The first step is to loop through all the attacks. Attacks are placed in the appropriate map (or maps) depending on their group (and binding) and global configuration.

The second step is to use these maps to generate groupings of attacks, which are to be applied by the postconditions processor in the same manner as single attacks themselves are applied. First, each global group is inserted as its own aggregated attack group. Second, groups and bindings are looped through. An attack group is generated per binding per group name. Finally, each remaining (that is, non-grouped) global exploit is added as its own attack grouping. The collection of groupings is

returned.

Note that, in spite of the nested for loops, each possible attack binding is still processed at most twice.

4.7 Examples

4.7.1 Illustrative

For completeness, this section provides the updated syntax's version of Chapter 3's example, which is otherwise unchanged. The network model is provided in Fig. 4.4, and the exploit patterns are provided in Fig. 4.5.

4.7.2 Blunderdome

This section presents an attack graph model of the Blunderdome exercise described in section 2.5.1.

Network Model: The network model for the Blunderdome exercise follows in a straightforward manner from the network diagram (Fig. 2.8), including each network element as an asset, plus an asset for the attacker. The facts include the attacker's grade on the web server, the SSH and HTTP connection topologies, and platform facts for the relevant applications and operating systems subject to exploitation: OpenSSL, Linux, and the Blunderdome specific grades tracking application (which receives a dummy CPE entry).

In this model there are three access topologies in play: access_admin, access_user, and access_other_blunderdome, representing access to the Blunderdome web application.

Exploit Patterns: The exploits in the Blunderdome exercise, provided in Fig. 4.7 fall into three categories. The first is the set of straightforward CVE-based exploits that could be extracted automatically from the National Vulnerability Database. The

```
globl_group_dict = map keyed on group name
group_dict = map keyed on group name
globl_dict = map keyed on exploit name

for attack in attacks:
    group = attack.group
    globl = attack.globl
    binding = attack.binding.values # asset list

    if group and globl:
        if group in globl_group_dict:
            globl_group_dict[group].append(attack)
        else:
            globl_group_dict[group] = [attack,]
        continue

    if group:
        if group in group_dict and binding in group_dict[group]:
            group_dict[group][binding].append(attack)
        elif group in group_dict:
            group_dict[group][binding] = [attack,]
        else:
            group_dict[group] = new map{binding : [attack,]}
        continue

    if globl:
        if attack.name in globl_dict:
            globl_dict[attack.name].append(attack)
        else:
            globl_dict[attack.name] = [attack,]

agg_attacks = [] # Aggregated attacks, the goal of this code

for group_id in globl_group_dict: # Global groups:
    group_attacks = globl_group_dict[group_id]
    agg_attacks.append(globl_group_dict[group_id])

for group_id in group_dict: # Non-global groups:
    group_attacks = group_dict[group_id]
    for binding in group_attacks:
        agg_attacks.append(group_attacks[binding])

for globl_attack in globl_dict: # Non-grouped globals
    agg_attacks.append(list(globl_dict[globl_attack]))
```

Figure 4.3: Group and global attack selection

```
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 4.4: "Illustrative example" network model in the new format

```
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 4.5: "Illustrative example" exploit patterns in the new format

```
network model =
 assets :
  attacker;
  login_server;
  web_server;

 facts :
  quality:web_server,grade=F;
  topology:attacker->login_server,connected_network_ssh;
  topology:login_server->web_server,connected_network_http;
  platform:login_server,cpe:/a:openssl_project:openssl:0.9.8c-1;
  platform:login_server,cpe:/o:linux:kernel:2.6.24;
  platform:web_server,cpe:/a:isec:blundergrades;
.
```

Figure 4.6: Blunderdome network model

second is the set of hand generated exploits like blunder_sqli which has custom behavior to modify the attacker's grade. The third is the set of exploit patterns that do not necessarily represent abuse cases but rather state transitions due to attacker actions.

The first two exploits could be automatically generated from the CVE entries in the NVD. The third was hand generated due to its being an internal custom piece of software without NVD entries and the complex behavior the exploit must encode.

The next exploit, ssh_http_tunnel, represents a common idiom involved in attack chaining: using access to one machine to "pivot" and create new access topologies to adjacent machines. The last exploit, blunder_login, represents access to the Blunderdome credentials due to administrative access to the login server.

The resulting attack graph is provided in Fig. 4.8; the graphical representation of the starting state and ending state in Fig. 4.9 and Fig. 4.10.

```
exploit CVE_2008_0166_1(a, l)=
 preconditions:
  platform:l,cpe:/a:openssl_project:openssl:0.9.8c-1;
  topology:a->l,connected_network_ssh;
 postconditions:
  insert topology:a->l,access_user;
.

exploit CVE_2008_0600_1(a,l)=
 preconditions:
  topology:a->l,access_user;
  platform:l,cpe:/o:linux:kernel:2.6.24;
 postconditions:
  insert topology:a->l,access_admin;
.

exploit blunder_sqli(a,w)=
 preconditions:
  platform:w,cpe:/a:isec:blundergrades;
  topology:a->w,access_other_blunderdome;
 postconditions:
  insert quality:w,grade=A;
.

exploit ssh_http_tunnel(a,l,w)=
 preconditions:
  topology:a->l,connected_network_ssh;
  topology:a->l,access_user;
  topology:l->w,connected_network_http;
 postconditions:
  insert topology:a->w,connected_network_http;
.

exploit blunder_login(a,l,w)=
 preconditions:
  topology:a->w,connected_network_http;
  topology:a->l,access_admin;
 postconditions:
  insert topology:a->w,access_other_blunderdome;
.
```

Figure 4.7: Blunderdome exploit patterns

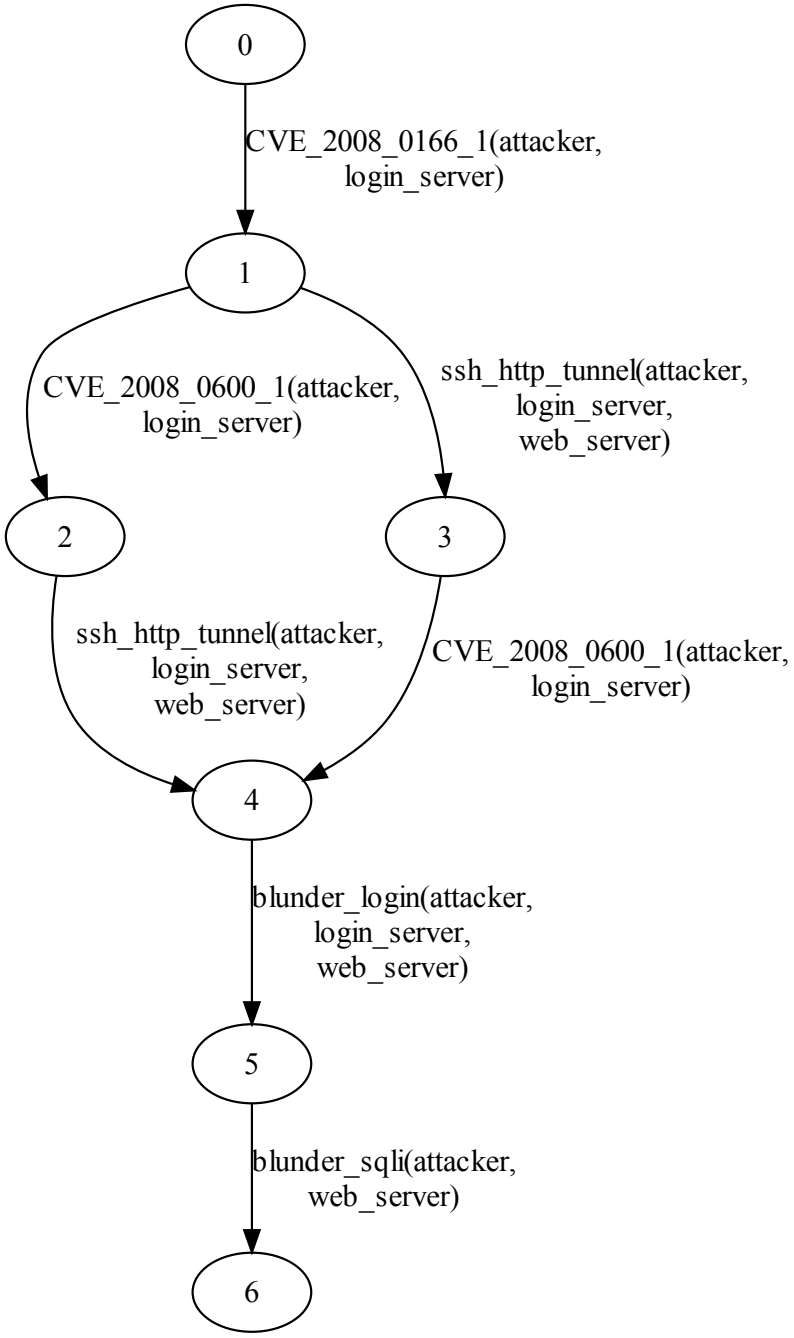

Figure 4.8: Blunderdome exercise attack graph

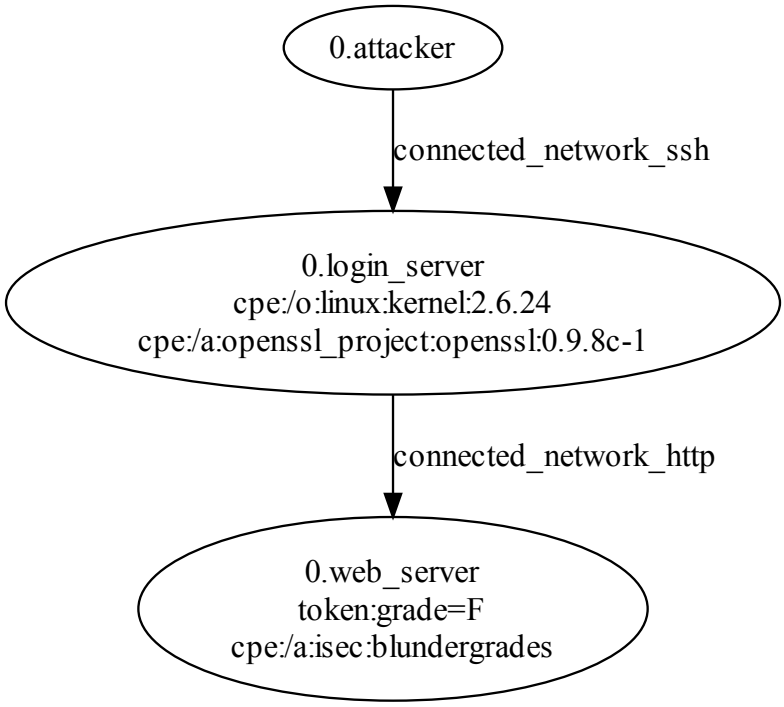

Figure 4.9: Blunderdome exercise starting state

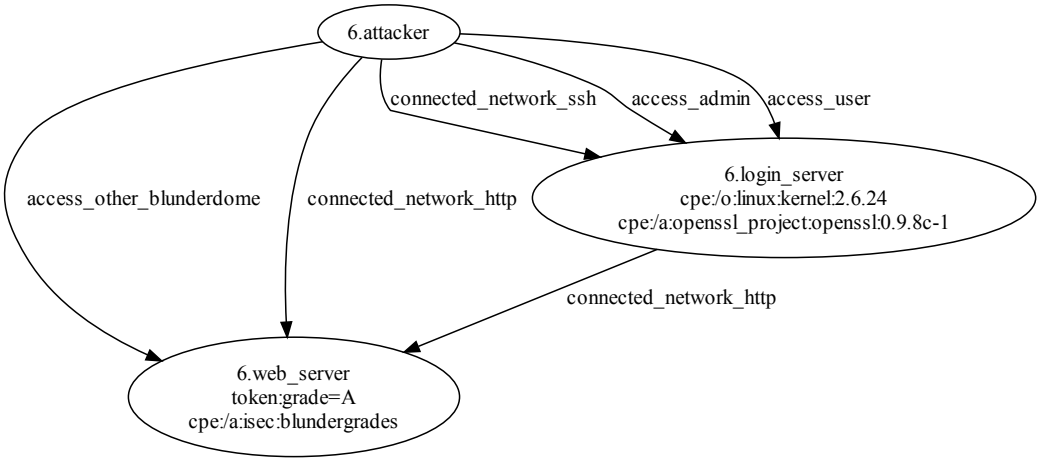

Figure 4.10: Blunderdome exercise ending state

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.

The ultimate goal of the specification of a hybrid attack graph for cyber physical systems modeling must be the creation of a formalism capable of modeling a network containing both traditional information systems (as existing attack graph incarnations can) and hybrid systems (i.e. those that can be modeled as hybrid automata). Therefore, the ideal hybrid attack graph should be able to define systems (assets) that have some equivalence to a hybrid automaton.

This thesis presents two novel attack graph elements with this requirement in mind. The first is the ability to give both qualities and topologies real (continuous domain) values, and correspondingly to permit exploits to test and operate on these values in the expected ways with a full selection of real-valued relational and assignment operators. These new types of facts are at the heart of this new model.

The second contribution is modeling the progression of time; as implemented in this thesis, time progression does not require any new syntactic elements except those provided in the preceding chapter; those syntactic elements, however, are used in a novel fashion to specify time evolution of facts.

The remainder of this chapter is structured as follows. First, the new syntactic elements are introduced. Second, the semantic changes that these new elements impose upon the generation process are described. Then the convention for the specification of time evolution is described. Finally, a pair of examples is provided.

5.2 Definition of New Syntax

5.2.1 Terminology

First and foremost, a discussion of the new terminology involved in the transition to hybrid attack graphs is required. This is mostly concerned with facts, as the transition to hybrid attack graphs consists mostly of changes to the fact system.

Until now, qualities have had only string type values, and topologies have had no values at all. In this chapter, that changes. Although there will always remain a need for these discrete valued qualities and name only topologies, there is also a need for qualities with continuous, numerical values. The introduction of a second type of quality value is somewhat less jarring, however, than that of topologies with values.

As the hybrid attack graph introduced by this thesis is a strict superset of the discrete attack graph described in the previous chapters, the existing style of quality and topology facts remains valid in the new language and therefore must be distinguished from the new type of fact in some way. The discrete only facts, therefore, are called *token valued* (or *token facts*), whereas the new continuous facts are called *real valued* (or *real facts*).

5.2.2 Operators

At the heart of the hybrid attack graph lies the new real facts. Their existence marks the introduction of a true type system to the attack graph model, which must be dealt with in some fashion. Furthermore, these new real facts demand new real operators to deal with them.

Typing is dealt with by ensuring that the sets of both assignment and relational operators used for token and real facts are disjoint. While token qualities are assigned with the `=` operator, real facts (both quality and topology) are assigned using a `:=` operator.

As for the relational operators, token facts are tested using `=` and `!=`. The relational operators permitted for real facts (both quality and topology) are different: `==`, `>`, `>=`, `<=`, and `<>` (for not equal).

Finally, because a principal property of real values is their suitability for arithmetic, a number of new operators are introduced for use in exploit postconditions only, with their straightforward C-like meanings: `:=`, `+=`, `-=`, `*=`, and `/=`. Note that currently these operators take only literal reals as their second operand; variables are not permitted.

5.2.3 Topology values

Due to the usefulness of permitting real valued relationships between assets, be they physical such as distance, or IT related like latency or signal strength, in hybrid attack graphs, real topologies may take values. The syntax for this is identical to the existing token topology syntax, except followed by an operator and a number.

5.2.4 The update operation

Because of the semantics of binary arithmetic operators such as `+=`, the term `insert` in postcondition operations is somewhat psychologically unsatisfactory. Therefore, it is now aliased to `update`, a semantically identical operation. In spite of its apparent redundancy, it provides for future expansion as well as readability.

5.3 Time

A fundamental requirement of modeling hybrid systems is the progression of time, as the most interesting and distinctive properties of physical processes are their

```
network model=
    assets:
        civic;
        wall;
    facts:
        platform:civic,cpe:/h:honda:civic;
        quality:civic,compromised=true;
        quality:civic,status=up;

        platform:wall,cpe:/h::wall;

        topology:civic<->wall,distance:=50;
.
```

Figure 5.1: Car example network model

evolution over time. Another goal is to avoid introducing additional special cases. To that end, this section introduces a method for handling time using only previously introduced attack graph functionality.

5.3.1 Time exploits

Exploits are not just for adversary actions in hybrid attack graphs. Time is implemented using a single group of global exploits, each of which increments a class of assets' position in time depending upon its particular state. Their globalness triggers on all assets simultaneously, while the grouping causes all the time exploits to trigger in concert. As before, an important requirement is that the sets of facts affected by time exploits be disjoint over all grouped time exploits to avoid indeterminate behavior.

A selection of time exploits is provided in Fig. 5.2 that might, when paired with the network model in Fig. 5.1, simplistically model a car driving away from a wall unless somehow compromised by an attacker, who causes the car to drive toward the wall at a constant rate.

This method of modeling time requires time to be quantized and time stepping behavior to be used.

```
global group(time) exploit car_depart(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised != true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
    postconditions:
        update topology:c<->w,distance+=25;
.

global group(time) exploit car_approach(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised=true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
        topology:c<->w,distance>25;
    postconditions:
        update topology:c<->w,distance-=25;
.

global group(time) exploit car_crash(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised=true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
        topology:c<->w,distance<=25;
    postconditions:
        update topology:c<->w,distance:=0;
        update quality:c,status=down;
.
```

Figure 5.2: Car example exploit patterns (note time)

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

type network_state:
    assets : set of strings;
    factbase : set of fact-tuples;
    qualities : map (keys=assets,
                     vals=map(keys=quality_name,
                              vals=quality_value))
    topologies : map (keys=src_assets,
                      vals=map(keys=dest_asset,
                               vals=map(keys=topology,
                                        vals=values))
```

Figure 5.3: Updated hybrid network state data type pseudocode

5.4 Generation process changes

5.4.1 Topologies

Even without concern for real types, the move to hybrid attack graphs introduces a new, more basic requirement for the processing of preconditions and postconditions: topology values. The topology fact handling is updated to be virtually identical to the handling of qualities. In fact, in the hybrid attack graph reference implementation, every topology, even token topologies, has a value: real topologies have floating-point values, and token topologies have boolean values.

This results in the new fact base data structure provided in Fig. 5.3. The intermediate hash map representation for topology facts is keyed on source asset names, with values that are themselves hash maps keyed on destination asset names, with values that are *also* hash maps keyed on topology names, with values storing topology values. Again, for token topologies, this value is always the boolean true.

5.4.2 Matching

With the introduction of new relational operators, more advanced precondition processing is required. Happily, the changes introduced in the previous chapter

anticipated this, so the heavy lifting required for fact value lookup already exists. All that remains is the mapping of the parsed attack graph operators onto implementation operators in the generation software.

As illustrated in the pseudocode in Fig. 5.4, this is done in the reference implementation by maintaining a map from the string representations of the relational operators onto (pointers to) boolean functions that apply them. The new relational operator precondition processing does not impact the remainder of the generation process.

5.4.3 Updating

The hybrid attack graph includes assignment operators that require new post-condition handling that closely parallels their precondition handling. The same issues are addressed here; value querying and setting is already handled, but operator selection is not. The pseudocode for this process is presented in Fig. 5.5.

5.5 Examples

5.5.1 Car

A useful preliminary example is the scenario involving a car as described in Figs. 5.2 and 5.1. The corresponding attack graph for that scenario is given in Fig. 5.6. As the car is compromised from the start, it only has three states. Note the exploit transitions, both of which are collected into a group called "time", signifying that they represent time steps. The global and grouping mechanisms, however, only truly become dramatic once combined with discrete exploits causing mode transitions, and even more so once multiple assets are involved.

Suppose the initial conditions were changed so that the `compromised` quality began as `false`, and an additional exploit called `own_civic` were added so as to match the patterns given in Fig. 5.7.

```
RELOPS = map(keys = ('==', '<>', '>=', '<=',
                     '<', '>', '=', '!='),
             vals = (associated functions))

def matches_topology(source, dest, name, value=True,
                     op='==', check_reverse=False):
    if source in factbase.topologies and
      dest in factbase.topologies[source] and
      name in factbase.topologies[source][dest]:
        my_value = factbase.topologies[source][dest][name]
    else:
        return False

    if type(my_value) != type(value):
        Error: Cannot match token values with real values.

    if check_reverse:
        return RELOPS[op](my_value, value) and
               matches_topology(dest, source, name,
                                value=value, op=op)
    else:
        return RELOPS[op](my_value, value)

def matches_quality(asset, name, value, op='=='):
    if asset in factbase.qualities and
      name in factbase.qualities[asset]:
        my_value = factbase.qualities[asset][name]
    else:
        return False

    if type(my_value) != type(value):
        Error: Cannot match token values with real values.

    return RELOPS[op](my_value, value)
```

Figure 5.4: Hybrid precondition matching pseudocode

```
ASSIGNOPS = map(keys=('+=', '-=', '*=','/=', ':=', '='),
                vals=(associated functions)

def set_quality(factbase, asset, name, value, op='='):
    if asset in factbase.qualities and
      name in factbase.qualities[asset]:
        my_value = factbase.qualities[asset][name]
    else:
        my_value = 0
    new_value = ASSIGNOPS[op](my_value, value)
    update factbase with:
        factbase.qualities[asset][name] = new_value

def set_topology(factbase, source, dest, name,
                 value=True, op=None):
    if not op: # Token
        update factbase with:
            factbase.topologies[source][dest][name] = value
    else: # Real
        if source in factbase.topologies and
          dest in factbase.topologies[source] and
          name in factbase.topologies[source][dest]:
            my_value = self.assets[source].get_topology(
                dest, name
              )
        else:
            my_value = 0
        new_value = ASSIGNOPS[op](my_value, new_value)
        update factbase with:
        factbase.qualities[source][dest][name] = new_value
```

Figure 5.5: Hybrid postcondition application pseudocode

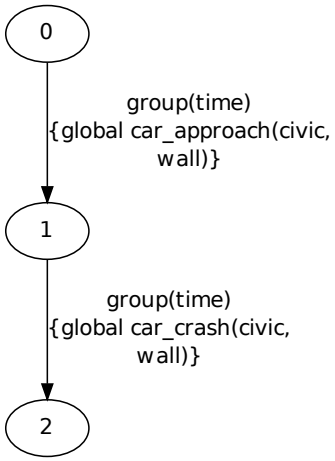

Figure 5.6: Simple car attack graph

In this case, the resulting attack graph (shown in Fig. 5.8 to a maximum generation depth of 5) clearly illustrates the interaction of the discrete behavior of the attacker (an attack occurring in the "cyber" world) with the continuous evolution of the car (in the physical world). This embodies a common trait of hybrid systems and also hybrid attacks: their discrete behavior acts to place them into operational modes wherein the passage of time causes important behavior. In this case, the evolution of time is clearly denoted by the transitions labeled `group(time)`.

One final automobile example serves to illustrate the behavior of the combined global and grouped exploits and their use in implementing timing behavior. Consider the case that, while keeping the exploits as in Fig. 5.7, introduces an additional car that drives away from the wall at the same rate as `civic` but is *not* vulnerable to the `own_civic` exploit. This network model is provided in Fig. 5.9.

The resulting attack graph is provided in Fig. 5.10. Observe the sometimes heterogeneity of the time evolution exploit groups. Of particular interest is the transition from state 3 to state 6, in which the crashing of the Civic has the discrete consequence of placing it into a mode in which it no longer changes state with time.

```
global group(time) exploit car_depart(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised != true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
    postconditions:
        update topology:c<->w,distance+=25;
.

global group(time) exploit car_approach(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised=true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
        topology:c<->w,distance>25;
    postconditions:
        update topology:c<->w,distance-=25;
.

global group(time) exploit car_crash(c,w)=
    preconditions:
        platform:c,cpe:/h:honda;
        quality:c,compromised=true;
        platform:w,cpe:/h::wall;
        quality:c,status=up;
        topology:c<->w,distance<=25;
    postconditions:
        update topology:c<->w,distance:=0;
        update quality:c,status=down;
.

exploit own_civic(c)=
    preconditions:
        platform:c,cpe:/h:honda:civic;
        quality:c,compromised=false;
        quality:c,status=up;
    postconditions:
        update quality:c,compromised=true;
.
```

Figure 5.7: One-car hybrid example

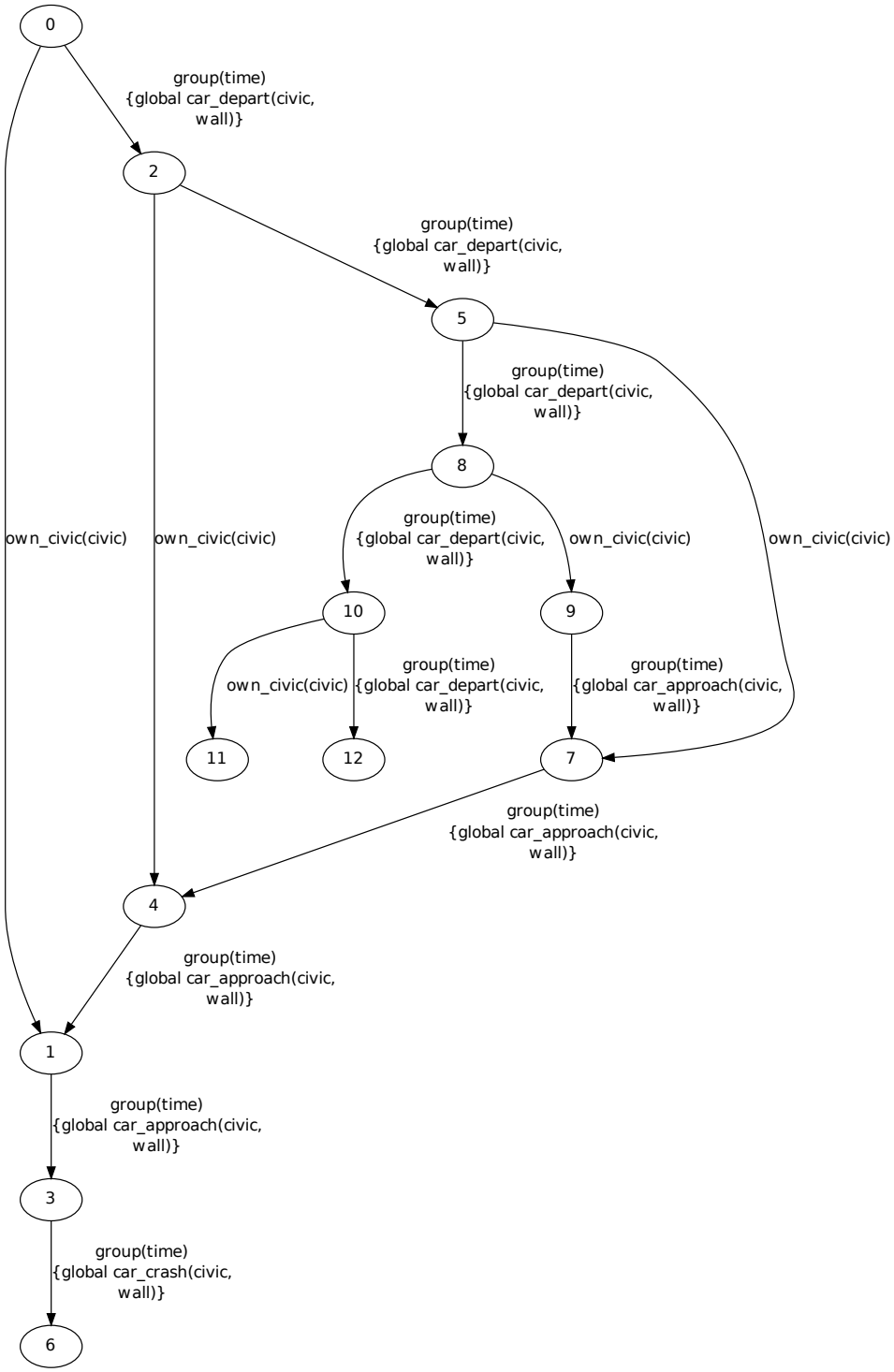

Figure 5.8: One-car automotive attack graph

```
network model=
    assets:
        civic;
        accord;
        wall;
    facts:
        platform:civic,cpe:/h:honda:civic;
        quality:civic,compromised=false;
        quality:civic,status=up;

        platform:accord,cpe:/h:honda:accord;
        quality:accord,compromised=false;
        quality:accord,status=up;

        platform:wall,cpe:/h::wall;

        topology:civic<->wall,distance:=50;
        topology:accord<->wall,distance:=60;
.
```

Figure 5.9: Two-car example network model

5.5.2 Denial of Sleep

The final example serves to demonstrate further the interaction of discrete attacks with time; it is also used later to discuss scaling behavior. The network model given in Fig. 5.11 describes a system of a single RFID reader with a single RFID tag, in the ISO 18000-7 system as described in Section 2.5.2. The tag begins with a battery life of 100, which is drained at a rate of 25 per time step if the sleep denial attack is being executed, and at a rate of 10 per time step if not. This behavior is shown in the exploit patterns in Fig. 5.12. The scenario's discrete attacks `own_reader` (which gives the attacker control of the reader) and `deny_sleep` (which causes a tag to enter the denial of sleep mode) are shown in Fig. 5.13.

Two attack graphs help to demonstrate the execution of this model. The first is given in Fig. 5.14, which is the attack graph generated to a depth of 6. Observe that the leftmost set of edges is the worst case scenario: the attacker immediately takes control of the network and kills the tags in only 4 time steps. The rightmost

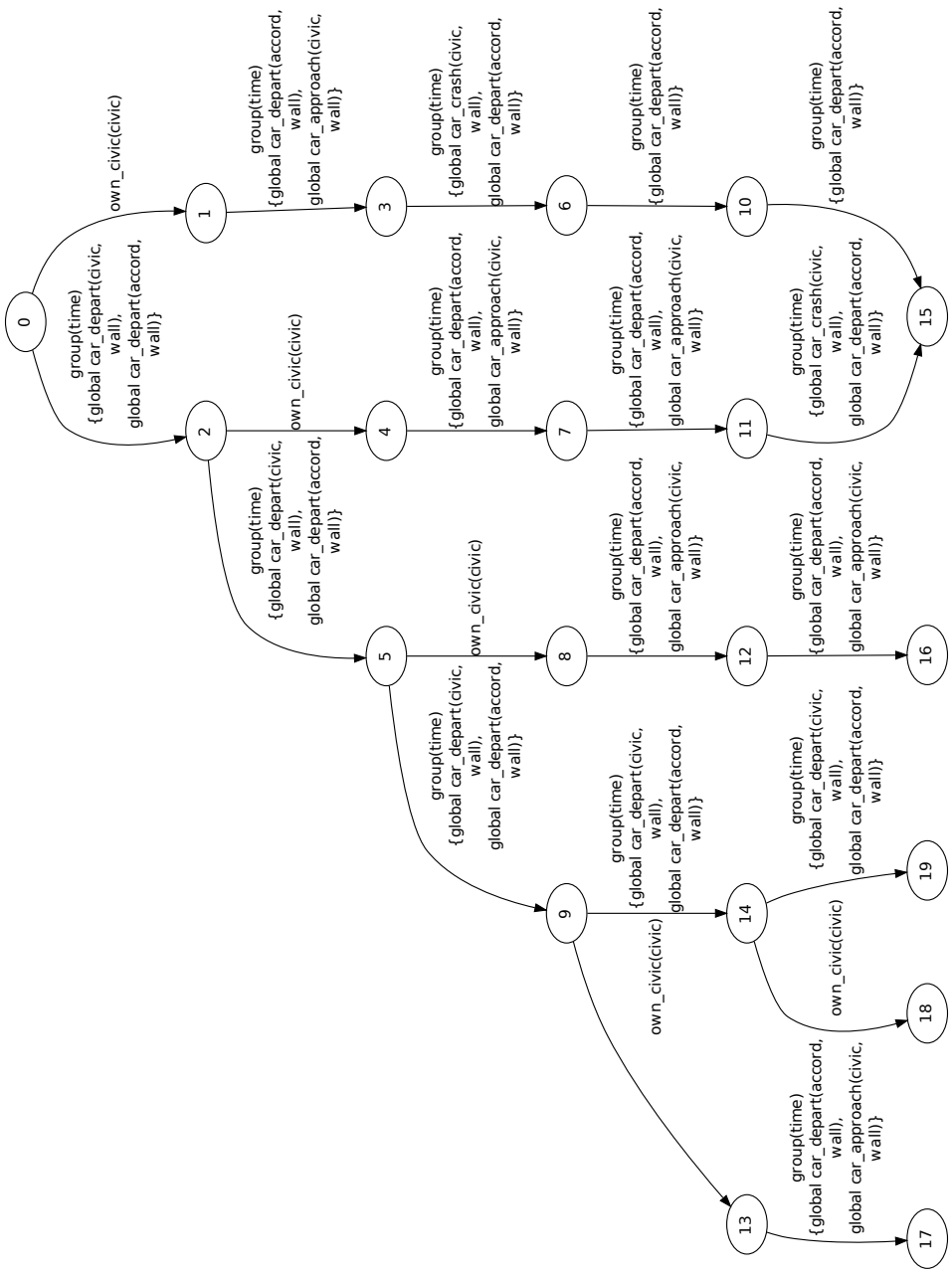

Figure 5.10: Two-car automotive attack graph

```
network model =
    assets :
        attacker;
        reader;
        tag0;

    facts :
        # Reader:
        platform:reader,cpe:/h::VulnerableReader;
        quality:reader,status=up;

        # Tag:
        platform:tag0,cpe:/h::Tag;
        quality:tag0,status=up;
        quality:tag0,power:=100;
        quality:tag0,mode=sleep;

        # Topologies:
        topology:attacker -> reader,connected_network;
        topology:reader -> tag0,connected_rfid;
.
```

Figure 5.11: One tag RFID system network model

edge, then, is the best case scenario: in this situation, the attacker never (to a depth of 6) acts. The only transitions are those caused by the normal time evolution of the system.

Likewise, Fig. 5.15 is the attack graph generated to its full depth. There are only two leaf nodes: state 21, in which the attacker has executed the denial of sleep attack to the demise of the tag, and state 40, in which the tag has died on its own.

```
global group(time) exploit wake_power_dec(t)=
    preconditions:
        platform:t,cpe:/h::Tag;
        quality:t,status=up;
        quality:t,mode=wake;
        quality:t,power>25;
    postconditions:
        update quality:t,power-=25;
.

global group(time) exploit sleep_power_dec(t)=
    preconditions:
        platform:t,cpe:/h::Tag;
        quality:t,mode=sleep;
        quality:t,status=up;
        quality:t,power>10;
    postconditions:
        update quality:t,power-=10;
.

global group(time) exploit wake_power_die(t)=
    preconditions:
        platform:t,cpe:/h::Tag;
        quality:t,mode=wake;
        quality:t,status=up;
        quality:t,power<=25;
    postconditions:
        update quality:t,power:=0;
        update quality:t,status=down;
.

global group(time) exploit sleep_power_die(t)=
    preconditions:
        platform:t,cpe:/h::Tag;
        quality:t,mode=sleep;
        quality:t,status=up;
        quality:t,power<=10;
    postconditions:
        update quality:t,power:=0;
        update quality:t,status=down;
.
```

Figure 5.12: One tag RFID system continuous exploit patterns

```
exploit own_reader(a, r)=
    preconditions:
        platform:r,cpe:/h::VulnerableReader;
        quality:r,status=up;
        topology:a->r,connected_network;
    postconditions:
        insert topology:a->r,access_admin;
.

exploit deny_sleep(a, r, t)=
    preconditions:
        platform:r,cpe:/h::VulnerableReader;
        quality:r,status=up;

        platform:t,cpe:/h::Tag;
        quality:t,status=up;
        quality:t,mode=sleep;

        topology:a->r,access_admin;
        topology:r->t,connected_rfid;
    postconditions:
        update quality:t,mode=wake;
.
```

Figure 5.13: One tag RFID system discrete exploit patterns

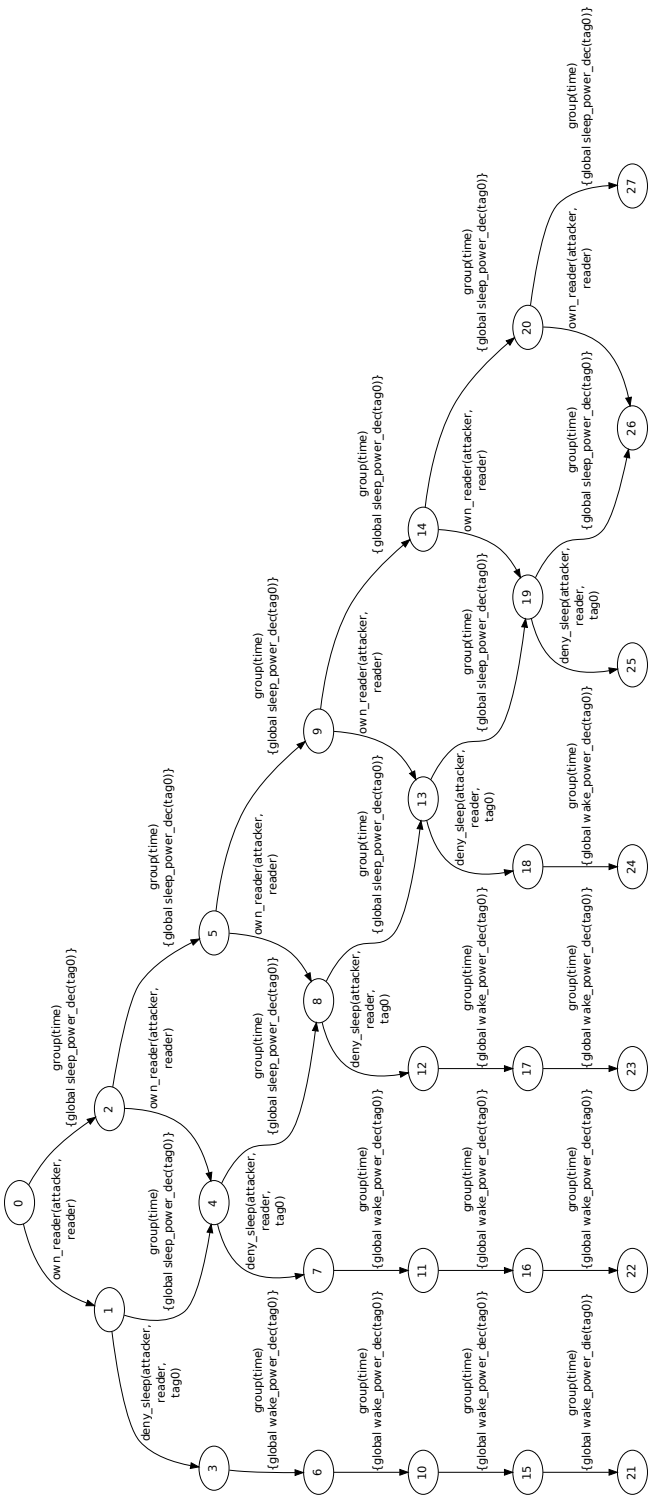

Figure 5.14: One tag RFID system attack graph (depth of 6)

Figure 5.15: One tag RFID system attack graph (full depth)

CHAPTER 6

RESULTS

6.1 Introduction

The preceding chapters introduced a framework for discrete attack graphs, a set of enhancements to that framework to improve its modeling capabilities for discrete systems, and a set of extensions to enable it to model cyber-physical systems.

This section provides an analysis of the resulting framework (including a discussion of its performance) and its capabilities (articulating its contribution to the hybrid systems modeler), as well as a discussion of the network and system analysis that it enables.

6.2 Discrete Capabilities

Previous chapters articulate the enhancements provided by platform facts, host processing, and the standard lexicon, which provide conveniences to the modeler and ease model acquisition from a standard network representation. However, further discussion is warranted of the power added by global exploits and grouping exploits.

In short, globalness applies one exploit to multiple asset bindings, whereas grouping applies multiple exploits to the same asset binding. Consider the goal of modeling a fuzzer or automated attack script that simultaneously attempts to exploit a variety of common web vulnerabilities on a form on a web server. A successful buffer overflow may give the attacker login access to the server; successful cross-site scripting may give the attacker the ability to spoof messages to users; and a successful SQL injection attack may give the attacker access to sensitive information. Grouping these automated attacks together enables the modeler to better capture the behavior of

```
global exploit kill_router(a, r, h)=
    preconditions:
        platform:r,cpe:/h::VulnerableRouter;
        quality:r,status=up;
        quality:h,status=up;
        topology:a->r,connected_network;
        topology:h->r,connected_local_to_internet;
    postconditions:
        insert quality:r,status=down;
        insert quality:h,status=down;
.
```

Figure 6.1: Global exploit disabling an entire network

such a "script kiddie."

Consider also the case of executing a denial of service attack on a single network of arbitrary size by disabling a perimeter device. Such an attack would be impossible to encode in a single transition without the use of a global exploit. Such an exploit might appear as in Fig. 6.1.

Additionally, the hybrid extensions also permits the use of networking concepts such as port number (in which, typically, the use of a port number 1024 or below requires administrative rights) or latency for the purposes of modeling certain scenarios on information systems without physical components.

6.3 Hybrid Systems Modeling Capabilities

The real power of the new extensions show in modeling hybrid systems. For this discussion it is useful to draw upon the theory of hybrid automata for hybrid systems. A useful goal is the ability to model any hybrid system (expressible as a hybrid automaton) in the hybrid attack graph formalism. This is hindered by two major constraints: the inability to use variables in the right hand side of precondition and postcondition expressions in hybrid automata, and the requirement that the time domain be discretized.

6.3.1 Classes of hybrid systems modeled by hybrid attack graphs

Henzinger and others have articulated a special case of the hybrid automaton called the linear hybrid automaton [16]. The linear hybrid automaton is a hybrid automaton in which (1) all conditions in the automaton are boolean combinations of linear inequalities, and (2) all flow conditions use only free variables from $\dot{X}$. In this case, a linear inequality is an inequality on linear terms, or linear combinations of integer constants with real valued variables.

Clearly condition (1) does not hold, because of the lack of variables on the right hand side of the operators in the attack graph language. However, the subcase in which the left hand side is the variable in question, and the right hand side is the constant k_0 does.

Condition (2) is more subtle but less restricting. As it permits free variables only from $\dot{X}$, so long as only one free variable from $\dot{X}$ is used this is exactly equivalent to the behavior permitted in time evolution exploit patterns. In the less deterministic (and less common) case in which multiple free variables from $\dot{X}$ are involved in a flow condition, the hybrid attack graph language permits an execution that conforms to the hybrid system specification, but not one that explores all possibilities.

6.3.2 Hybrid systems to hybrid attack graph mapping

The hybrid attack graph permits discrete time simulation of a small, very well behaved subclass of linear hybrid automata. Nevertheless, this provides sufficient power to capture a wide variety of systems. Informally, including a HA in a hybrid attack graph can be accomplished using the following general mapping:

Hybrid automata onto assets The automaton is represented by an asset. Therefore its variables take the form of facts on that asset, its conditions take the form of preconditions and postconditions on exploits, and events, jumps, and time evolution are done on the exploits.

Control modes onto qualities The control mode is selected based upon a quality on the asset. The mode is implemented by the time evolution exploit responsible for that particular asset in that particular mode. Control switches are not explicitly encoded but can be thought of as part of the exploits that change the asset's mode.

State variables onto qualities The state variables are encoded explicitly as qualities about the asset in the fact base.

Initial conditions onto network model The initial conditions are implemented in the network model initial state.

Jump conditions onto non-time exploits Jump conditions, which select when a mode transition *may* occur, are mapped onto the conditions of a non-time exploit, so that they may be fired, but not as part of the time stepping.

Invariant conditions onto time exploits Invariant conditions indicate when a mode *must* change in order for the inexorable passage of time to proceed. Therefore, these conditions are placed explicitly into time exploits to ensure that they are executed before time may continue to pass. Their postconditions include a change of the discrete mode quality.

Flow conditions onto time exploits Flow conditions, which dictate the continuous evolution of the system over time, are implemented as rates of change in time exploit postconditions (e.g. `temp-=5`). Their repeated application over discretized time intervals cause the simulation of continuous time evolution.

Events onto global groupings Event labels, which are used to synchronize mode transitions between multiple automata, are implemented as global group names, which force the globally grouped exploits to fire on the synchronized assets simultaneously.

6.3.3 Zeno concerns

Two additional concerns exist, to which brief mention is due: *globally zeno* behavior and *isolated zeno* behavior.

Globally zeno behavior occurs in a hybrid attack graph when the repeated application of non-timed exploits is capable of providing an infinite transition path in which there are no time evolution attacks. That is, a model exhibiting globally zeno behavior not only generates an infinitely large attack graph, but it also generates one that is capable of causing the convergence of time, which obviously fails to capture real world behavior.

Such models are probably (but not necessarily) ill-formed. Infinitely repeated application of discrete exploits is a sign of poor modeling decisions: ideally, exploits should not be applied multiple times to the same assets; if they are (as in the setting of a mode), they should depend upon either the source mode being different from the destination mode or else depend upon some other result of time evolution. In the case of mapping onto hybrid automata, this means that self loops (jumps from a mode onto itself) need to be addressed with caution.

Isolated zeno behavior, which may also be the sign of an ill-formed model, occurs when an otherwise active hybrid asset is not fully covered by the time exploit group. This may occur legitimately if no continuous evolution is occuring in the asset's mode, or if (as in the case of the Civic in the car crash example pictured in Fig. 5.10) the asset is no longer an active part of the simulation. If this behavior appears, care should be given that it is not an oversight on the part of the modeler.

6.4 Performance

Although the purpose of this work is not focused upon the performance of the generation process, a discussion of the performance of the reference generator is warranted. This section first provides an outline of the execution of the generation

Domain	Symbol
Facts	$F = \lvert\mathcal{Q} + \mathcal{T}\rvert$
Fact names	f
Exploits	$E = \lvert\mathcal{E}\rvert$
Depth	d
Assets	$A = \lvert\mathcal{A}\rvert$
Qualities	$Q = \lvert\mathcal{Q}\rvert$
Topologies	$T = \lvert\mathcal{T}\rvert$
Most preconditions in any exploit	p
Most postconditions in any exploit	P
Most parameters in any exploit	a
Possible asset bindings	B

Table 6.1: Symbols for the magnitudes of attack graph input domains

program, decorated with the order of magnitude of the asymptotic bound of the execution time at each step. This is used to provide a bound on the scaling of the generator as a whole. The next subsection demonstrates the empirical results of a performance analysis, providing graphical scaling data from code profiling to confirm the predictions of the outline.

It is worth adding that, although this thesis's reference generator scales poorly, a significant amount of the attack graph research community as a whole focuses specifically upon the performance of their generators. Furthermore, care has been given that the introduction of continuous values does not alter the complexity of any step of the generation process. For these reason, the poor performance of the reference generator is assuaged by the applicability of the wider attack graph research community's performance results.

Driven by the formal definition of the attack graph domains given in Section 2.4.2 and treating platform facts as just a special case of qualities, notation will henceforth be as in Table 6.1, presented in order of appearance in the program outline.

6.4.1 Program Outline and Asymptotic Analysis

The following outline articulates the order of execution steps in the reference

generator. Each boldface step is decorated with the order of the number of times that piece of code is executed. Indented steps are substeps, which are executed on the order of their own decorated order, multiplied by the orders of each of their parent steps. The outline uses the symbols defined in Table 6.1.

Build attack graph $O(1)$

- **Load initial state** $O(F)$
- **Load exploits** $O(E)$
- **Generate attack graph (recursive)** $O(d)$
  - **For each analysis state** $O(states)$
    - **Copy network model** $O(A) + O(F)$
    - **Get valid attacks** $O(1)$
      - **For attack in attack bindings** $O(BE)$
        - **Validate attack** $O(p)$, which is technically bounded by $O(A^2) \cdot O(F)$ but is in practice quite small
        - **Process groups and globals** $O(1)$
    - **For each valid attack** $O(BE)$
      - **Get successor state** $O(1)$
        - **Build network model** $O(A) + O(F)$
        - **Add postconditions** $O(P)$, which is technically bounded by $O(A^2F)$ but is in practice quite small.

There are two non-obvious variables included here. One is B, the possible binding combinations value, which is actually the result of a permutation operation. This is bounded loosely by by $O(A!)$ (or $O(A^A)$) and more tightly for a given scenario by $O\left(\frac{A!}{(A-a)!}\right)$ (alternatively $O(A^a)$), where a is the length of the longest parameter list on any exploit pattern.

The other is "states." In this case, it refers to the number of "analysis states" in an iteration of the generation function, which is actually defined in terms of the previous iteration's states. Luckily, we can avoid having to solve for this quantity by combining the sections "Generate attack graph (recursive)" with "For each analysis state." Using the knowledge that the body of that for loop is reached once and only once for each state generated, the bound for the number of executions of the combined structure is equivalent to the bound on the number of states in the attack graph. Because a scenario is not guaranteed to converge to a closed attack graph, this is necessarily a function of the maximum depth. Each state may produce up to $B \cdot E$ new states. This may happen up to d (generation depth) times. Using the derivation of B from earlier in this section, this means that the number of states produced is at worst $(A!E)^D$. More tightly, this is in $O\left(\left(\left(\frac{A!}{(A-a)!}\right)E\right)^d\right)$. This controls the scaling behavior of the recursive generator function.

Furthermore, the fact "count" should actually be broken up into qualities and topologies (treating platform facts as special cases of qualities for the purposes of this analysis), as the possible number of these fact types grow with respect to the asset count at different rates. The bound on facts, then, is actually $O(A \cdot Q + A^2 \cdot T)$.

Refer to Table 6.1. It includes P and p, which refer to the largest number of pre- and post-conditions in any single exploit in the input. These can be bounded (in practice, very loosely) by $O((AQ + A^2T)(A))$, but this bound is due to the (never practically implemented for nontrivial A) possibility of an exploit taking every possible asset parameter and operating on every possible quality and topology for each possible asset combination. Similarly, the number of parameters in exploit patterns could be bounded loosely by $O(A)$, but this is not realistic. Therefore, the new a, p and P inputs are used instead. The symbol for the number of fact names, f, is used in the next section to simplify some notation.

The outline reveals the following bound on the generation process:

$O((EB)^d((A+F)+BEp+BE(A+F+P)))$.

Recall that $B = \frac{A!}{(A-a)!}$, so is in $O(\frac{A!}{(A-a)!})$ (more loosely, $O(A^a)$ or $O(A!)$ or even $O(A^A)$ are also valid), and that $F = AQ + A^2T$ (More loosely, A^2f). Then the generation process is more precisely in:

$O(((\frac{A!}{(A-a)!}E)^d)((A+AQ+A^2T)+\frac{A!}{(A-a)!}Ep+\frac{A!}{(A-a)!}E(A+AQ+A^2T+P)))$

Switching to the looser bounds of A for p and P, A^A for B (thereby eliminating a), and A^2f for F simplifies the presentation of this expression to a function only of f, E, A, and d. It is already clear that execution time is controlled almost completely by the growth of A and d, but this resultant asymptotic bound for execution time of the (hybrid) attack graph generation process presents that fact quite dramatically:

$O((A^AE)^d((A+A^2f)+A^AEA+A^AE(A+A^2f+A)))$

$= O(fEA^{Ad+A+2})$

$= O(E^{d+1}fA^{A(d+1)+2})$

$\approx O(E^{d+1}fA!^d)$ (in a mild abuse of notation)

Worst case generation time, then, is expected to grow linearly with fact names, polynomially with exploit patterns, exponentially with generation depth (for divergent graphs), and factorially with assets. The next sections demonstrate empirically the effects of generation depth and asset growth on execution time.

6.4.2 Execution Time Results

Two sets of trials serve to demonstrate the scaling behavior of execution time: on assets, and depth. These were executed using a combination of the homogeneous automotive example (in which the only cars are "Civics") from Section 5.5.1 and the RFID example from Section 5.5.2.

Asset scaling: The first set of tests, which studies the effect of increasing the number of input assets while holding everything else (including depth) constant, includes both the car and RFID scenarios. Inclusion of additional cars and tags provides

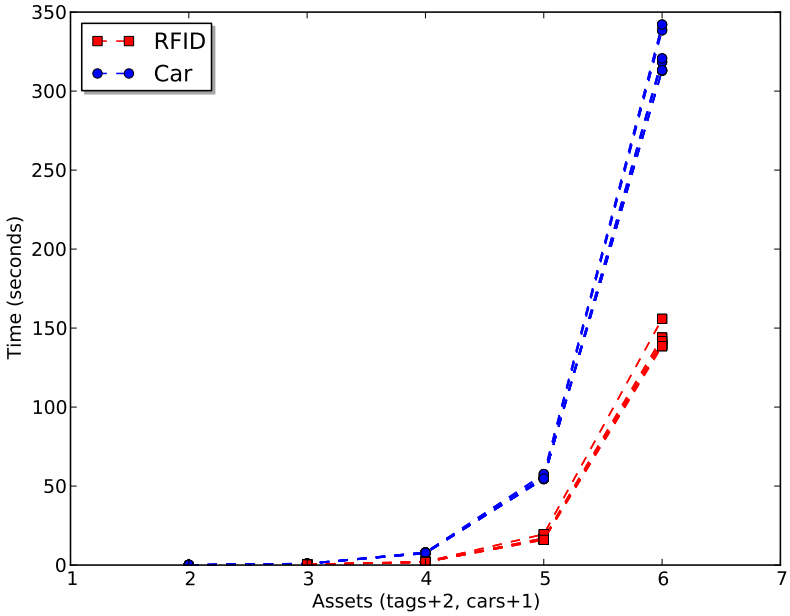

Figure 6.2: Execution time vs. number of assets

the asset scaling. The RFID scenarios were allowed to continue until convergence; the car scenarios, which do not converge, were generated to a depth of 10. Each asset number (from 2 to 6, representing 1 to 5 cars; and 3 to 6, representing 1 to 4 RFID tags) was executed on each scenario 6 times. The results, showing dramatic exponential behavior, are provided in Fig. 6.2. These are graphed on a logarithmic scale with a reference line of $\frac{A^A}{2^4}$ in Fig. 6.3.

This exponential behavior especially affects hybrid attack graphs with time because time group exploits are likely to bind to nearly every permutation of asset choices. As an example, observe the large number of permutations of bindings of the time group exploits to car assets in Fig. 6.4.

Depth scaling: The second test set, performed by varying the generation depth of the program, again with 6 trials, serves to illustrate both the exponential growth of execution time with depth, as well as the difference in the way that depth affects converging attack graphs (such as the RFID example) and diverging ones (such as

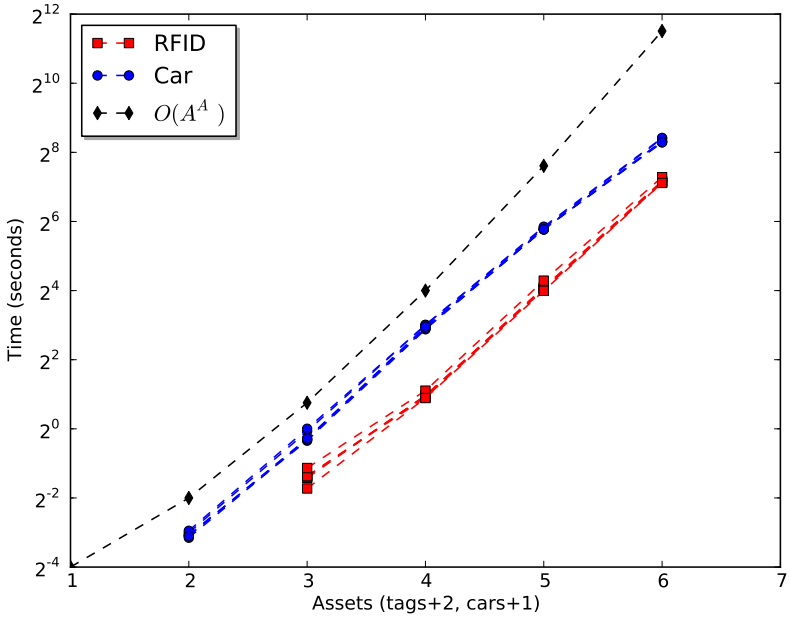

Figure 6.3: Execution time vs. number of assets (log scale)

the car example). The graph of this performance is provided in Fig. 6.5 and on a logarithmic scale in Fig. 6.6.

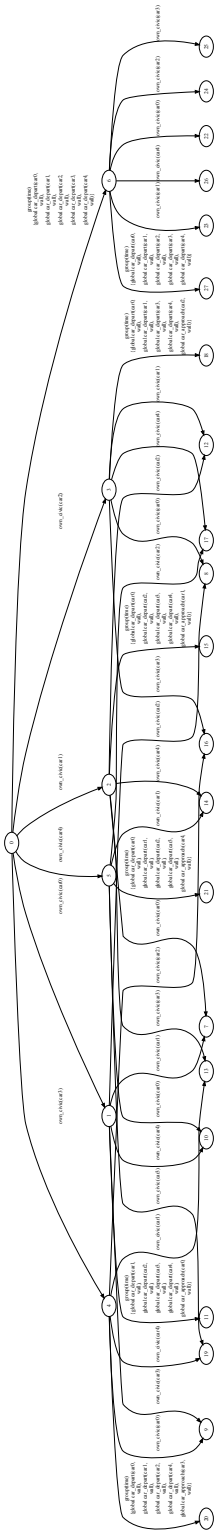

Figure 6.4: 5-car hybrid attack graph to depth 2

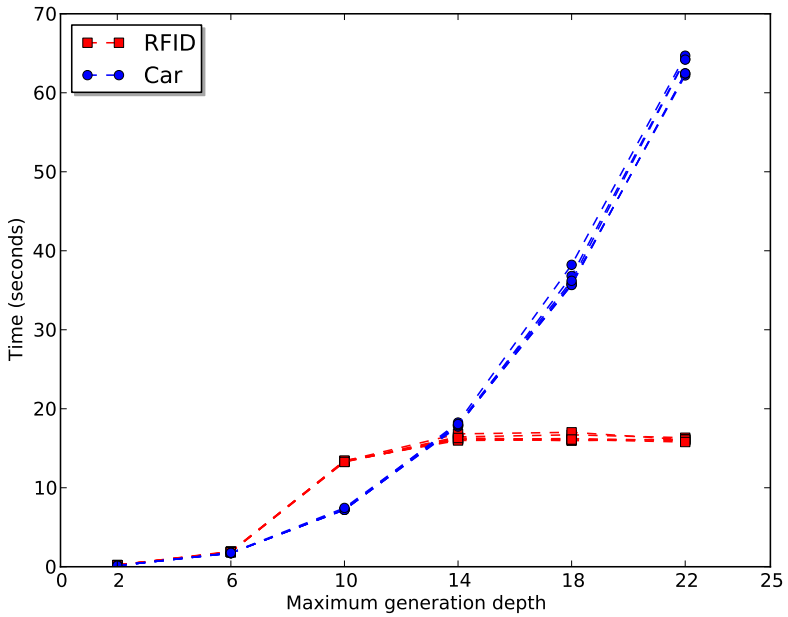

Figure 6.5: Execution time vs. generation depth

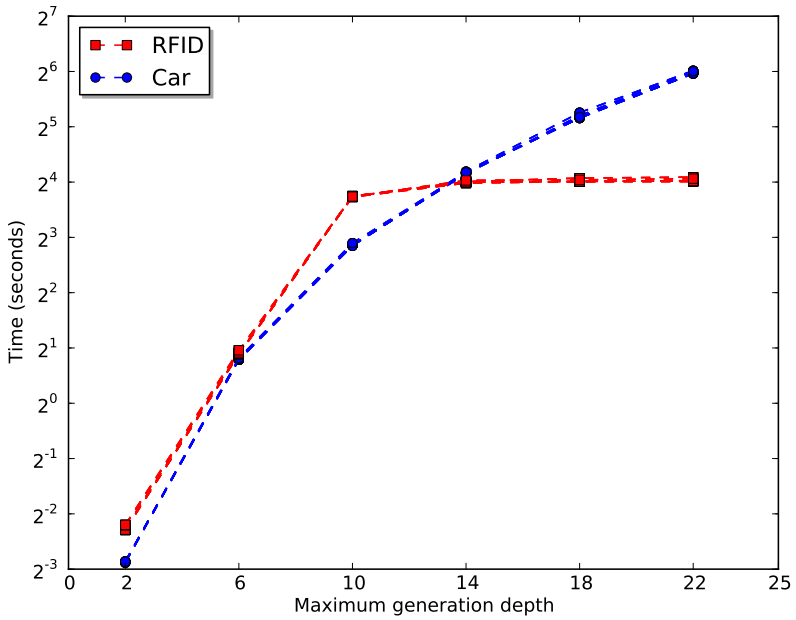

Figure 6.6: Execution time vs. generation depth (log scale)

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

7.1.1 Summary

This thesis presents a set of expansions to the attack graph formalism permitting the modeling of hybrid systems in the context of interacting exploits and components. Several contributions in the modeling domain include integration with the Common Platform Enumeration, proposal of a standard lexicon of fact types for easing integration with the Common Vulnerability Enumeration and the National Vulnerability Database, and dealing with network objects without loss of generality in the framework.

The hybrid extensions provide the ability to model a small but well-behaved subset of hybrid systems that nevertheless proves quite expressive. However, its generation needs effort to measure up to the current state of the art in performance for attack graph generation.

7.1.2 Shortcomings

Model acquisition: Although this thesis takes great strides forward in the articulation of a language and framework that will be well suited to a wide variety of networks, a significant roadblock to its wide use is in network model and exploit pattern acquisition: although their formats are convenient, they must be automated before large scale adoption is possible.

Performance: The worst-case execution time of the reference implementation of the attack graph generator scales factorially with input assets and exponentially with maximum depth. This is exacerbated by the tendency of time group exploits to bind with large numbers of assets in nearly exhaustive combinations, behavior that can be seen in the automotive attack graph examples. Using this tool, generation of attack graphs from nontrivial scenarios (with tens of hybrid assets) in acceptable time is not likely.

Fortunately, the addition of these hybrid extensions does not qualitatively affect the complexity of attack graph generation. For this reason, existing research into improvement of generation performance is expected to be equally applicable to hybrid attack graphs.

Cognitive scalability: Perhaps the most severe difficulty with attack graphs is their size and visual complexity. On their face, they are valuable tools for applying human intuition to network attack scenarios. However, once the scenario grows beyond a handful of nodes, the size balloons quickly out of hand. This is even more marked in time-driven hybrid attack graphs; repeated application of timed exploits that give little useful data individually further frustrates efforts to promote cognitive scalability.

Separation of concerns: The use of "exploits" to implement a system's evolution over time is unsatisfying. Although it was not stated initially, it is clear that separating concerns—keeping network model and behavior in the network model and attacker actions in the exploit patterns—is a useful design goal especially for the benefit of a modeler.

7.2 Future Work

7.2.1 Network Model

First and third person modeling: This thesis used almost exclusively a third person strategy for modeling attackers and their connection to network assets. In the future, it is worth exploring the benefits of the first person strategy as well as considering a blended model in which the Internet (or any network sufficiently large to wholly cover the domain of the scenario) is implicit and connections to it are treated as qualities; only local and adjacent access would be implemented as topologies. This has the potential to ease some of the problems inherent in chaining together connection topologies across multiple nodes to give the attacker gradual, intermediated access to a distant host. It may also simplify exploits.

Model acquisition: Significant work needs to be done in the acquisition of models. The integration of the Common Platform Enumeration eases this; more integration with the Security Content Automation Protocol (SCAP) may ease this further. Nevertheless, some kind of network scanner tool will need to be adapted to provide automated acquisition of network models in order for attack graph specification to be automated enough to be a tractable task for larger systems.

Furthermore, such acquisition may be difficult or even impossible in a hybrid setting. A significant research challenge exists in model acquisition on the physical side of the cyber-physical divide.

7.2.2 Exploit Model

Postcondition classification: There would be value in automated classification of exploit postconditions. That is, there is a limited number of possible security consequence classifications of any attack. The National Vulnerability Database contains a partial attempt to enumerate them. Microsoft makes a similar attempt with STRIDE [34]. If an exploit can be automatically (or even manually by its source prior to acquisition for use in the attack graph generator) tagged with its consequences or

potential consequences, this could (1) ease automated creation of a general exploit pattern and its postconditions, and (2) aid in analysis by flagging possible consequences of reaching a particular node.

7.2.3 Exploit Pattern Enhancement

There are a variety of systematic exploit pattern processing enhancements that would prove useful.

Wildcard exploit conditions: If exploit preconditions were to permit regular expressions or wildcard expressions in their statements, this would enable several useful behaviors. In fact, even permitting the prefix property to apply would help as well. For example, a precondition of "connected" could be satisfied by either "connected_network" or "connected_local." This would permit a hierarchy of network localities that more accurately reflects the real world.

Pivot exploits: On a similar note, a valuable tool for modelers would be generic "pivot" exploits. For example, consider the `ssh_http_tunnel` exploit in the Blunderdome attack graph exploit patterns (see Fig. 4.7). This exploit permits an attacker with certain access to one asset to "pivot" from that asset onto all the assets that are connected to it. Specification of these generic pivot exploits, possibly supported by wildcards in the generator, would greatly ease the task of a modeler.

Boolean preconditions: Permitting boolean operators in exploit preconditions would be valuable, as otherwise (currently) multiple exploits with the same sets of postconditions but slightly varied preconditions must be used to represent exploits that may affect, for example, Linux version 2.6.33.19, 2.6.34.10, and 2.6.38.8.

Variable conditions: Currently, the right hand side of exploit conditions must be literals. If they could be expressions involving variables loaded from the fact base,

this would satisfy the required conditions for linear hybrid automata equivalence (see Section 6.3.1).

7.2.4 Hybrid Modeling challenges

Time is difficult to model and interpret when implemented in this way. One alternative may be to again to hybrid automata for inspiration and adopt a strategy similar to its *time-abstract* semantics, which indicate *that* time has passed, but not precisely *how much* has passed. This kind of implementation would mitigate issues of size and, potentially, scaling.

7.2.5 Generation

Performance: The generation performance of the hybrid attack graph generation needs to be brought into step with the current state of the art. This is not the goal of this particular work, but it is an important step in its future application.

Parallelization: One route to improving performance that would be of use to the wider world is the parallelization of the generation procedure. As scaling remains an issue in attack graph generation in general, parallelizing the process could permit greater scaling and possibly allow attack graph analysis to grow to the scale of large enterprise networks.

7.2.6 Attack Dependency Graphs

Finally, one route for future work that deserves specific emphasis is the recently articulated attack dependency graph (also called the exploit dependency graph). A variation of the attacks graph adopted by some researchers, attack dependency graphs build graphs of the dependencies of attacks upon each other [20, 29]. These smaller models may be more efficiently generated, encode equivalent information, and eliminate visual complexity.

Attack dependency graphs address several of the shortcomings of the stateful hybrid attack graphs described in this chapter. They are more efficient to generate, easing performance concerns. Furthermore, proposed hybrid versions [24] exhibit the desired time abstract behavior and separation of concerns mentioned above. These may very well be the best direction for future work in hybrid attack graph study to proceed, and much of the work presented in this thesis is applicable to hybrid attack dependency graphs.

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] Andrew Buttner and Neal Ziring. Common platform enumeration (cpe) — specification, March 2009.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. *Managing Cyber Threats*, pages 247–266, 2005.

[21] 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.

[22] 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.

[23] 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.

[24] G. Louthan, P. Hardwicke, P. Hawrylak, and J. Hale. Toward hybrid attack dependency graphs. In *Proc. Cyber Security and Information Intelligence Research Workshop*, 2011.

[25] 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.

[26] 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.

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

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

[29] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118. ACM, 2004.

[30] 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.

[31] 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.

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

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

[34] A. Shostack. Experiences threat modeling at microsoft. In *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*, 2008.

[35] 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.

[36] 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.

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