Bond University
ePublications@bond

Information Technology papers | Faculty of Business

7-1-2002

Using model checking to test a firewall : A case study.

Padmanabhan Krishnan
Bond University, Padmanabhan_Krishnan@bond.edu.au

Danita Hartley
University of Canterbury, New Zealand

Follow this and additional works at: http://epublications.bond.edu.au/infotech_pubs

Recommended Citation
Padmanabhan Krishnan and Danita Hartley. "Using model checking to test a firewall : A case study." Jul. 2002.

http://epublications.bond.edu.au/infotech_pubs/12

This Conference Proceeding is brought to you by the Faculty of Business at ePublications@bond. It has been accepted for inclusion in Information Technology papers by an authorized administrator of ePublications@bond. For more information, please contact Bond University's Repository Coordinator.

Using Model Checking to Test a Firewall : A Case Study

Padmanabhan Krishnan
School of IT, Bond University
Gold Coast, Queensland 4229, Australia
Email: pkrishna@bond.edu.au
Fax: +(61)(7)5595 3320

Danita Hartley
Department of Computer Science
University of Canterbury,
Christchurch, New Zealand

Abstract

This paper summarises our experience in using model checking technology to test concurrent programs. We use the tool Verisoft to understand various aspects of a firewall tool kit by instrumenting three components of the firewall tool kit with hooks to test their behaviour. Some of the key changes include changing socket communication to message passing queues and adding appropriate synchronisations so that the behaviour of the system can be tracked. We aim to minimize the number of changes to the original source code so that its original behaviour is not affected. The main conclusion is that it is possible to inspect source code with a view towards verifying key behavioural properties without understanding the entire behaviour of the system.

1 Motivation

It has been argued if a practitioner can use formal methods, it will increase the reliability of software [6]. A concrete case study is described in [7] in which a simple experiment showed the benefits of formal methods. Within formal methods, techniques such as model checking [1] have become off the shelf technology, although this is mainly in the context of hardware verification. This is because the requirements for successful use of model checking includes the availability of a finite state model (not exceeding a certain size) expressed in a specification language. Software systems do not usually meet these requirements as they are usually much larger than typical hardware circuits and are written in languages such as Java or C. The use of both standard and user defined libraries also imposes a limitation. Thus complete model checking of Java or C programs is not widespread. Model checkers such as Bandera [3] use pre/post conditions to verify key properties exist. They build a finite state model of the program by performing static analysis such as slicing and abstraction.

A different approach is adopted by Verisoft [5]. It does not try to compute the complete state space of the program. It instead analyses the complete run-time behaviour of a program upto a given depth. That is, all states reachable from an initial state up to a given depth are examined. Therefore Verisoft can be used to perform to examine the properties of all the states up to a certain depth.

In this paper we describe a case study which uses systematic state space exploration to understand complex programs with a view towards independent product examination. We play the role of an *independent* body which can examine the source code, run the system to test it. That is, we are neither the authors of the system to be checked nor the authors of the tool(s) used in the process of verification. While the process of verification could require changes to the source code, for example, to enable testing, we aim to minimize such changes. The main aspect we focus on is verification of key behavioural properties without necessarily understanding all the components of the system. We describe the techniques and effort required by a third party to assure key properties, by examining the source code of a software system.

While we focus on the analysis of a system which has been fully developed, it is possible to use Verisoft to ensure correctness during the development process. Verisoft can be used to test the individual modules as well as appropriate subsystems.

A brief overview of the paper is as follows. The next section describes the firewall software we are analysing, and our analysis method including the type of examination conducted. A brief overview of the features of Verisoft is also presented. Owing to space limitations, we assume that the reader is familiar with Unix including the process system and sockets. In Section 3 we describe in detail the steps involved in instrumenting the analysis with Verisoft, so as to analyse the behaviour of our chosen firewall system. Section 4 draws some conclusions and suggests how to improve the process of code inspection.

2 System and Tools

The system we chose to analyse is FWTK (FireWall ToolKit) system from Trusted Information Systems (TIS) `http://www.tis.com/research/software`. TIS makes available the sources free of charge for educational and research purposes. Hence it is possible to study the existing code as well as change the code to instrument the analysis. FWTK is organised as a collection of programs including access control (`netacl`), authentication (`auth`), and application programs such as FTP `ftp-gw`, telnet `tn-gw` etc. This organisation makes it easier to inspect the various components independently. These programs are supported by a common library consisting of approximately 2400 lines of C code. In fact, the FWTK documentation states that it is designed to be verified for correctness as a whole or at a component level. FWTK is essentially a concurrent system with processes communicating via sockets. Being a firewall system, it is easy to state some of the key properties that the system should satisfy. For example, safety properties include that a user from a banned host must not be allowed to login. We now describe the key aspects of the access control mechanism `netacl`, the FTP gateway `ftp-gw` and the network authentication service `authd` that are relevant to our analysis.

`Netacl` manages network access control expressed in terms of access from IP-addresses, service requested, and other information. `Netacl` is used to block access to services such as FTP and also isolate certain functions using the Unix call `chroot` (i.e., run a command with a specified root directory). It is only about 200 lines of code and, in principle, should be easily analysable. `Netacl` can be configured to allow or deny particular services. For example, the following rule states that one can invoke an FTP server (via the command `in.ftpd`) from the host with the specified IP-number, viz., `132.181.11.188`.

```
netacl-ftpd: permit-hosts 132.181.11.188
-exec /usr/sbin/in.ftpd -l
```

One of the properties we verify is that a request is accepted only if there is a suitable rule in the configuration file.

`Ftp-gw` is a proxy server for FTP and like `netacl`, permits or denies FTP commands based on IP address. It logs all bytes that are transferred for further analysis. The program can be used to run `chroot` to an empty directory. This can be used to prevent a rogue user from breaking into the firewall as their root directory has been altered. `Ftp-gw` consists of about 1000 lines of code and is larger than `netacl`. Here we do not formally verify any property. Rather, we aim to understand the interaction between the FTP proxy and its operating environment.

The authentication service component, `authd`, is an optional component of FWTK and is required if the other subsystems such as `ftp-gw` uses authentication. `Authd` essentially maintains a database of users, permissions, failed and successful accesses, etc. on a secured host. We focus on the authentication server `authsrv.c` which is about 1400 lines of code. The man page for the authentication server identifies a class of users with each command. For example, to use the `group` command the user must be authenticated as a global administrator. We verify a variety of these properties.

Verisoft is a tool that can be used to detect deadlocks and assertion violations in concurrent C programs. While it uses model checking ideas, it does not compute the entire state space of a program. It generates the relevant states up to a given depth from a given state and performs exhaustive analysis on the generated states. The C program can fork processes (up to a fixed number of times as specified in a configuration file) which may communicate via message passing (using message queues established using the Unix system call `msgget`), and control access to data using semaphores (using `semget`, `semctl` calls). One can also write Verisoft specific assertions using the construct `VS_assert(e)` where `e` denotes a boolean expression. If `e` evaluates to false, an error is signalled. `VS_print` can also be used to print various messages. Operations on message queues and semaphores, assertions and `VS_print` are classified as visible. Hence `VS_print` is different from `printf` in that the former is a hook used by (and hence visible to) Verisoft. A global state is where all processes can only execute visible actions.

The key property that one can analyse deadlocks and assertion violations by only examining the global states is exploited by Verisoft. Verisoft also provides a scheduler which can be used to control the execution of the concurrent program. This allows one to study behaviours under a variety of scheduling algorithms. Required non-determinism, e.g., due to different possible operating conditions, can be simulated by the Verisoft operation `VS_toss(n)` which returns a number between 0 and `n`. Verisoft also provides other useful features such as a graphical presentation of the visible operations being performed and the ability to find a scenario leading to an undesirable state.

3 Instrumentation and Analysis

In this section we explain in detail the analysis of FWTK using Verisoft. We begin with a few general remarks on the basic approach and present details later in the section. Our analysis is based on the notion of as needed comprehension while performing software inspection [4]. We use Verisoft to verify our understanding of the behaviour of the program, obtained from the man pages and other supporting documents. We do not try to understand the system or the code in its entirety.

As noted earlier, running a program under the control of Verisoft requires the program to be annotated with appropriate visible actions. An important aspect in transforming the code is to make as few changes to the original code as possible, thereby minimizing the probability that any new errors are introduced. As communication between processes in Verisoft is limited to message queues and semaphores, all socket communications in FWTK have to be translated using the message passing primitives. While it should be possible to keep the original socket communication and add message passing only for testing purposes, this was not the case for FWTK. The details of this will be explained when we describe the analysis. To minimize changes to the original source code, any function that requires Verisoft additions is called within a *wrapper* function. The interface to a *wrapper* function is kept identical to that of the original function. This not only keeps Verisoft code separate from the original source code, but also reduces Verisoft code duplication. Ideally the only changes required then are the replacement of particular function names with their Verisoft wrapper equivalents. This can easily be done systematically by a search and replace technique. To further separate the Verisoft additions from the original code, all such additions are kept in a separate file and used in the original source via `#include`.

For socket calls being simulated by Verisoft message passing, there are two cases to consider. In the first case, where the receive from and send to destinations are known, socket simulation is simple. For each socket, create two message queues A and B. If processes 1 and 2 communicate via queues A and B then process 1 can send data on A and receive data from B. Similarly process 2 can send data on B and receive data from A. If communication is only half duplex only one message queue is required. The second case is where the send to and receive from destinations are not known and communication is full duplex. An example of this scenario is when socket descriptors are passed as arguments to functions that can potentially handle several socket descriptors. The problem arises because a single socket identifier is translated into two message queue identifiers. In these cases the processes involved are passed only one of the message queue identifiers (the queue for receiving data in our case). The actual queue that is used to sent data is determined from the receiving queue identifier passed as an argument to the Verisoft wrapper function that does the sending. A socket write call is simulated by a Verisoft `send_to_queue` call, while a socket read call is simulated by a Verisoft `rcv_from_queue` call. When data spans several write calls, semaphores are used to force the receiving end to wait until all data has been sent to the queue before any reading from the queue commences.

Function calls that are incompatible with Verisoft need to be replaced by some simulated Verisoft equivalent. For example, the `execv` system call causes a divergence in Verisoft because it does not return to its calling process. We simulate `execv` by executing the body of the program followed by an `exit` system call. This explicitly informs Verisoft that the process in question has now terminated. There are limits to this as Verisoft does not support the unbounded creation of processes.

We now focus on the processes that need to be created to test the behaviour of the FWTK components. Each component interacts with a client and can also run as a daemon process. To keep the testing environment as close to reality as possible at least a *client* (also called the *environment*), and a *daemon* process are required. The number of *client* processes required depends on the component. In addition to the environment processes, a process called `process_assert`, that keeps track of the state of the main process, is also implemented. Its purpose is to test whether particular properties of the system hold. The process `process_assert` is synchronised with the main process via various Verisoft primitives. For example, in the authorisation server, to let the process know when a user is being added to the database the original function `auth_dbputu` (which puts an user into the database) is wrapped up in a Verisoft wrapper `vs_auth_dbputu` function which simply sends the appropriate message to `process_assert` and calls the actual `auth_dbputu` function. In certain cases, one has to check the log file to determine the result of an operation. This is because the function being tracked does not return a value but simply makes an entry in the log file. While it is possible to alter the source code to return values, we did not do so in the interests of keeping the changes to a minimum in order to understand the original program.

Simulating an environment process is simple if the interactions with the process being verified (the main process) are understood. The simulation simply consists of sending the appropriate data to the main process and acting appropriately on data received from the main process. When the choice of data sent by an environment process is nondeterministic, `VS_toss` operations are used. That is, we use external non-determinism to simulate an arbitrary environment. We now present details of the verification carried out on the three components of FWTK.

3.1 Netacl

We summarise the general behaviour of `netacl` as specified in the man pages, which forms the basis for our verification. `Netacl` runs on different ports for different applications as specified in `/etc/rc.local`. When it receives a request it checks the configuration information and determines whether the initiating host has the correct permissions. If so, the daemon runs and starts the program

specified in the configuration table. It is also possible to `chroot` to a directory and confine the user to a limited area of the file system. However, the `chroot` command can only be executed by those having super-user privileges.

In this section we describe the verification of two key properties. The first is that a connection is accepted only if a 'permit' rule occurs in the configuration file. The second property ensures that a normal user cannot use the 'chroot' option.

In order to verify these properties, two of the standard Unix system calls need to be modified. The first is `getpeername`, which, given a socket, returns the name of the host connected to it. There is no equivalent function of message queues. However, a client (the environment) process must have generated the request. Hence it is possible to obtain the address of the host from the data generated by the client process. The change in `getpeername` forced a minor change to the `hostmatch` (which determines if a given hostname matches a given pattern) function used by FWTK to verify the permissions.

The 'permit' rule verification is carried out as follows. The property that the modified `hostmatch` must have returned the value true before the required service program was executed is asserted. This required adding a Verisoft call to the function `acceptrule` (which is invoked when a request satisfies a rule) in the source. Verisoft had no difficulty in verifying this requirement for the various environments we defined.

If the *negation* of this assertion was used, Verisoft found a run that violated this assertion. A particular case involving the negation of the correct assertion is shown in Figure 1 (as generated by Verisoft) and discussed below. One deadlock (the lighter circle) and four assertion violations (the darker circles) are shown. The four assertion violations represent the four cases where the environment process is not permitted access. This is based on the behaviour coded in the environment. `VS_toss` determines which of the requests one should simulate. The top part of the diagram shows the visible set of actions performed by the processes, including the exchanged messages which lead to one of the assertion violations. That is, it shows the IP-address that is not permitted in the database and is rejected by the function `hostmatch`. The `VS_print` command is a visible action and is used to indicate which function has been invoked.

More specifically, `Process_2` (the external requester) non-deterministically chooses an IP address and sends it on queue 2 to `Process_3` (the listener on the port). This request is forwarded on queue 3 to `Process_1` (the main process that determines validity). `Process_1` determines that this host is not permitted to connect and rejects the request. The bottom part of the diagram shows the relevant state space computed by Verisoft. The visible actions that are shown above form a path in the state space graph. In general, Verisoft can be used to determine how a particular state (a point in the pruned state diagram) can be reached. One can click on the point and the sequence of visible actions (the top part of the figure) that lead to that state will be displayed.

The deadlock reported is not an error as it indicates the behaviour when a process exits, i.e., the process cannot take part in future interactions. The sequence of visible actions leading to such a deadlock can also be discovered using Verisoft. One particular behaviour has been found to lead to deadlock. This is when a request is denied and the daemon terminates as desired. This scenario is a completion of the behaviour described in Figure 1. After the request to connect has been turned down, the request sent by `Process_2` to terminate the daemon is processed by `Process_3`. This completes the discussion of the testing of the 'permit rule'.

To verify that a normal user cannot use the `chroot` option, an assertion after `chroot` call in the source was inserted. Again Verisoft had no difficulty in verifying that a user without root privileges could not execute the `chroot` option.

Towards these analyses, Verisoft explored 74 states and 73 transitions. This was based on a depth of 50. Increasing the depth did not affect the number of states as the violation is found within the specified depth.

3.2 FTP

The behaviour of the FTP proxy is complicated as it exhibits both security/authentication interactions as well as data transfer related interactions with the server. Therefore Verisoft was used to understand the various interactions rather than verify any particular property. Three scenarios aimed at convincing ourselves that the FTP proxy works as intended were used. In the first scenario, the user was not permitted to use the proxy and the program was expected to reject the connection. In the second scenario the user was not required to be authenticated, while in the third scenario the user was required to be authenticated (using `authsrv`).

Before we describe the scenarios, we outline the changes we had to make to the existing code. In the case of FTP proxy, a number of Unix system calls needed to be modified. These include `select`, `bind`, `listen`, etc. These are all related to the socket calls and they were translated to equivalent calls on message queues. Semaphores, to prevent concurrent writing or reading on the same queues, also were added. This was required when a message was split into more than one part. In principle, one can use both message queues and sockets together. That is, sockets would be used as per the original program, while message queues would be used only for Verisoft book keeping. However, this was not possible in this case as it caused a divergence error from Verisoft. This is related to the way the server

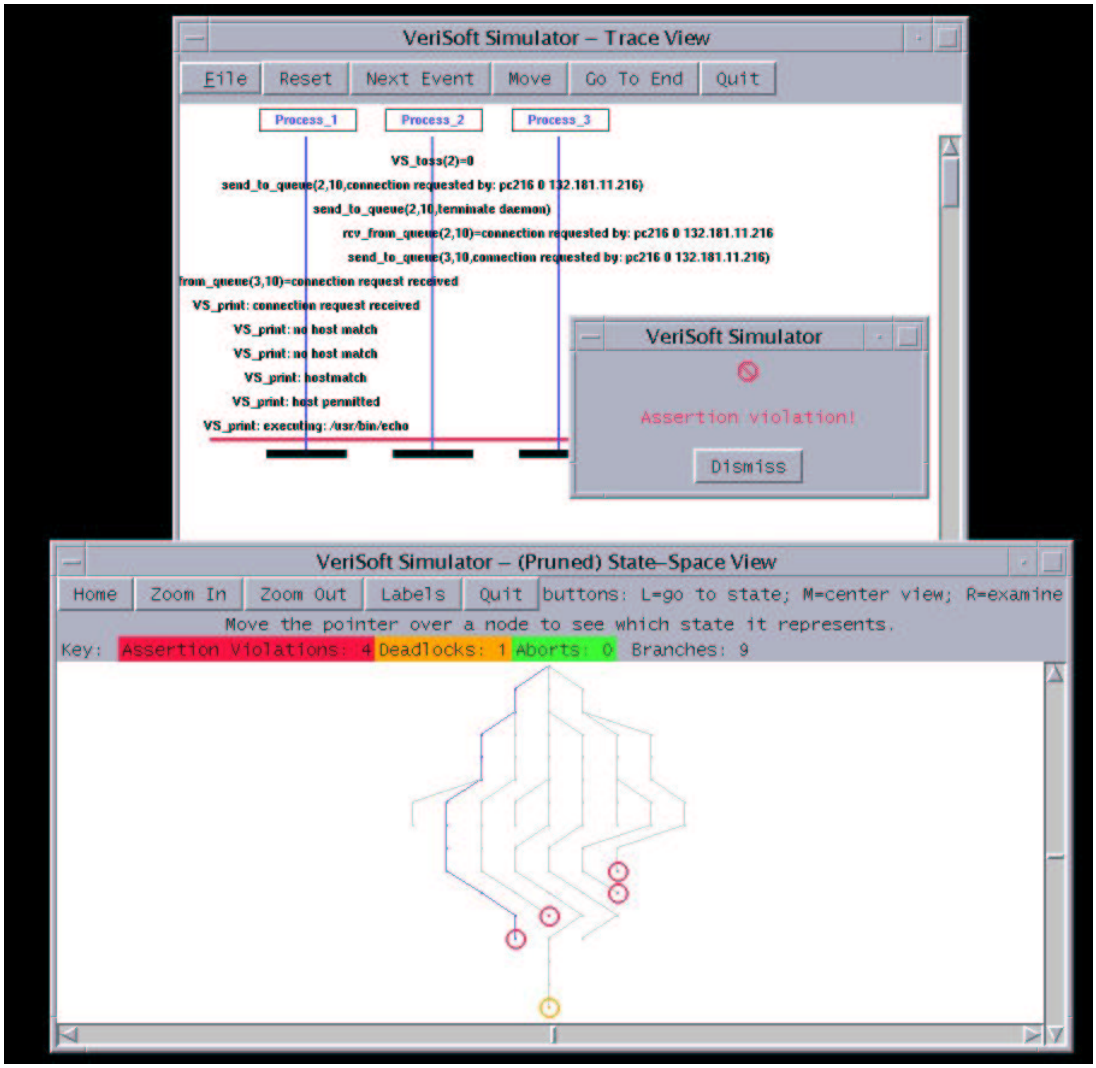

Figure 1. Verisoft detection of failing an assertion in netacl.

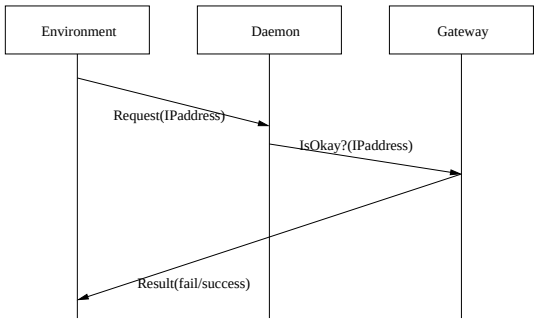

Figure 2. FTP Interaction.

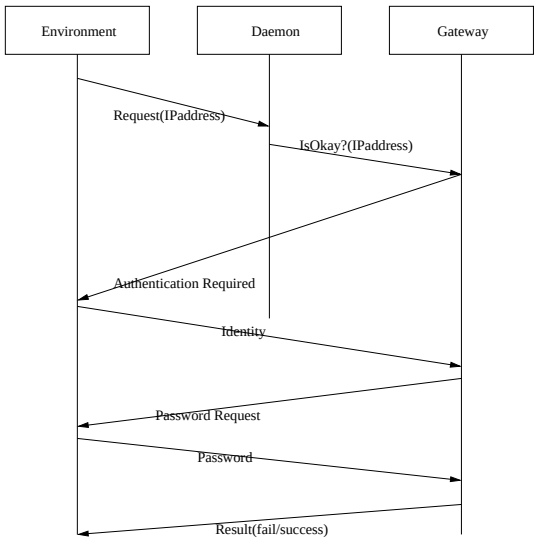

Figure 3. Verisoft scenario of user requiring authentication.

listens to sockets. As there is no known deadline on when listening succeeds, the source code was altered to use only message passing.

The general behaviour identified is shown in Figure 2. This diagram is based on the message sequence charts generated by Verisoft. In this case the server is not activated and thus takes part in no interaction. This behaviour is similar to that seen for `netacl`. If the rules do not allow the host to use the FTP proxy, the request is rejected directly. When a host is permitted, the request is accepted and the gateway is ready to connect to the FTP server. Note that for these behaviours, the server has not taken part in any interaction. However, we need to include it in our testing as Verisoft does not permit dynamic process creation and we cannot create the server process after accepting a request.

The third scenario is presented in summary form in Figure 3. After a request is accepted from a permitted host, the proxy is ready to authenticate the user. The client process simulates a user entering the system and being prompted to send a password. If the password is valid, the user is authenticated to the proxy and the proxy is ready to connect to the server. In both Figures 3 and 2 the server is not shown as is not involved in the examined behaviour. This is because we are analysing only the first phase of the interaction.

From a user's perspective this is not surprising. However, it was hard to extract this information from the source code. One of the problems was that the `ftp-gw` uses a table of pointers to functions. Depending on the interaction initiated, a suitable index is computed and the function invoked. It is quite impossible to verify by reading the code that the correct functions are invoked and the use of Verisoft allowed a reasonable certainty that this occurred. Verisoft explored 246 states and 277 transitions before diverging. Divergence occurred because once the connection was established we did not simulate the behaviour of a user actually interacting with the ftp server.

3.3 Authentication

Our final inspection deals with the authentication server. As noted earlier the `process_assert` process which maintains the state of the authentication server through messages it receives from the server was required. The process uses its knowledge of the server's state to check whether particular properties hold in the current state. To minimize changes to the server (`authsrv`), the message containing state information were sent to `process_assert` via the wrapper functions. For example, whenever a permission denied log message was issued, `process_assert` was told permission had been denied. The process then updated its state and checked that the specified properties held.

The properties we tested were influenced by the description of the behaviour in the man pages. For example, it states which class of users may use particular commands. A particular instance is that to use the `group` command the user must be authenticated as a global administrator. All of these permission type properties have been demonstrated to hold (via assertions) on each class of user with one exception. For example, we have determined that only a global administrator or a group administrator can add or delete users from the system. If the user was not authenticated as a suitable administrator they were prevented from making any changes. We have also determined that when a non-existing user name is given, they are prompted for a bogus password which will always result in the user being denied any access.

The test that failed is now described. The man page states that the `password` command can only be used by a global administrator or a group administrator of the group

to which the user whose password is being changed/set belongs. However, the authentication server also permits any authenticated user to change their own password. This is more likely to be an error in the documentation rather than an error in the code. We used Verisoft to capture both the interaction that occurs, as well as the location in each process when the assertion error is detected. It is the examination of the source code at this point that led us to believe that the error was in the documentation.

This concludes our discussion of the analysis of three aspects of the FWTK software. In summary, the following were the steps necessary to test various aspects of the firewall.

1. Identify system properties to be checked. These would ideally be documented in the description of the software.
2. If the system has many processes, create a new `main` process to generate all the other processes. This is required by Verisoft.
3. Identify all types of interprocess communication used and replace them with Verisoft message queues.
4. Identify code incompatible with Verisoft and replace them with *wrapper* functions for this code. For example, system calls such as `read` and `write` that use file descriptors must be instrumented with appropriate wrapper functions that read and write data to message queues. This can be in conjunction with, or be replacements for, operations involving sockets, file descriptors, etc.
5. Implement the processes that model the environment for the processes under analysis. That is, these processes should provide a simulated environment for the main process under analysis.
6. Implement a procedure to track the state of the main process under analysis. This can either be a separate process or some code periodically executed by the main process.
   (a) Introduce additional wrapper functions to report on the state of the main process. These functions will typically send a message to the state tracker process and call the replaced function.
   (b) Add assertions to the state tracker process to verify that the identified properties hold.

We now present some of the difficulties faced when trying to use Verisoft. While some of these are due to the requirements of Verisoft, the others pertain to the way in which the software is written. Verisoft requires that all required message queues must be created before the initial global state is reached. Hence we predefined a large number which were allocated on demand. While this was adequate for our testing, it is not clear if this will always be the case. Similarly, the number of processes should also be fixed and defined in the configuration file. Hence dynamic process creating was not possible, so it was not possible to test multiple rounds of requests from the environment. We could have changed the source code to loop rather than exit (or execute `execv`) to simulate multiple rounds. However, that require more extensive changes to the source code than was desirable. In some cases Verisoft runs into livelock problems. This is due to the way in which the `vs_select` function, the wrapper for the `select` call, needs to implemented. Unfortunately there is no simple solution as there is no bound on when `vs_select` may have anything to select and what it can select. We had to increase it to a large value like 1000 to enable Verisoft to analyse the program. However this was more of a trial and error technique.

Except for concurrency and `VS_toss` operations, Verisoft assumes that the system being tested is deterministic. This assumption produces an error of the authentication server during automatic Verisoft simulation. The actions taken by `authsrv` are dependent on the state of its database file which can be modified during a Verisoft simulation. It is assumed that this violates the above assumption because it is possible that replaying a specific scenario may have a different outcome due to a different database state. For example, the problem arises whenever a user's database account is disabled. The disabling modifies the database state which cannot be reversed during the replaying of scenarios where the user was previously enabled. To make the process deterministic, the database must be re-initialised between paths in the state space. Since this can not be done during automatic simulation mode, an exhaustive search of the authentication server state space up to some depth when all functionality is examined is not possible. However, an exhaustive search may be possible if the functions that modify the database state are not verified. We could also examine one scenario at a time and reset the database after the analysis of each scenario.

We were also unable to verify conditions such as whether all transactions are logged. This was because we had no easy access to when a transaction was initiated. The source code does not provide any direct feature that we could use. Note that we could verify well defined aspects of a transaction. For example, we can verify that a user exit is logged. This is because user exiting is well defined. In general, the code needs to be structured so that Verisoft hooks can be added routinely. Extensive changes to the software is both time consuming and has the potential of introducing new errors and was not undertaken.

4 Conclusions and Future Work

We were able to gain a thorough understanding of the aspects of FWTK that we studied. We can state with some confidence that the program behaviours we examined are reasonably robust. It has taken about 150 man hours (as dictated by the budget to complete this task. This included installing the Verisoft and FWTK systems, examining the documentation, deciding which aspects to verify and test and actually carrying out the required inspection. Although this is only the first step, (as only a few subsystems of FWTK were examined) the results indicate that independent examination of source code using Verisoft is indeed viable. In the case of a firewall, the principal interest was in the initial phase (where requests are handled). Once a user is permitted to use the resources, the role of the firewall is diminished. Thus a tool like Verisoft which performs exhaustive analysis up to a depth is well suited.

One of the main limitations is that only one configuration could be tested. The code has various `ifdefs` and there is a need to automate the testing of all these configurations. Not all the features of the programs have been examined or tested. For example, there are various ways that authentication can work including DES, one-time passwords, and challenge response calculators. None of these have been examined and we have assumed that the authentication is robust.

More generally, there is a need to identify coding practices that will simplify the inspection process. These could include, for instance, key functions that could be presented as wrapper functions which the inspector could extend, and there should be a variable or function that corresponds directly to a key behavioural level concept (such as transactions). It is also necessary to ensure that the testing environment which interacts with the system addresses all issues. A tool which allows the automatic creation of such environments (which could be based on the approach described in [2]) needs to be constructed.

Acknowledgement

This work has been partially supported by a UoC Summer Scholarship. The authors thank the authors of Verisoft and FWTK for making the tool and the sources available free of charge. Special thanks to Patrice Godefroid for his encouragement and feedback on our work.

References

[1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[2] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, 1998.

[3] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[4] A. Dunsmore, M. Roper, and M. Wood. The role of comprehension in software inspection. *The Journal of Systems and Software*, 52(2–3):121–129, June 2000.

[5] P. Godefroid, R. Hanmer, and L. Jategaonkar-Jagadeesan. Model checking without a model: An analysis of the heart beat monitor of a telephone switch using verisoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 124–133, 1998.

[6] C. Heitmeyer. On the Need for *Practical* Formal Methods. In A. P. Ravn and H. Rischel, editors, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume LNCS 1486, pages 18–26, Lyngby, Denmark, 1998. Springer Verlag.

[7] J. Widmaier. Building more reliable software: Traditional software engineering or formal methods? In *Proc. of ISSRE: Industry Practices and Fast Abstracts*, pages 253–260, Nov. 1999.