Model-Based Vulnerability Analysis of Computer Systems*

C.R. Ramakrishnan
cram@cs.sunysb.edu
Department of Computer Science
State University of New York
Stony Brook, NY 11794.

R. Sekar
sekar@cs.iastate.edu
Department of Computer Science
Iowa State University
Ames, IA 50011.

Abstract

Vulnerability analysis is concerned with the problem of identifying weaknesses in computer systems that can be exploited to compromise their security. Most vulnerabilities arise from unexpected interactions between different system components such as server processes, filesystem permissions and content, and other operating system services. Existing vulnerability techniques (such as those used in COPS and SATAN) are based on enumerating the known causes of vulnerabilities in the system and capturing these causes in the form of rules, e.g., a world- or group-writable `.login` file is a well known vulnerability that enables one user to gain all access privileges of another user. However, the generation of the rules relies on expert knowledge about interactions among many components of the system. Issues such as system complexity, race conditions, many possible interleavings, hidden assumptions etc. make it very hard even for experts to come up with all such rules. In contrast, we propose a new *model-based* approach where the security-related behavior of each system component is modeled in a high-level specification language such as CSP or CCS. These component models can then be composed to obtain *all possible behaviors* of the entire system. Finding system vulnerabilities can now be accomplished by analyzing these behaviors using automated verification techniques (*model checking* in particular) to identify scenarios where security-related properties (such as maintaining integrity of password files) are violated. In contrast to previous approaches that mainly address well-known vulnerabilities, our model-based approach has the potential to automatically seek out and identify known and as-yet-unknown vulnerabilities.

Keywords: Vulnerability analysis, intrusion detection, network security, computer security, model checking, automated verification.

1 Introduction

Information and networking technologies are playing increasingly important roles in our infrastructures for such critical services as power generation and distribution, telecommunication, commerce and banking, and transportation. Although this change brings about several benefits, new dangers arise as well, since damage to the underlying computing and communication infrastructure can compromise the availability of critical services. Therefore it is important to secure the computing and networking infrastructures against damage due to malicious attacks or spontaneous faults. One of the first steps in securing a computer system is to understand the *vulnerabilities* in the system that can be exploited to compromise its security. In this paper, we describe a novel technique to identify vulnerabilities that arise from unexpected interactions between (apparently correct) system components.

1.1 Vulnerabilities in Computer Systems

Some system vulnerabilities arise due to errors in individual system components: *e.g.*, buffer overflow attacks are aimed at memory errors in server processes [3, 1]. However, a majority of vulnerabilities arise due to interactions among several components such as the operating system kernel, file system, server processes, etc. For instance, consider a vulnerability that existed in early versions of the `fingerd` service [1]. In servicing a query "finger `username`," this program needs to read a file named `.plan` in the home directory of the user `username`. A malicious user u could symbolically link a file f as his/her `.plan` even if the user has no read access to f. The user can then read the file f by simply running `finger` u, since the `fingerd` server ran with root privileges. The vulnerability here arises due to the interaction between the way the finger server operates and the way the filesystem implements symbolic links. As a second example, consider the

*This research is supported partially by DARPA-ITO under contract number F30602-97-C-0244 and by the NSF under grants CCR-9705998 and CCR-9711386.

vulnerability involving the mail notification program **comsat**, which waits for reports of incoming mail for any user and prints the first few lines of the message on the terminal in which the user is logged on. This terminal is determined from the file `/etc/utmp`, which was configured to be world-writable. A malicious user could modify this file by substituting the `/etc/passwd` in the place of the terminal that he/she is logged on. The user then sends mail to self containing a line that starts with `root::0:0:`. Of particular significance here is that the second field in this line, which corresponds to the password field, is empty, which implies that the superuser has no password. The `comsat` program promptly overwrites the password file with the message. The user can now login as `root` without providing a password. This second vulnerability also arose from an interaction among several components: the `comsat` program that assumed the correctness of `/etc/utmp` file, the file system (specifically, the permission settings on this file), and the mail delivery program. Once again, each of the components seems to exhibit "reasonable behavior," but their combination would permit a malicious user to compromise system security.

More generally, many vulnerabilities arise from unexpected interactions between different components, violation of hidden assumptions, improper setting of system parameters and configurations, etc. Use of good software engineering practices has the potential to eliminate some of these vulnerabilities, but given the fact that new vulnerabilities continue to surface in UNIX server programs that have been operational for well over a decade, we clearly need alternative mechanisms to guard against vulnerabilities. Consequently, several recent research efforts have focussed on *vulnerability analysis* and *intrusion detection* techniques (which detect misuse by run-time monitoring) [2, 10, 12, 13, 14] as a retrofit approach to secure existing systems.

1.2 Vulnerability Analysis: State of Art vs. New Approach

Research efforts in vulnerability analysis have focussed primarily on identification of *configuration errors* such as improper file permission settings. Existing approaches [11, 4, 21] can be broadly characterized as *rule-based,* i.e., they employ a set of rules that enumerate known causes for vulnerabilities. The tools then systematically check the system configuration to identify if these causes are present in the system. For instance, a world- or group-writable `.login` file is a well known vulnerability that enables one user to gain all access privileges of another user. Widely used tools, such as COPS and SATAN search for occurrences of such known vulnerabilities [11]. However, the generation of the rules relies on expert knowledge about interactions among many components of the system. Unfortunately, few experts have a complete understanding of the interactions among all components of modern computer system. Issues such as race conditions, many possible interleavings, hidden assumptions etc. [5] make it very hard for humans to come up with all such rules.

In contrast, we propose a new *model-based* approach for vulnerability analysis of computer systems. In our approach, the security-related behavior of each system component (such as the filesystem and other subsystems of the OS, system startup processes and other application processes) is modeled in a high-level specification language. These abstract models can be provided manually or synthesized using existing program analysis techniques from the source code of each component (if available), or can be supplied by the vendor for that component. Any possible security-related behavior of the entire system can then be obtained by composing these component models. System vulnerabilities can therefore be identified by analyzing these behaviors using automated verification techniques, *e.g.*, model checking [8, 17, 9], for deriving scenarios where security policies (such as preventing any user from usurping the privileges of another user) are violated. In this paper, we describe a vulnerability analysis method that we are currently developing following the model-based approach.

2 Modeling Security-related Behaviors of Systems

The first step in our vulnerability analysis approach is to develop abstract models of the security-related behaviors of various system components. In this section, we first describe our model for a small subset of a UNIX-based system that enables us to capture the vulnerability due to **comsat**. This subset captures a simplified view of the file system, mailer program and the comsat server itself. We will later extend this model to accommodate several other aspects, such as symbolic links, printing services, etc.

Although it is clear that the modeling language must have facilities to describe concurrent processes that manipulate structured data, it is currently less obvious what features will enable us to directly describe

complex processes. In the following, for the purpose of illustration, we choose a value-passing language based on Milner's CCS [16] augmented with algebraic datatypes (in the form of Prolog terms), a syntactic variant of the language described in [18]. In CCS, a system is viewed as a collection of *processes* that can communicate (and synchronize by communicating) along *channels*. Processes may be combined sequentially (using the '∘' operator), or in parallel (using the '||' operator). Nondeterministic actions can be captured using the choice operator, '+'. Processes can have parameters that represent the state of a process, or may be arguments to initialize the process. Recursion is the primary means of iteration in this language. In addition, we introduce helper functions (units of pure computation) and other features that are needed to simplify our models.

We begin with a model of the file system `fs` as shown below. The file system state is modeled as a list of records. A record of the form `content(F,C)` captures the fact that the file `F` contains `C`; a record of the form `perm(F,U,P)` captures that user `U` has permission `P` (one of 'r', 'w' or 'x') on file `F`. To simplify the presentation, the model shows only read or write operations on files, but no file permission changing operations. All these operations are accomplished by sending an appropriate message to the `fs` process. The notation `C?M` is used to denote the reception of message `M` on channel `C`. Similarly, `C!M` denotes sending a message `M` on channel `C`. We use the convention that all variable names start with an uppercase letter, while the names of processes and constants start with a lowercase letter.

```
fs(S)::=  write?(U, F, C)
             ∘ if access(S, U, F, 'w')
                  then fs(add_record(S, F, C))
                  else fs(S).
          ||
          read?(U, F, Chan)
             ∘ if (exists(S, F) && access(S, U, F, 'r'))
                  then read_record(S, F, R) ∘ Chan!R else Chan!error
             ∘ fs(S)
```

We then model the behavior of a mailer program as follows. Again, for simplicity, we do not model a mail server, but treat it as if the mail sending program directly updates a file that corresponds to the recipients mailbox. Note that we use a list to represent file names, e.g., the name "/etc/passwd" would be represented as the list `['etc','passwd']`.

```
mailer::=  send?(Sender, Receiver, Msg)
              ∘ write!('root', ['var', 'spool', 'mail', Receiver], Msg)
              ∘ mailer
```

Note that the mailer program needs to update a file that is typically owned by the message recipient, and the the sender does not have write permission on that file. As such, the mailer program runs with superuser privileges.

Finally, we model the `comsat` program and the user behavior as follows.

```
comsat(CChan) ::=   read!('root', ['var', 'spool', 'mail', Receiver], CChan)
                       ∘ CChan?Msg
                       ∘ if (Msg ≠ error)
                            then read!('root', ['etc', 'utmp'], CChan)
                                 ∘ CChan?(Receiver, Tty)
                                 ∘ write!('root', Tty, Msg)
                       ∘ comsat(CChan)
        user ::=   ( write!(U, F, C) + send!(U, Receiver, Msg) ) ∘ user
      system ::=    user || comsat(cchan) || mailer || fs(initState)
```

The comsat program waits for a message to be delivered to the `/var/spool/mail` directory. This is done by sending a request to the filesystem for a file in this directory. If the file exists, then `fs` would respond with the contents of the file in the channel `CChan`. Then comsat would proceed to read the `/etc/utmp` file to obtain the terminal on which the recipient of the message is logged on, and print the message on this terminal, and then proceed to do the same thing all over. (Note that in our simplified model, comsat may announce the same message many times.) We point out the use of unbound variable `Receiver` in the first line of `comsat`.

Such variables are treated as being existentially quantified— in particular, the read operation will read any one file `f` in the `/var/spool/mail` directory, and moreover, further occurrences of `Receiver` will all refer to this file. (The semantics of such variables is similar to that of variables in logic programming languages.) Use of such variables enables us to develop a very natural specification of a user: the user may write an arbitrary file with arbitrary content, or send an arbitrary message to an arbitrary user.

The behavior of the entire system is captured as a parallel composition of the comsat, mailer, filesystem and user interactions. The user interactions are captured by the `user` process, which nondeterministically writes arbitrary files and sends mail to arbitrary users. Once again, we use logic variables to denote that the target files and message recipients may be arbitrary.

3 Detecting System Vulnerabilities

Once we have modeled a system, we can analyze all system states that can be reached by some execution of the system. In the simplest case, the security violations can be described as propositions on states, for instance, labeling states as safe and unsafe. In the simple model described in the previous section, we can label those states where the content of the password file have been modified as unsafe states (since an ordinary user may be able to obtain superuser privileges by such modification[†]). In general, however, we will need to label entire execution sequences, rather than individual states, as safe or unsafe. For instance, in a more complex model of the system, an unsafe execution sequence will be one where a password file is modified without involving the `passwd` program, or even more generally, when the password of a user is changed by someone different from the user and the superuser. We can use formulas in *temporal logics* [15] to express such path-based properties, such as ordering relationships among events.

Temporal logics have been extensively used for describing correctness properties of concurrent systems such as hardware and communication protocols. For finite-state systems, temporal properties can be verified with respect to a system specification using *model checking* techniques [8, 17, 9]. One of the imporatant features of model checking techniques is their ability to generate counterexamples: sequences of states that lead to violation of the property to be verified. In our application, the counterexamples correspond to the steps that an attacker can use to exploit system vulnerabilities. It should be noted that, in contrast to typical applications of verification techniques, we are interested in generating *all* counter examples in the case of vulnerability analysis. This is because we may not be able to fix all of the vulnerabilities, as some of them may depend on bug-fixes to vendor software. In such cases, we have to rely on *intrusion detection*, which is a runtime monitoring technique that detects attempts to exploit the vulnerabilities. The traces produced by a model checker correspond directly to the exploitation attempts, and thus, getting an exhaustive list aids in the development of effective intrusion detection systems. One of the problems in this context is that there may be many attack scenarios that are equivalent. Since a human may have to ultimately sort through these scenarios, it is necessary to present only a minimal set. Identifying such a set of scenarios that cover the rest is a research issue.

A second problem in our case is that the systems we consider are infinite-state, and hence we cannot readily apply the available model checking techniques. There are two main ways to ensure termination of the the model checker in spite of the infiniteness of its search space. One of them is to use *abstraction* to map the infinite-state model checking problem to an "equivalent" finite-state one. Another is to modify the search strategy to ensure that only finite portions of the infinite state space will be explored. The soundness and completeness of these strategies will depend on the property of interest. For instance, in the case of vulnerability analysis, we are interested in generating all sequences of state transitions that lead to an unsafe state. If we make use of a strategy that performs bounded-depth search, then the strategy will be "incomplete" in that it will not generate scenarios longer than the depth parameter. This limitation is quite reasonable in our case if we assume that the system vulnerabilities will be exploited by human attackers using their intuition and expertise to come up with attack scenarios. This implies that the sequence of actions that they would perform to achieve intrusion will typically be a short sequence, and thus it may be acceptable to miss out vulnerabilities that require long sequences of actions. Based on this assumption, our method uses a search procedure with iterative deepening, stopping the search after a predetermined depth.

[†]Note that the simplified model has no way for the password file to be changed legitimately; thus, all modifications are to be treated as attacks.

In our implementation, we first translate the high-level models of the system into Prolog facts. The model checking procedure is implemented as a meta interpreter (over these facts) in Prolog, evaluated using tabled resolution [19, 7] in XSB [20]. We use the power of logical variables to succinctly capture system variables with as-yet-unknown values. With unification, which binds these variables only when neeeded, we ensure that case-splits are done lazily. We augment the specification language with constructs to annotate the trace, so that the final traces are human-readable.

Although model checking possesses several advantages over other techniques for vulnerability analysis, it also has the drawback of being potentially slow. To speed up the process of verifying system security when some aspect of the system state changes (e.g., file permissions are changed), we can use the model checker to generate simple rules that capture conditions for the existence of vulnerabilities. Using our approach, we can accomplish this by leaving the initial state of the system unspecified, i.e., a variable. State changes are reflected by the instantiations to the variable corresponding to the initial state. When a compromised state is reached, the instantiations spell out the conditions on the initial state that can lead to intrusions. We can capture these conditions in the form of rules, which can then be used to perform faster vulnerability analysis. However, the rules may need to be regenerated the system's capabilities change, (e.g., addition of new services, or change of configuration of existing services). Development of incremental techniques for determining the validity of rules in the presence of system changes, and regeneration of minimal rule sets are topics of future research.

4 Preliminary Results

Given the simple model of a UNIX system described in Section 2, our current implementation identifies the following vulnerabilities that would ultimately enable the password file to be overwritten. The vulnerabilities are presented in the format

```
when <vulnerability>
<exploit scenario>
```

The first vulnerability, which was described earlier in the introduction, happens when a user `U1` has permission to write the `/etc/utmp` file.

```
when [perm(U1,[/etc,/utmp],w)]
[[U1, writes, [/etc,/utmp], with, (U2, [/etc,/passwd])]
[U3, sends msg, M, to, U2]
[comsat sees mail for, U2, reads,(U2,[/etc,/passwd]),from /etc/utmp,
          overwrites, [/etc,/passwd], with message, M,)]]
```

The second vulnerability is similar to the first one, but rather than using mail delivery as the mechanism to update the spool file, the user directly overwrites the spool file in this case.

```
when [perm(U3,[/etc,/utmp],w), perm(U1,[mailDir,U2), w)]
[[U1,writes,[mailDir,U2],with,M]
[U3,writes,[/etc,/utmp],with,(U2,  [/etc,/passwd])]
[comsat sees mail for, U2,  reads,(M  ',',  [/etc,/passwd]),from /etc/utmp,
          overwrites, [/etc,/passwd], with message, M)]]
```

Finally, the trivial case when the password file is world-writable:

```
when [perm(U1,[/etc,/passwd],w)]
[[U1,writes,[/etc,/passwd],with,M]]
```

4.1 Adding Symbolic Links

We enhanced the model to introduce symbolic links so as to capture intrusions that are based on the fact that many programs perform insufficient permission checking on such files. In the new model, content of files may be `normal(C)`, which denotes an ordinary file with content `C`, or `link(F)`, which denotes a symbolic

link to file F. File access functions are updated to resolve symbolic links. With this change, the following additional vulnerabilities are identified by our system. Neither require write access to the utmp file, but rely on write permission on the spool file. By symbolically linking the spool file to the utmp or password file and sending a message to self, one can overwrite the password file:

```
when [perm(U1,[mailDir, U2],w)]
[[U1,symlinks,[mailDir,_U2],to,[/etc,/utmp]]
[U3, sends msg, (U2,[/etc,/passwd]), to U2]
[comsat sees mail for U2,   reads,(U2, [/etc,/passwd]),from /etc/utmp,
          writes[/etc,/passwd] with (U2,[/etc,/passwd])]]

when [perm(U1,[mailDir, U2],w)]
[[U1,symlinks,[mailDir,U2],to,[/etc,/passwd]]
[sends msg  M to U2]]
```

Although we have not modeled the finger vulnerability described in the introduction, it should be clear that it can be captured in our model with an appropriate choice of the property to be verified.

4.2 Adding Lpr

Finally, we added a model of the printer program lpr to our system. This program first copies the file to be printed to a spool directory, and then prints it. It cycles through a fixed, finite set of names for files in the spool directory. (In the model, we have set the size of this set to 1, but it could be any small number.) To avoid copying large files, this program offers an option to symbolically link a file rather than copy it over. This combination of features introduces the following vulnerability:

```
when [perm(U1,F1,w),perm(U2,/etc/passwd,r),perm(U3,F1,r)]
U1 writes file F1
U2 prints file /etc/passwd with option -s
U3 prints file F1
```

Since we start with an initial state that corresponds to an unbound variable, there are no files that can be printed in the initial state. The assumptions in the first line above simply indicate that such a file can be created, and later read. It also requires read permission on the password file since that is needed for printing it.

5 Automating Model Generation from Implementations

So far we have assumed the existence of models describing behaviors of the various components. Recall that rule-based systems depend on expert knowledge of the behavior of the composite system. In contrast, the model-based approach requires description of individual system components one at a time, and each component itself often very well understood. In some cases, when source code is available, we can deploy available program analysis technology to automate the generation of these models themselves. For components without source code, we assume that the vendors will provide the needed model.

Based on control-flow analysis of programs, we can automatically build conservative models that cover all executions of the program. We plan to first build analyzers to extract control flow of shell scripts, and use the control flow information to construct an execution model. For instance, consider the fragment of /etc/rc.d/rc script from Linux kernel 2.0.30:

```
runlevel = $argv1
for i in /etc/rc.d/rc$runlevel.d/S*; do
   ...
   $i start
```

Since argv1is a runtime argument, we conservatively assume that runlevel can take any value. By the semantics of the for statement, we can determine that i takes values that match the regular expression

`/etc/rc.d/rc*.d/S*`. Given the current configuration of the system, we can instantiate `i` to every file that matches `/etc/rc.d/rc*.d/S*`. We then construct models for each of these scripts, and compose them in order to obtain a model for `/etc/rc.d/rc`. The same approach can be used to construct execution models for programs written in other languages, such as C, C++ and Perl. It should be noted that, for security related behaviors, we need to infer the control and data flow only to know the sequence of system calls are made, and files accessed. From the modeling experience we have aquired thus far, it appears that the current program analysis techniques are powerful enough to infer the above information accurately.

6 Summary

The salient advantages of the approach described in this paper are:

- Our model-based approach has the potential to automatically seek out and identify known and *as-yet-unknown* vulnerabilities. In contrast, the rule-based approaches are limited to examining the system for known vulnerabilities. By composing the models of subcomponents, including configuration scripts, we derive a model of the entire system and hence can detect authorization leaks that arise from "deeper" faults (e.g., programs that themselves are invoked from `.login` or scripts that get executed at boot time from the `rc` files). It should be noted that most system programs are controlled by configuration scripts, and it is cumbersome (if not impossible) to introduce new rules that check for vulnerabilities associated with each configuration. By treating configuration scripts as any other system component and deriving abstract execution models for them, the model-based approach can be applied to detect authorization leaks in any system component— from boot-time processes and login processes (local as well as network) to application-specific processes such as CGI scripts.

- Our approach can be used for troubleshooting a system, as it not only tells us whether there is a problem, but can generate concrete scenarios that exhibits the vulnerability. To aid in troubleshooting, we ensure that the scenario set is in some sense minimal.

- We can use our technique for *automatic generation* of a rule base that can be used to perform vulnerability analysis more efficiently. To accomplish this, we perform perform model checking with incomplete information about initial system configuration. The model-checking process produces vulnerability scenarios that are conditional upon the (unspecified) initial configuration. Each of these conditions then capture a potential vulnerability that can be checked using a rule.

- Our analysis can be coupled tightly with intrusion detection systems. Note that intrusions are effected by exploiting vulnerabilities in the system. *Misuse intrusion detection* techniques are based on specifying rules that capture the exploitation of vulnerabilities. Specifically, for each vulnerability identified by our analysis, we can either reconfigure the system to eliminate the vulnerability, or when this is not feasible[‡], we can generate rules for use in an intrusion detection system. In this manner, intrusion detection mechanisms can then be adapted to recognize any action that threatens to exploit a system vulnerability, and not just those that are *known* (by past experience or expert knowledge) to be potentially dangerous.

The method sketched in this paper is a first step towards applying a model-based approach to vulnerability analysis. For the approach to be used in practice, we need to design a modeling language with constructs that permit succinct description of real-life systems, strengthen the model checking techniques to handle the new constructs as well as to infer minimal sets of attack scenarios.

References

[1] 8lgm Security Advisories, `http://www.8lgm.org/advisories-f.html`.

[‡] In many cases, we may not be in a position to rectify a problem either because the rectification will interfere with proper operation of other system components, or because the fix involves a vendor-provided software in binary form.

[2] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[3] T. Aslam, I. Krsul and E. Spafford, A Taxonomy of Security Faults, Proceedings of the National Computer Security Conference, 1996.

[4] R. Baldwin, Rule based analysis of security checking. MIT LCS Technical Report No. 401, 1988.

[5] M. Bishop, M. Dilger , Checking for Race Conditions in File Access". Computing Systems 9(2), 1996, pp. 131-152.

[6] M. Bishop and B. Bailey, A critical analysis of vulnerability taxonomies, TR CSE-96-11, Dept. of Comp. Sci., University of California at Davis, 1996.

[7] W. Chen and D.S. Warren, Tabled evaluation with delaying for general logic programs, Journal of the ACM, 43(1):20–74, January 1996.

[8] E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, Proceedings of the Workshop on Logic of Programs, LNCS 131, 1981.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM TOPLAS*, 8(2), 1986.

[10] D. Denning, An Intrusion Detection Model, IEEE Transactions on Software Engineering, Feb 1987.

[11] D. Farmer and E. Spafford, The COPS Security Checker System, CSD-TR- 993, Department of Computer Science, Purdue University, 1991.

[12] C.C.W. Ko , Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach", Ph.D. Thesis, Department of Computer Science, University of California at Davis, August 1996.

[13] C. Ko, G. Fink, and K. Levitt, Automated detection of vulnerabilities in privileged programs by execution monitoring, Computer Security Application Conference, 1994.

[14] T. Lunt, A survey of Intrusion Detection Techniques, Computers and Security, 12(4), June 1993.

[15] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1991.

[16] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[17] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[18] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Terrance Swift, S.A. Smolka, and D.S. Warren, Efficient model-checking using tabled resolution, Proceedings of CAV'97, June 1997.

[19] H. Tamaki and T. Sato, OLDT resolution with tabulation, International Conference on Logic Programming, pages 84–98. MIT Press, 1986.

[20] The XSB logic programming system v1.8, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog`.

[21] D. Zerkle, K. Levitt , NetKuang–A Multi-Host Configuration Vulnerability Checker, Proc. of the 6th USENIX Security Symposium. San Jose, California, July 22-25, 1996, pp. 195-204.