Combinatorial Analysis of Network Security

Steven Noel[a], Brian O'Berry[a], Charles Hutchinson[a], Sushil Jajodia[a] ,
Lynn Keuthan[b], and Andy Nguyen[b]

[a]George Mason University Center for Secure Information Systems
[b]Defense Information Systems Agency

ABSTRACT

We extend the traditional analysis of network vulnerability by searching for sequences of exploited vulnerabilities distributed throughout a network. While vulnerabilities considered in isolation may seem innocuous, when considered in combination they may lead to serious security breaches. Our approach establishes encoding rules to reason about interdependent vulnerabilities and exploits. It then reasons about the rules to perform critical failure analysis for a given network. We have developed a prototype software tool for automating the analysis, which can be integrated with existing network security tools such as vulnerability databases and network discovery tools. We demonstrate our approach through an example application. We also perform a scaling experiment to show the performance of our approach for larger networks.

Keywords: Information security, vulnerability analysis, formal methods, model checking

1. INTRODUCTION

In the current state of the art, security vulnerability analysis tools consider individual vulnerabilities, independent of one another. Moreover, they analyze single machines only, in isolation from other machines in the network. But the interdependency of vulnerabilities and the connectivity of a network make such analysis incomplete. While a single vulnerability itself may not pose a significant direct threat to a system, a combination of vulnerabilities may.

Thus even well administered networks are vulnerable to attacks, because of the security ramifications of offering a variety of combined services. That is, services that are secure when offered in isolation nonetheless render the network insecure when offered simultaneously. Many current tools address vulnerabilities in isolation and in the context of a single host only. We extend this by searching for sequences of interdependent vulnerabilities, distributed among the various hosts in a network.

The search for sequences of vulnerabilities within a network is very similar to the problem of generating test cases for system verification. For each of these problems, an enumeration of all possible combinations of system inputs is desired. For vulnerability analysis, the system to be tested is a model of the network security attributes and their relationships.

Within the area of formal methods, model checkers are particularly adept at generating test cases because of their ability to generate counterexamples. For network vulnerability analysis, test cases generated by a model checker correspond to attack scenarios. We encode the vulnerabilities in a state machine description

Additional author information
S. Noel: snoel@gmu.edu; phone (703) 993-3946; fax (703) 993-1638; B. O'Berry: boberry@gmu.edu; phone (703) 993-3946; fax (703) 993-1638; C. Hutchinson: chuckhutchinson2@yahoo.com; phone (703) 993-3946; fax (703) 993-1638; S. Jajodia: jajodia@gmu.edu; phone (703) 319-0877; fax (703) 993-1638; L. Keuthan: keuthanl@ncr.disa.mil; phone (703) 882-1557; fax (703) 882-2824; A. Nguyen: nguyena@ncr.disa.mil; phone (703) 882-1557; fax (703) 882-2824

suitable for a model checker. We then assert via temporal logic that an attacker cannot take a given action on a given host. The model checker either offers assurance that the assertion is true (i.e. that the target is secure), or provides a counterexample detailing each step of a successful attack.

Our approach establishes encoding rules to reason about interdependent vulnerabilities and exploits, to perform critical failure analysis for a given network. We have also developed a software tool for automating the analysis. This tool for advanced vulnerability analysis can also be integrated with existing network security tools such as vulnerability databases and network discovery tools.

In the next section, we describe general considerations for network attack. Section 3 then describes our approach for combinatorial analysis of network attack vulnerability. In Section 4, we discuss details for reasoning engines that search for valid combinations of attacker techniques. Section 5 then shows an example application of our approach. Finally, Section 6 performs an experiment to show how our approach scales for larger networks.

2. CHARATERSTICS OF NETWORK ATTACK

To advance an attack, the attacker must know various techniques, called exploits. An exploit generally has specific conditions that must exist for it to be successful. Such attack pre-conditions might include the presence of certain vulnerable programs, sufficient user privileges, or a particular form of connectivity to another machine. Successful exploits then generally induce a new set of conditions within the network. Such exploit post-conditions might include elevated user privilege, increased connectivity, or establishment of a trust relationship. The concept of exploit pre- and post-conditions is shown in Figure 1.

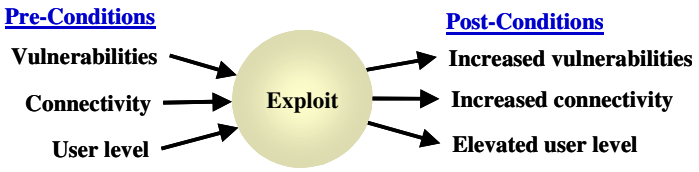

Figure 1 – Exploit pre-conditions and post-conditions.

Successful attacks generally consist of a series of exploits that gradually increase the vulnerability of the network, until some final attack goal is reached. Such attacks are possible because of dependencies among exploits in terms of pre-conditions and post-conditions. That is, a pre-condition of one exploit may be a post-condition of another. Such dependencies form a directed graph in terms of exploits and vulnerabilities shown in Figure 2.

Vulnerabilities generally differ among the various machines in a network. That is, different platforms, configurations, available services, etc. imply different sets of vulnerabilities on the corresponding machines. Thus, various machines may have differing exploit dependency graphs, as shown in Figure 3. In the figure, the arrows between machines show their connectivity.

Attackers search for vulnerabilities on the machines with which they have connectivity. Once a target machine is sufficiently compromised, the attacker can launch attacks from it. With this new attack locus, the attacker can extend the number of machines that can be searched for vulnerabilities, perhaps eventually be taking control of other machines. The attacker can continue this process until his goal is met, as in Figure 4, or until there are no other vulnerabilities to exploit.

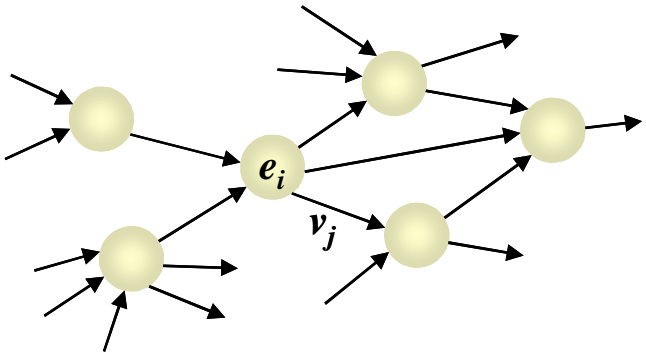

Figure 2 – Interdependencies among exploits.

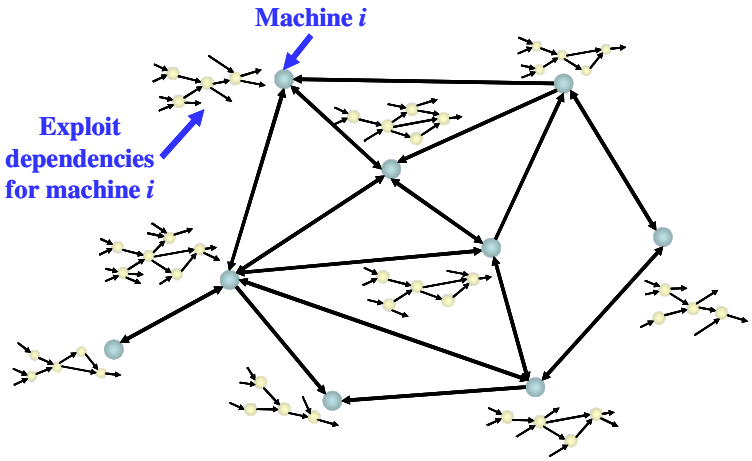

Figure 3 – Network-wide vulnerabilities and exploits.

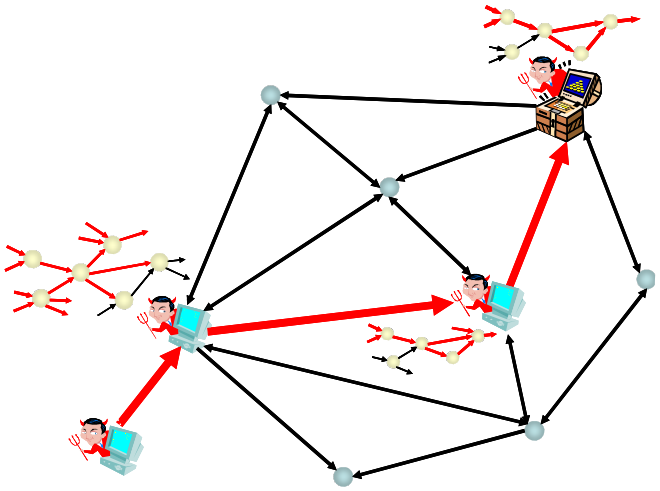

Figure 4 – Attacker's goal is met.

3. COMBINATORIAL ANALYSIS OF NETWORK VULNERABILITY

Because of the interdependencies of exploits and their dependence on machine connectivity, a combinatorial approach is necessary for full understanding of attack vulnerability. The traditional approach of considering network components in isolation and reporting vulnerabilities independent of one another is clearly insufficient.

Our general approach to the combinatorial analysis of network vulnerability involves rule-based modeling of attacker exploits in terms of exploit pre- and post-conditions. This captures the interdependencies of exploits, including connectivity dependencies. Inference engines then reason about combinations of exploits. That is, assertions are made concerning whether particular attack goals can be met, which are then proven through rule inference. The result is the discovery of attack paths for the assumed attacker goals.

We implement this approach to combinatorial vulnerability analysis through the architecture in Figure 5. In an initial modeling phase, one specifies a rule-based model of network attack. A subsequent reasoning phase determines whether the attack goal can be reached from the initial network state, based on the attack rules. The sequence of steps leading from the initial state to the goal state constitutes an attack path through the network.

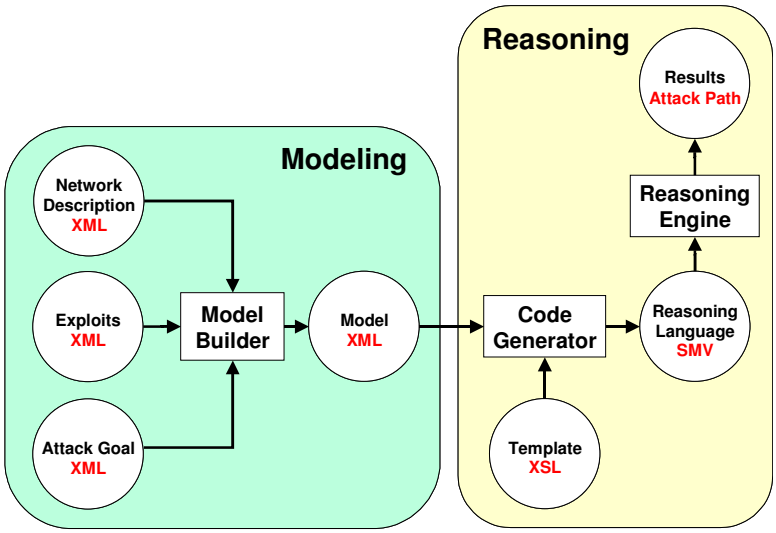

Figure 5 – Architecture for combinatorial vulnerability analysis.

In our implementation of combinatorial vulnerability analysis, attack models are specified in the Extensible Markup Language (XML) [1]. XML is a license-free, platform-independent and well-supported web standard for structuring data. An XML Document Type Definition (DTD) [2] defines the structure of XML documents that serve as valid models for combinatorial vulnerability analysis.

From the initial model for network attack, code-generation software generates code for input to a reasoning engine. This process is controlled through a general-purpose code-generation templating language. This decoupling of the modeling phase and reasoning phase means that the reasoning language can be replaced by merely modifying the code-generation template. No change to the original model specification is necessary.

The particular templating language we employ for code generation is the Extensible Stylesheet Language (XSL) [3]. Stated most simply, an XSL style sheet describes how to display a particular type of XML document. But more generally, XSL provides general-purpose XML processing via the XSL Transformations (XSLT) language.

XSL is frequently used to render XML documents as HTML. But we apply it here in a novel way: to render XML documents as input to a reasoning engine. The reuse of XML and XSL leverages existing tools and web infrastructure. It also provides an excellent degree of tool flexibility. In particular, replacing a particular reasoning engine requires simply the development of a new XSL stylesheet.

Figure 6 shows the XML structure of network attack models. XML documents are strictly trees, but we collapse redundant structures to simplify the figure. Here boxes show XML elements, with XML attributes appearing beneath the element names.

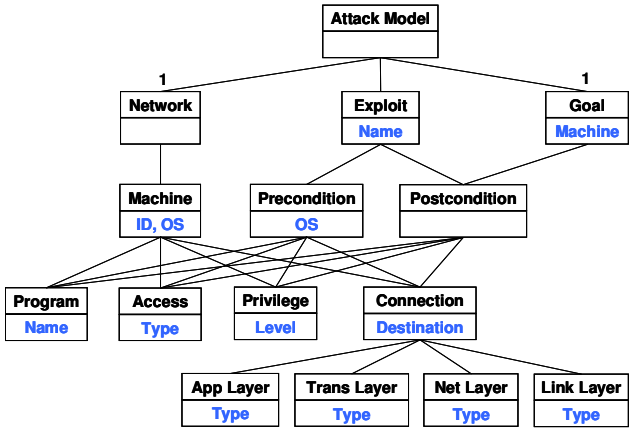

Figure 6 – Structure for network attack models.

The element *Network* specifies the initial configuration of the network under analysis, before any attacker actions. Various existing tools for vulnerability scanning and network discovery can automate the gathering of this information.

The *Network* element is comprised of some number of *Machine* sub-elements, representing machines within the network. Each *Machine* element has sub-elements for installed programs, access types (e.g. interactive or file transfer), user privileges, and connections to other machines. Connections are specified at various levels, i.e. link layer through application layer.

The model element *Exploit* is an action that the attacker can take to diminish network security. In the modeling phase, a network security analyst crafts exploits that model atomic attacker actions. This includes exploits for reported vulnerabilities, taken from archives such as Bugtraq [4]. It also includes attack-expert exploits that have the potential to incrementally advance an attack, in the same way that an actual attacker would. The model element *Exploit* has sub-elements *Precondition* and *Postcondition* for exploit rule pre- and post-conditions. These are Boolean expressions of the various network elements.

The model element *Goal* is the assumed attack goal. During the reasoning phase, our approach reports whether the attack goal can ever be reached, and if so, the sequence of exploits that lead to it. It is defined as the post-condition of a particular exploit on a given machine.

4. REASONING ABOUT NETWORK-ATTACK MODELS

Given that we have a security model of a particular network, how do we proceed to reason whether an attacker can successfully attain a certain attack goal? And if the attack is successful, what steps would the attacker need to take?

Out approach models exploits as rules with pre-conditions and post-conditions. The conditions are essentially atomic facts, and the application of one exploit typically establishes facts that are the pre-conditions of other exploits. This general fact-and-rule framework fits well with the way in which hackers attack actual networks; each successful exploit by the attacker leaves the target network vulnerable to a new set of exploits.

There are a variety of possible reasoning mechanisms to determine which attacks are possible. Our current strategy is to use, to the extent possible, existing off-the-shelf tools to solve the reasoning problem. The rationale is that developing a custom reasoning tool for an initial prototype is not the most effective approach.

With our approach, we are essentially faced with a verification problem. In particular, we are trying to verify if a network is immune to attack, at least to the extent of the completeness of the network model. The research community has two basic approaches to solving verification problems. These two approaches are explicit enumeration, implemented by model checkers, and the deductive inference, implemented by theorem provers.

The chief advantage of model checking [5] is its "automatic" aspects: after describing a model, one simply runs the model checker – essentially without human interaction. An added bonus of the model checking approach is that if a particular conjecture is false, a model checker automatically produces an explicit counterexample. In our case, such a conjecture addresses the security of some host in the network. That is, we conjecture that the network is in some way secure. The counterexample shows why the conjecture is false, i.e. why the network is insecure. Thus the counterexample is effectively an attack path.

Although model checking began as a method for verifying hardware designs, there is growing evidence that model checking can be applied with considerable automation to specifications for relatively large software systems [6]. The increasing usefulness of model checkers for software systems makes model checkers attractive targets for use in aspects of software development other than pure analysis, which is their primary role today. Model checkers are desirable tools to incorporate because they are explicitly designed to handle large state spaces [7] and they generate counterexamples efficiently.

There are a variety of model checkers in the community, but two in particular have achieved broad usage and stability in their respective domains. These two are the SMV model checker [8]], which evaluates conjectures in computational tree logic, and the SPIN [9] model checker, which evaluates conjectures in linear temporal logic. Although these two logics are strictly incomparable in expressive power, for our purposes either is more than satisfactory. This is because network security conjectures tend to be simply invariants, e.g. the attacker never obtains root privileges on a particular host.

The chief problem with model checkers is the state explosion problem. That is, models can become so large that the model checker simply cannot finish its computations in a reasonable amount of time. There are a variety of "tricks" that are used to control the state space, and we may take advantage of various characteristics of the network security problem to control the state space for our problem. For example,

most of the activity of the attacker is monotonic, i.e. the attacker typically does not have to lose one privilege to obtain another.

The basic search algorithm for the SMV model checker is breadth first, and the basic search algorithm for the SPIN model checker is depth first. The consequence is that counterexamples in SMV tend to be shorter, and counterexamples in SPIN can easily be unnecessarily long. This argues in favor of SMV, since attack the network administrator will better understand attack paths if they contain only the significant actions taken by the attacker. In fact, SMV has been applied in early work at George Mason University in this area [10].

We expect to encounter the typical scaling problems associated with model checking. But we also have confidence that useful and interesting networks will ultimately fit inside the SMV tool. If this proves false, we are prepared to look at more powerful inference technologies.

However, we don't want to approach logic programming or theorem proving unless really necessary, since these approaches tend to be much more user intensive, and require significant expertise on the part of the network administrator. A further disadvantage of these alternate approaches is that explicit counterexamples are not typically produced. It is worth reiterating that our modular architecture supports automatic language generation for whatever reasoning engine is chosen.

5. EXAMPLE APPLICATION

This section shows attack paths discovered in an example application of or approach. Here the attack goal is to obtain super user shell access, which is the most devastating access level an attacker can gain.

Figure 7 shows the network architecture for the example application. The network under attack is protected by a firewall that permits all outbound traffic but blocks all inbound traffic except for the domain name service (DNS). Here we assume that the *Fred* machine is vulnerable to a hypothetical remote buffer overflow attack in DNS BIND.

Figure 8 shows the attack paths discovered for this example. While *Fred* is the only machine running the vulnerable version of DNS BIND, other machines can be attacked through the *Fred* using other remote buffer overflow attacks. This ability is despite the fact that the firewall blocks access to the ports associated with the other buffer overflows. The details for the individual exploits in the attack appear below.

The Exploit #1 (rmt_su_bof_bind) is the hypothetical buffer overflow exploit, modeled after existing exploits for some versions of BIND. The attacking machine for this initial exploit is *Attack*, and the victim machine is *Fred*. The exploit pre-conditions are that the attacker has shell access on his own machine, the program for exploiting the vulnerable version of BIND exists on attacking machine, and the the attacking machine has connectivity to the victim machine's BIND service through the firewall. The post-condition of this exploit is that super user shell access is obtained on the victim machine.

After the ability to execute programs on a remote machine has been obtained, Exploit #2 (rcp_download) is applied to download a "root kit." In this case, the attacker has configured his machine to trust the remote machine and allow this exploit. The root kit contains utilities and other tools used to escalate the privilege level to super user. The exploit pre-conditions are: execute access (the ability to run programs) has been obtained on *Fred*, the rcp program exists on *Fred*, *Fred* has connectivity to the *Attack* machine's rsh service, and *Fred* is trusted by the *Attack* machine (as specified in the *Fred*'s rhosts file). The exploit post-condition is that root kit programs are copied from the *Attack* machine.

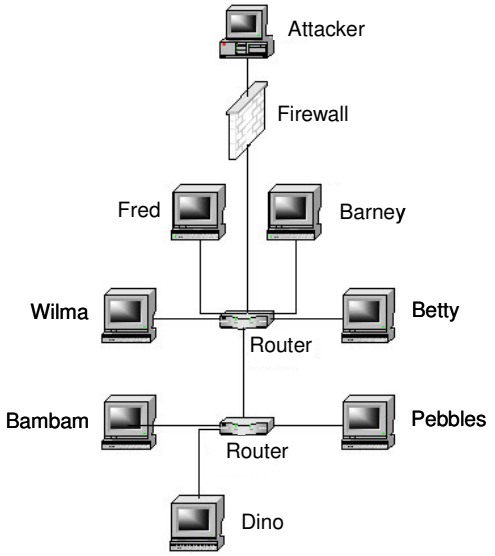

Figure 7 – Example network under attack.

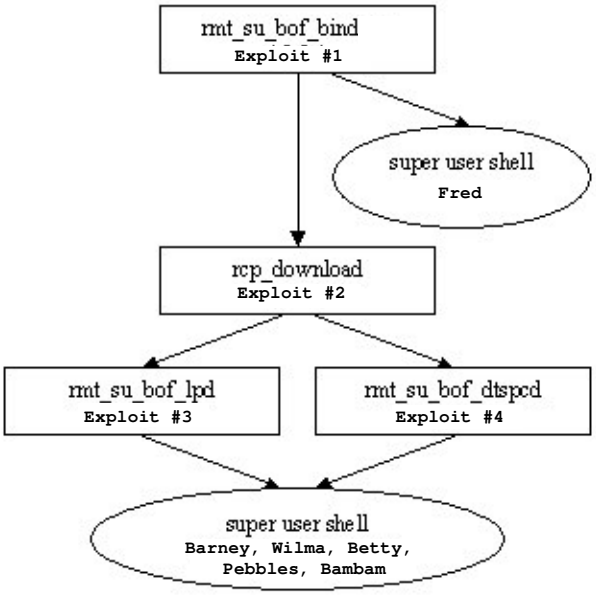

Figure 8 – Discovered attack paths.

After the root kit has been downloaded on *Fred*, the print protocol daemon, 'in.lpd' (or 'lpd'), shipped with its operating system may be exploited to allow the attacker to execute arbitrary commands with super user privileges on other hosts. This is Exploit #3 (rmt_su_bof_lpd). Here the attacking machine is *Fred*, and the victim machine is any of *Barney*, *Wilma*, *Betty*, *Pebbles*, or *Bambam*.

After the root kit has been downloaded, a buffer overflow may be exploited through the 'dtspcd' service to allow the attacker to gain administrative privileges on other hosts. This is Exploit #4 (rmt_su_bof_dtspcd),

in which the attacking machine is still *Fred*, and the victim machine is any of *Barney*, *Wilma*, *Betty*, *Pebbles*, or *Bambam*. The result of this exploit is that the attack gains super user access privileges on the victim machine.

6. SCALABILTY OF THE APPROACH

We now investigate scalability for our approach. In particular, we show how execution time increases with model size. This is particularly interesting given the known scaling problems with model checking, which we apply as a reasoning engine.

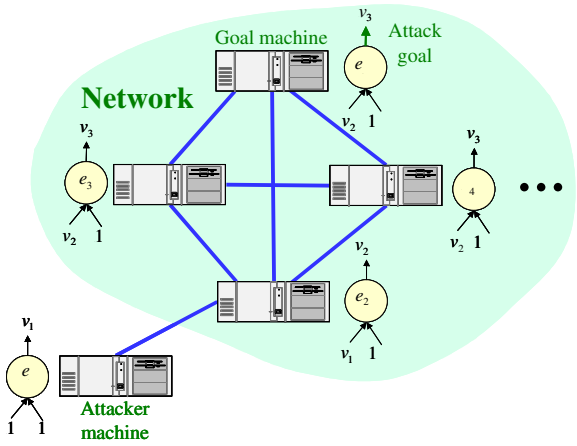

Figure 9 – Model structure for scalability experiment.

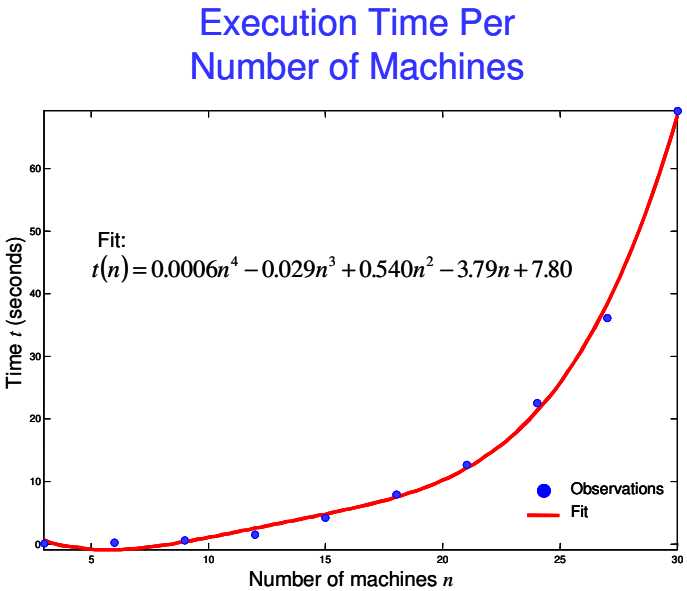

Figure 10 – Model structure for scalability studies.

Figure 9 shows the model structure that we apply for our scaling experiment. There is a single attack machine, outside the network under attack. The network under attack is fully connected. Each network

machine has a single exploit associated with it, which has two pre-conditions that must both be true, and a single post-condition.

The scaling parameter for the experiment is the number of machines in the network. In the experiment, we record execution time for increasing numbers of machines in the network. Figure 10 shows the experimental results. A fit of the measured execution times is polynomial, with negligible 4th-order and 3rd-order terms. We conclude that the execution times scale reasonably well.

We should point out, however, that memory size is the limiting performance factor in this experiment. We experienced severe memory problems for 50 network machines. In particular, 512 megabytes of memory was insufficient for this problem size. Again, this is a known problem with the model checking approach, and we will consider alternative reasoning engines in the future.

7. CONCLUSION

Our approach extends traditional vulnerability analysis by searching for sequences of exploited vulnerabilities distributed throughout a network. While vulnerabilities considered in isolation may seem innocuous, when considered in combination they may lead to serious security breaches. Indeed, our approach closely follows actual attack patterns, in which a series of exploits incrementally diminishes network security.

There are key features of our approach that provide substantial benefit to the analysis of network security. For example, the technique allows multiple attack scenarios to be tested using the same model description, for example to model an insider attack. Once the model has been constructed, it is trivial to show what an attacker can accomplish starting from a different access level on a network host.

Also, our approach automatically explores the total security ramifications of vulnerabilities accessible to an attacker. Applying our tool, it is thus easy to demonstrate why defense in depth is important in the design of network security.

REFERENCES

1. *Extensible Markup Language (XML)*, World Wide Web Consortium, http://www.w3.org/XML/.
2. *XML DTD Tutorial*, XML 101 Web Site, http://www.xml101.com/dtd/default.asp.
3. *The Extensible Stylesheet Language (XSL)*, World Wide Web Consortium, http://www.w3.org/Style/XSL/.
4. Bugtraq computer vulnerability archive, http://www.securityfocus.com/.
5. E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, Cambridge, MA: MIT Press, 2000.
6. W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin, "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering*, 24(7), pp. 498-520, July 1998.
7. J. Birch, E. Clarke, K. McMillan, D. Dill, and L.J. Hwang, "Symbolic Model Checking: 1020 States and Beyond," in *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January 1991.
8. *Formal Methods – Model Checking*, Carnegie Mellon, School of Computer Science, http://www.cs.cmu.edu/~modelcheck.
9. G. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, 23(5):279-295, May 1997.
10. R. Ritchey, P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2000)*, Berkeley, California, 2000.