Simulating Concurrent Intrusions for Testing Intrusion Detection Systems: Parallelizing Intrusions*

Mandy Chung Nicholas Puketza Ronald A. Olsson
Biswanath Mukherjee

Department of Computer Science
University of California, Davis, CA 95616
{*chungm, puketza, olsson, mukherje*}*@cs.ucdavis.edu*

Abstract

For testing Intrusion Detection Systems (IDS), it is essential that we be able to simulate intrusions in different forms (both sequential and parallelized) in order to comprehensively test and evaluate the detection capability of an IDS. This paper presents an algorithm for automatically transforming a sequential intrusive script into a set of parallel intrusive scripts (formed by a group of parallel threads) which simulate a concurrent intrusion. The main goal of parallelizing an intrusion is to distract an IDS's attention away from the intrusive activity. We identify constraints on the execution order among commands, and the way commands can be classified based on the effect of their execution. Synchronization and communication mechanisms are used to guarantee that the execution order among commands is preserved even under the parallelized scenario. We show that, experimentally, our work constitutes a major part of testing the ability of an IDS to detect intrusions and is especially useful for the users and developers of IDSs. We show that an intrusion is less likely to be detected if the suspicious activity is distributed over several sessions. Finally, we discuss some aspects of parallelizing intrusive scripts, including some practical difficulties that are open problems for future research.

Keywords: Intrusion Detection, Concurrency, Testing, Parallelization, Synchronization, Data Flow Analysis, Dependence Analysis.

1 Introduction

Intrusion detection provides a practical alternative approach to computer security besides designing a secure system [6, 12]. Intrusion Detection Systems (IDS) have been under investigation for many years [7, 14] and have started to move from laboratories to the real world. There is thus a need for sound methodologies and tools for testing IDSs. This paper presents our continuing effort on testing Intrusion Detection Systems [13].

We are researching methods for testing IDSs. In our testing experiments, we simulate intrusive activity, and then study the corresponding output from the IDS. We have developed a software platform that can be used to create scripts that simulate both normal and intrusive activities. We have also developed mechanisms in the platform to support concurrent intrusion simulations, including mechanisms for synchronization and communication (message passing) among different processes.

A major challenge of our work is to be able to simulate intrusions in various forms so that we can test an IDS's capability to detect intrusions comprehensively. A single intrusion can be executed in many different ways. For instance, an intruder may type in the intrusive commands one by one from a single terminal, or an intruder may code them up in a script. An advanced intruder may partition (or parallelize) the commands and issue them from different sources (e.g., different login sessions) to reduce the noticeability of the intrusion by an IDS. Similarly, multiple intruders may attempt to conceal an intrusion attempt by distributing the suspicious behavior amongst themselves.

Manually transforming a sequential intrusion into a concurrent one is very tedious and time-consuming. Besides, a single intrusion can typically occur in a number of different concurrent forms. For this reason, we envision, during the course of testing IDSs, the need for an automated approach to fragment a sequential intrusive script into parallel scripts that cooperate with one another.

This paper presents an algorithm for parallelizing a sequential intrusive script of Unix shell commands. Parallelizing an intrusive script has some similarities to parallelizing a program, which has been studied in depth [2, 3, 4, 5, 8, 9, 11]. Our work adapts some basic techniques used in program parallelization to fit in our context, including data flow analysis, dependence analysis,

*This work has been supported by the National Security Agency (NSA) INFOSEC University Research Program (URP).

and control dependence to data dependence conversion. Our algorithm is based on a dependency graph that represents the meaning of commands as well as their interrelations. A sequential intrusive script is first analyzed to determine various kinds of dependencies among commands which, in turn, enable us to determine their execution order. The sequential script is then transformed into a parallel script, in which synchronization and data communication mechanisms are employed to enforce the dependence relations of commands. A parallel script is formed from a group of parallel threads generated from our algorithm. By assembling parallel threads in different ways, various parallel forms of an intrusive script can be generated. This paper emphasizes the parallelization of intrusive scripts for testing IDSs; however, this approach can be employed for parallelizing arbitrary (non-intrusive) scripts, perhaps for different goals and different optimization.

The rest of this paper is organized as follows. Section 2 presents some example scenarios to illustrate the importance of an IDS to be able to detect concurrent intrusions, which raises the motivation and the need for our work. Section 3 presents some initial results from testing both sequential and parallel intrusive scripts on an actual IDS showing that intruders could escape detection by an IDS by distributing their intrusive activity over several concurrent sessions. Section 4 describes the model for our parallelization algorithm. Section 5 describes the steps involved in the automated parallelization (or transformation) of an intrusive script and also presents an algorithm for generating parallel threads. Section 6 discusses several aspects of parallelizing intrusive scripts, including some practical difficulties which we will deal with in the future. Section 7 concludes the paper.

2 Scenarios

This section presents two intrusion scenarios — password-guessing and password-cracking — demonstrating the fact that an intruder can possibly defeat an IDS's detection mechanism by issuing the intrusive commands from different sources concurrently (e.g., from different login sessions).

In the first scenario, an intruder attempts to guess the password of a user on a target machine. Obviously, he/she can do this by attempting repeated logins with different passwords until he/she successfully enters the system or until he/she gives up after several attempts. An IDS could probably detect this intrusion because a number of failed login attempts from a source machine is very noticeable. However, the guessing of passwords can be distributed among several intruders so that a group of passwords are tested simultaneously, perhaps from different machines. An intruder may also be able to manage to test several passwords concurrently through several open login sessions, or perhaps by an automated script. First,

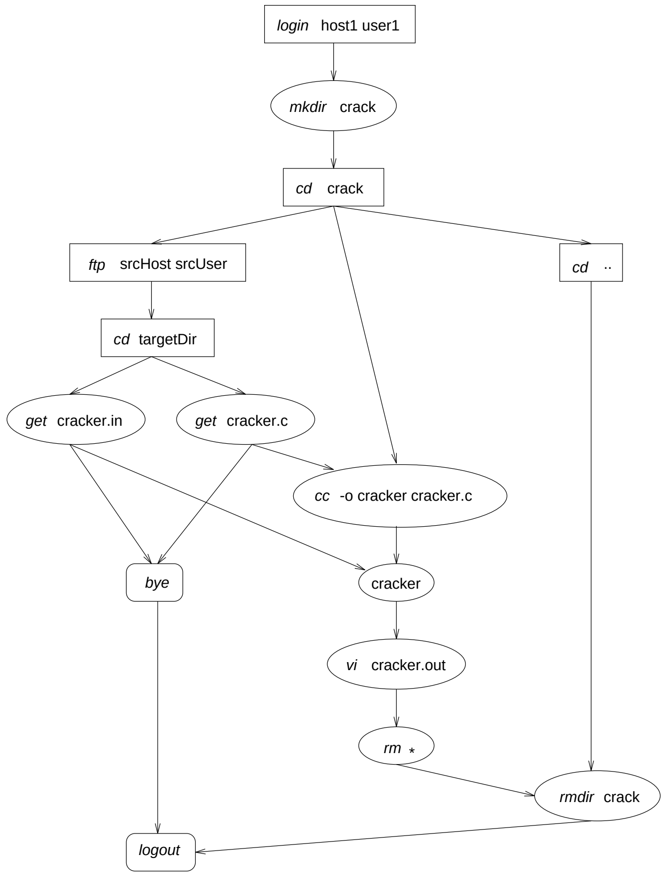

Figure 1: Dependence graph for a sequential password-cracking intrusion.

in this concurrent intrusion, the intruder can test all passwords in a shorter period of time. Second, this concurrent intrusion is less suspicious to an IDS than the sequential one because the logins are issued from different sources. Finally, if the target machine shares the password file with some other machines [1], the intruder can also test passwords on different machines simultaneously (rather than on one target destination). It is extremely difficult for an IDS to aggregate the activities issued from different sources to different destinations and to detect the coordinated intrusion.

In the second scenario, an intruder who manages to enter a target machine attempts to find any password that can be easily cracked. The intruder first logs into a target host host1 as user1 and then creates a temporary directory called crack under his/her home directory. He/she copies the password cracking program cracker.c and the dictionary file cracker.in from another machine srcHost and in a directory targetDir under srcUser home directory by *ftp*. Running the program cracker compiled from cracker.c will generate those cracked passwords into

[1] This is usually the case when Network Information Service (NIS) is running, where the password file is shared among all NIS clients.

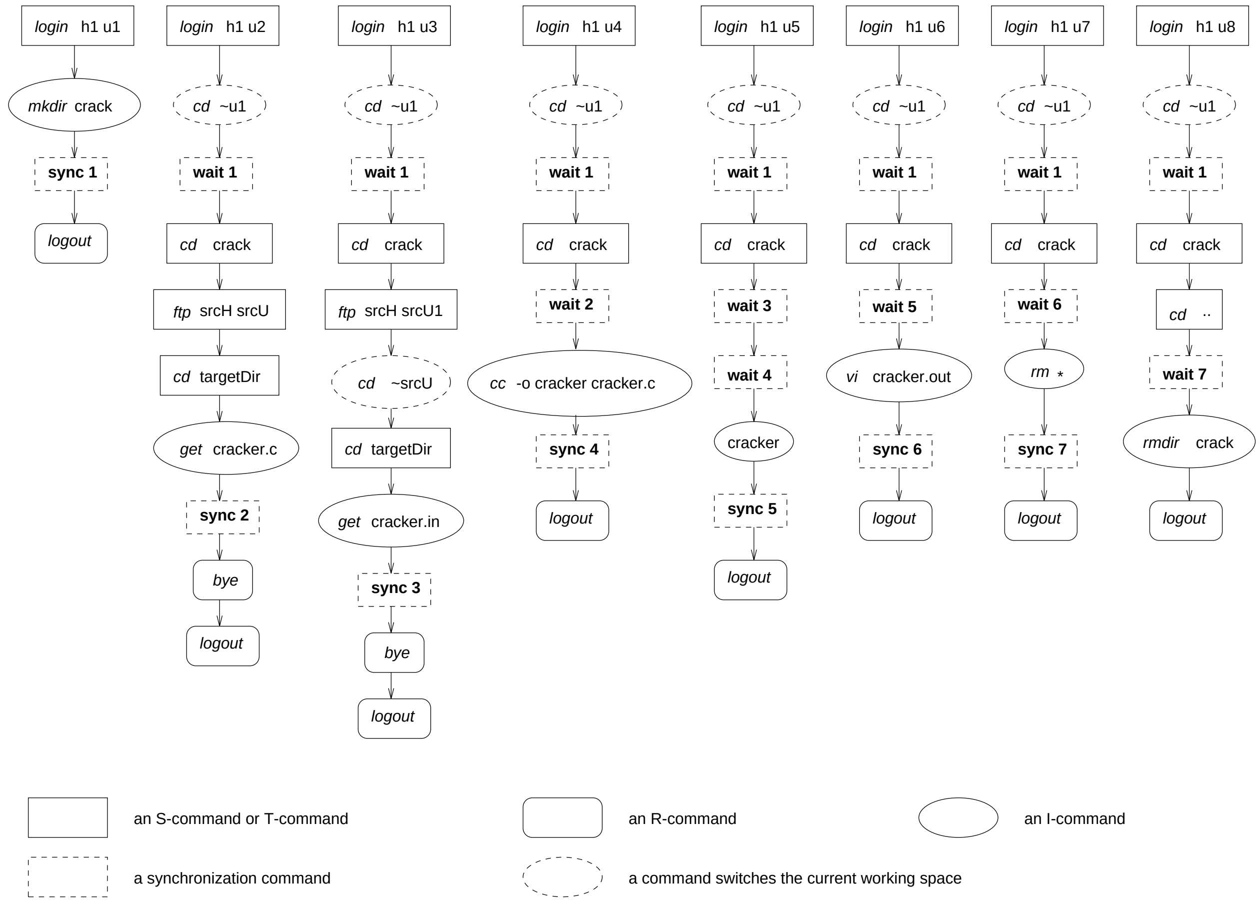

Figure 2: Parallelized password-cracking intrusion.

file cracker.out. After reviewing the output file, the intruder cleans up the working directory crack and leaves the session.

Although the actions in this intrusion, described above, seem to be necessarily performed serially in a single session, they can still be divided and performed in concurrent sessions by multiple coordinated intruders (or intrusion scripts). To illustrate, consider the commands executed in this intrusion and the dependence relations among them. Figure 1 depicts the dependence graph for this password-cracking intrusion. (In Figures 1 and 2, we distinguish between commands of four different types: S-command, R-command, T-command, and I-command, which will be described in Section 4.) Figure 2 shows one possible parallel version of this intrusion. It consists of eight individual intrusive sessions running simultaneously. Each of these intrusive sessions carries minimal activity as shown. That is, it is not possible to separate into two or more threads the activity performed in any one of these parallel threads. Concerning the benefits of parallelizing this password-cracking intrusion, the parallel version does not gain any considerable speedup. On the contrary, the elapsed time of the intrusion may be increased due to the overhead of synchronization between the various parallel threads. However, the speedup is not the main goal of the intrusion parallelization operation. The main goal is to disguise the intrusive activity performed by an intruder or a group of intruders. The major benefit obtained from parallelizing the password-cracking intrusion is to distribute the intrusive activity among concurrent sessions so as to minimize the chance of detection of the activity.

The two examples described above both involve the interaction between the intruder and the shell. Another form of intrusion could be due to a program that makes system calls. Activities issued from either of them may be collected by an audit trail on which the analysis of many IDSs rely. In this paper, we focus on the former form of intrusion involving shell-level commands.

3 Experimental Results

To escape detection by an IDS, intruders might try to distribute their intrusive activity over several concurrent sessions. The premise behind this strategy is that the IDS will assign a higher warning value to one *very* intrusive session than it will to several less intrusive sessions. We conducted some experiments to test this premise.

The IDS that we tested is the Network Security Monitor (NSM) [7]. The NSM monitors all of the packets that travel on the local area network (LAN) to which the NSM host computer is connected. The NSM can associate each such packet with the corresponding computer-to-computer connection. It assigns a warning value between 1 and 10 (higher is more suspicious) to each connection based on the contents of the packets, and on the likelihood of the connection occurring, given a record of recent connections. We ran the NSM on a Sun Sparc Station 2 workstation connected to the Computer Science LAN segment at UC Davis (UCD).

3.1 Test Procedure

We first selected four intrusive activities:

1. transmitting the */etc/passwd* file from one computer to another;
2. password-cracking by comparing the entries in the */etc/passwd* file to a list of encrypted password guesses;
3. password-guessing using a dictionary file; and
4. exploiting the *loadmodule* flaw to achieve super-user status.

For each of these activities, we created a sequential script to simulate the activity. Then, we manually created a concurrent script set which collectively included all of the commands from the sequential script. We activated the NSM and ran the scripts. We then compared the warning values for the sequential script with the warning values for the concurrent script set. The results are displayed in Figure 3. The NSM assigns a warning value to each network connection. Several of the scripts and script sets initiate more than one network connection, but for clarity the figure shows only the maximum warning value for all network connections associated with each script and script set.

INTRUSION DESCRIPTION	SCRIPT TYPE s = sequential c = concurrent	MAX WARNING VALUE
transmitting passwd file	s	7.472
	c	7.472
password-cracking	s	3.160
	c	3.160
password-guessing	s	8.722
	c	7.785
exploting loadmodule flaw	s	7.472
	c	4.972

Figure 3: NSM experimental results.

3.2 Analysis of Results

For the first intrusion simulation in Figure 3, the warning value for the concurrent script set is the same as the warning value for the sequential script, and the warning values are high. A possible explanation for this is that the sequential script contains a very suspicious command or set of commands which cannot be divided when the concurrent script set is created. As a result, at least one of the threads in the concurrent script set is by itself just as suspicious as the original sequential script. The warning values for the second intrusion simulation are again equal, but in this case the values are low. A likely explanation is that the NSM was not configured to be sensitive to that particular intrusion. So, neither the sequential script nor the concurrent script set produced activity that appeared suspicious to the NSM.

For each of the last two intrusion simulations in our experiments, the warning value for the concurrent script set is less than the warning value for the sequential script. In both cases, it was possible to divide up a set of suspicious commands in the sequential script among two or more threads in the concurrent script set.

Taken together, our experiments indicate that it is possible for intruders to reduce the chance of detection by an IDS by distributing their suspicious activities, although this strategy is not always successful. In future work we plan to investigate the effects of this strategy on different IDSs. For example, the NSM monitors each network connection independently. An IDS that, instead, keeps track of all the activity associated with each user may not be affected as much by this intruder strategy.

4 Model

Our work focuses on the automated parallelization (or transformation) of an *intrusive script* that is used to simulate an intruder's activity. An intrusive script is written in a simple programming language which allows us to specify shell-level commands, such as shell language [15] and *Expect* [10]. In addition, the language typically includes variables, procedure, and control-flow statements, such as if-then-else and loop.

This section presents a model for an intrusive script transformation which focuses on issuable shell-level commands in an intrusive script. The model is divided into two parts: shell-level commands and statements in an intrusive script.

An intrusion is considered to be a sequence of shell-level commands issued by an intruder. An intruder can issue commands one by one from a single terminal, or issue commands from more than one terminal. For example, an intruder can create two login sessions from two different windows on his/her workstation to a target host at the same time. On the target host, the identity of the user associated with these sessions can be different (if the intruder manages to get two different accounts on that system). In this paper, we refer to the commands that are issued from a terminal as an *intrusive session*. More specifically, a user can consecutively issue several commands that create a new user session, such as *rlogin*, *telnet*, and *ftp*, from a terminal in which a hierarchical structure of open user sessions is built. An intrusive session refers to the root session of this structure. A sequential intrusion involves only one intrusive session while a concurrent intrusion typically involves multiple intrusive sessions.

A shell-level command, like a procedure, can take parameters (e.g., from command line) and return a result (e.g., to standard output). In addition, a command may change two kinds of states in a computer system: the file system state and the intrusive session state. The file system state includes the existence and the content of files in the file system. The intrusive session (IS) state consists of all predefined and user-defined environment variables, including the real user ID (uid), the effective user ID (euid), the group ID (gid), the current working directory (cwd), and the hostname. The current IS state refers to the state of the active user session in an intrusive session. We characterize a command C by the following:

- Input parameters and output result.
- A set of file system objects from which C reads.
- A set of file system objects to which C writes.
- A set of IS state attributes[2] on which C depends.
- A set of IS state attributes which C changes.

Based on the above definition of a command, the dependence constraints on the intrusive script transformation are defined as follows:

Data dependence. Two commands are data dependent if the input parameter of one command is determined by the output of another, or some file system objects written by one command are referenced (read or written) by another and these file system objects are not written by other commands between the execution of these two commands.

Attribute dependence. Two commands are attribute dependent if some IS state attributes changed by one command are referenced (read or changed) by another and these attributes are not changed by other commands between the execution of these two commands.

An intrusive script typically consists of various constructs provided by the language. Control dependence is another constraint to the parallelization problem due to the presence of control-flow statements in the script (see Section 5.2.1 for details).

[2]In order to distinguish between an environment variable and a script variable, we refer to an environment variable as an attribute of the IS state.

We also classify commands into four different types according to their effect on the IS state:

1. I-command *(state Invariant)* — A command that does not affect the IS state. For example, *ls*, *cp* and *cat* are I-commands. These commands only affect the file system state, e.g., change the content of a file or create a new file.
2. S-command *(Session creation)* — A command that creates a new user session and changes the IS state, but the IS state before executing this command can be restored by an R-command. Examples of S-commands are *rlogin* and *ftp*, which change the IS state attributes, uid, euid, gid, cwd, and host. *su* is slightly different from the above since it only changes uid, euid, and gid.
3. R-command *(state Restoration)* — A command that closes the current user session and restores a previous IS state. Specifically, an R-command reverts the IS state to a state in which the corresponding S-command began without requiring the knowledge of those executed commands and their parameters, or the value of the previous IS state. For example, *logout* is an R-command corresponding to *rlogin* whereas *bye* is an R-command corresponding to *ftp*.
4. T-command *(state Transition)* — A command that changes the IS state, but no R-command corresponds to it. For example, *cd* is a T-command that changes the current working directory (cwd) and *setenv* is another T-command that defines an environment variable. Although the IS state before executing a T-command can be restored via a series of T-commands, *cd* and *setenv* are neither an S-command nor an R-command since they require the knowledge of the value of a previous IS state (for restoration) and since they do not open or close a user session.

As we described earlier, a script can contain variables, control-flow statements, and constructs for issuing shell-level commands. Figure 4 shows a simple *Expect* script that controls an *rlogin* session.

For simplicity, when parsing the script statically, we refer to a statement that issues a shell-level command as a "*command*" (e.g., lines 5, 7, 11, 13, and 14 in Figure 4), while we refer to other constructs provided by the language as "*statements*" (e.g., lines 2, 3, and 10 above). More precisely, the issuable commands (line number of its referred statement) in Figure 4 are: *rlogin* (lines 5 and 7)[3], *whoami* (line 11)[4], *ls -l* (line 13), and *logout* (line 14). Lines 10 and 11 together form a conditional statement containing an issuable command *whoami*.

[3]Inputting password is considered to be part of the rlogin.

[4]Line 11 will be invoked only if rlogin to the host occurs as root.

```
1   # get target host and user from arguments
2   set host [lindex $argv 1]
3   set user [lindex $argv 1]
4   # spawn an rlogin process
5   spawn rlogin $host -l $user
6   # expect the password prompt,
    # then send the password.
7   expect {"Password:" send "actualpassword\r"}
8   # expect the shell prompt,
    # then send shell-level commands
9   # The shell prompt is specified
    # in a regular expression.
10  if {$user == root} {
11     expect {-re ".*%|.*>|.*#" send "whoami\r"}
12  }
13  expect {-re ".*%|.*>|.*#" send "ls -l\r"}
14  expect {-re ".*%|.*>|.*#" send "logout\r"}
```

Figure 4: A simple *Expect* script.

5 Automated Parallelization of an Intrusive Script

Parallelization of a sequential intrusive script consists of the following steps :

1. Parse an intrusive script and build a flow graph.
2. Convert the control dependence to data dependence.
3. Perform dependence analysis and build a data dependence graph.
4. Create parallel threads for the intrusive script and insert synchronization and data communication commands to facilitate coordination between parallel threads.
5. Perform optimization and transformation, if any.
6. Generate a parallel intrusive script.

A flow graph representing an intrusive script is different from a flow graph representing a program [1]. We define a basic block, in our context, as a sequence of consecutive "*statements*" and one issuable "*command*". Therefore, it is possible to have a basic block containing a conditional statement or loop which contains no command. The intrusive script transformation procedure is only interested in the issuable shell-level command contained in a basic block. We assume that two basic blocks are dependent only if the issuable commands in the basic blocks are dependent (see Section 4). Information obtained from the evaluation of other statements in a basic block is only used within the basic block and is independent of other basic blocks. That is, script variables used or modified in a basic block are not referenced elsewhere.

In parallelizing an intrusive script, we must obey the constraints of the underlying dependence structure of the script. In our algorithm, a data dependence graph [8] is used to represent the data dependence, attribute dependence, and control dependence. Both data and attribute dependence of commands can be represented in a data dependence graph because the IS state attributes are another form of data in an intrusive session. We can also treat control and data dependence uniformly by applying a technique in parallelizing compilers introduced by Allen and Kennedy [2] to convert control dependence into data dependence. We also adapt the dependence analysis [3, 5, 8, 9] used in program parallelization to determine the dependence relations of all issuable commands in a script.

In the following, when we refer to a command in a dependence graph, we actually mean the basic block containing this command. A node in a dependence graph represents a basic block which contains one issuable command and other statements.

Section 5.1 presents an algorithm to generate parallel threads in parallelizing an intrusive script, which does not have branch, loop, and procedure. Section 5.2 extends the algorithm to handle conditional statements and loops. It also discusses another dependence due to the script variables used or modified in a basic block and referenced by another. Section 5.3 discusses the possible optimization and transformation performed on the parallelized intrusive threads.

5.1 Parallelization Algorithm

This algorithm consists of two phases: parallel threads generation phase and threads synchronization phase. Appendix A gives the pseudo-code for this algorithm.

5.1.1 Parallel Threads Generation Phase

We represent an intrusive script as a series of commands, $I = \{C_1, C_2, \ldots, C_n\}$ since there is no branch, loop, and procedure in the script. The IS state transition during the execution of I is $\{s_0, s_1, \ldots, s_n\}$ where s_0 is the initial IS state determined by the input of I. This phase creates a parallel thread for each I-command in I. In order to guarantee that an I-command executed in a parallel thread has the same effect as in the sequential script, the IS state s_{i-1} must be reached before the execution of C_i begins. Specifically, s_{i-1} is reached by executing all S-commands and T-commands in $\{C_1, C_2, \ldots, C_{i-1}\}$ serially.

The details of the algorithm are as follows. The algorithm processes each command and uses a stack to store those S-commands and I-commands whose executions reflect the current IS state. When processing an I-command, the algorithm creates a new thread for performing this command. We use a flow graph to represent a sequence of commands executed in a parallel thread. First, a flow graph is formed by creating a node for each element (each command) in the stack; the bottom one in the stack is the first command executed in the thread while the top one is the last command. The algorithm then appends a node representing the I-command to the graph. When processing an S-command or a T-command, this command is pushed onto the stack and no thread is created. If it is an S-command, all new threads containing this S-command are recorded so that their open sessions can be closed appropriately. When processing an R-command C_r, all recorded threads for the S-command C_s that corresponds to C_r are appended with C_r to close their current sessions. Precisely, C_s is the top S-command in the stack. All subsequent commands following C_s (including C_s) in the stack are popped (since the open session created by C_s is closed by C_r).

After processing all commands in the script, the number of parallel threads generated is the number of I-commands in the script. For example, Figure 1 is the dependence graph for the sequential password-cracking intrusion scenario discussed in Section 2, and it turns out that this example has eight I-commands. Figure 2 shows the eight parallel threads generated by this algorithm and each thread performs only one of these eight I-commands.

5.1.2 Threads Synchronization Phase

After all parallel threads are generated, the dependence relations among commands are enforced in this phase to guarantee the execution order of the commands by inserting synchronization and data communication mechanisms [11].

A dependence graph G consists of nodes and directed edges. A node represents a basic block containing a single command. A directed edge (u, v) represents a dependence relation between basic blocks u and v, i.e., the execution of v can begin only after the execution of u terminates. By performing breadth-first search on G, all commands (nodes) are visited. While visiting a node containing an I-command C_i, we insert a synchronization command to each successor of C_i in G, C_j, to ensure that its execution begins only after C_i terminates. If the input of C_j depends on the output of C_i, the output of C_i is sent to the thread containing C_j using data communication commands.

As in the parallel threads generation phase, all S-commands, T-commands, and R-commands may be duplicated in other threads whereas an I-command is executed in only one parallel thread. If C_i is either an S-command, a T-command, or an R-command, the execution order of C_i and its successors is guaranteed to occur in sequential order.

5.2 Language Constructs

5.2.1 Conditional Statements

Conditional statements are very useful in simulating an intrusion. As a simple example of control dependence in an intrusion, consider an intruder who checks the permission mode of a file named fileA and decides whether to read fileA or to modify fileA. The first command performed by the intruder is *ls -l* fileA to obtain the file access information. According to the file access permissions of fileA, if it is world-writeable, he/she then modifies the file by *vi*; otherwise, if it is world-readable, he/she reads the file by *cat*.

For simplicity, in the following example, we focus on the issuable commands and ignore all other statements within the if-statement.

```
if B then
    command1
    command2
else
    command3
endif
command4
```

Command1, command2, and command3 are control dependent on the boolean expression B since B determines which command is executed. (B may be obtained from the output of a previous command.) The conversion from control dependence to data dependence proceeds by first replacing the if-statement at the source of the dependence with an assignment statement to a boolean variable. The converted if-statement is:

```
b = B
command1 when b
command2 when b
command3 when not b
command4
```

All control dependent commands are tagged with a "**when** b" or "**when not** b" clause depending on to which arm of the original if-statement they belong. The operator **when** indicates that the expression on its left is executed only if the boolean expression on its right is true. A boolean variable b is used instead of B because statements in either arm of the if-statement could have side effects, i.e., these side effects could change B. After the conversion, the control dependences of command1, command2, and command3 become flow dependences generated by variable b and each command is contained in a single basic block in the flow graph.

After the if-conversion, the parallel threads can be generated as described in Section 5.1 but a slight modification must be made to handle the tagged R-commands as follows. When processing a tagged R-command with operand b, C_r **when** b, all recorded threads containing its corresponding S-command C_s are appended with this tagged R-command. Consider those S-commands and T-commands executed between C_s and C_r, i.e., $\{C_s = C_{i_0}, C_{i_1}, \ldots, C_{i_k}\}$ in the stack. First, for $0 \leq j \leq k$, all tagged C_{i_j} whose operand is satisfied with the current b value are popped from the stack because, if b is true, the current user session is closed. Then, all remaining C_{i_j} in the stack are modified to tag with a "**when not** b" clause if it is not tagged, or to replace the "**when** c" clause with a "**when** c **and not** b" clause. Therefore, all remaining C_{i_j} in the stack are guaranteed to execute under correct IS state.

The tagged commands now depend on the basic block that evaluates b. All threads that contain the tagged command may need to receive the b value from another thread, which we refer to as variable dependence (to be discussed in Section 5.2.3).

5.2.2 Loops

Recall from Section 2 that, in the sequential password-guessing intrusion example, an intruder repeatedly attempts to log into a target machine and guess a password until he/she successfully enters the target machine or finishes guessing all passwords in his/her stock. This intrusion obviously contains a loop for testing passwords.[5] In this example, an iteration of the loop — logging in and guessing a password — is independent of other guesses and can be executed concurrently. Although loops in an intrusive script may not be as commonly used as loops in a program, this construct is considerably useful in simulating certain types of intrusions.

We follow the terminology used in parallelizing compilers proposed by Banerjee [3] to classify three parallel loop forms.

- DOALL is a loop that allows total parallel execution, i.e., all iterations of the loop body are allowed to run simultaneously.
- DOACROSS is a loop that allows partial overlap of successive iterations during execution.
- DOSEQ is a sequential loop without parallelism.

After performing the loop dependence analysis, we can identify the type of loops contained in the script. Two kinds of DOALL loops are parallelizable. First, a DOALL loop that contains only I-commands is parallelizable. Second, a DOALL loop is parallelizable if the loop's body can be divided into three blocks such that the first and the last blocks contain only I-commands while the middle block starts with an S-command and ends with an R-command corresponding to this S-command. In other words, a parallelizable DOALL loop terminates with a previous IS state right before its execution begins. Because each iteration of the loop is independent on others, a new thread is generated for executing an iteration of the parallelizable loop. A special type of node is used to represent this parallelizable DOALL loop in the dependence graph so that the parallel threads generation phase can recognize this loop and parallelize it accordingly.

[5]When the connection is closed (after the configured number of incorrect login attempts), the intruder may need to issue a login command again.

DOACROSS loops containing only I-commands (no IS state change) can also be parallelized in a similar manner but synchronization and data communication mechanisms must be appropriately inserted both at the end of one iteration and before another iteration begins, just as in processing dependent commands.

If a thread generated from the loop parallelization procedure contains more than one I-command, this thread can further be parallelized as if it is a sequential intrusive script by recursively applying the algorithm on it. Parallel threads may be required to transmit data due to the loop parallelization because of DOACROSS loops.

Other kinds of loops are considered as non-parallelizable. Among these non-parallelizable loops, a loop containing only I-commands is treated as a single basic block so that a single thread can be created to perform this loop's activity. However, a non-parallelizable loop that contains commands other than I-commands will inhibit the parallelization. That is, all commands following this loop together with the loop is treated as a single basic block in the dependence graph, and hence, it is the last thread created in the parallelization. In particular, a script that has this kind of loop but no I-command before it is non-parallelizable. In fact, under some situation, further parallelization may be allowed even if the script contains such non-parallelizable loops. For example, if the execution of a loop restores the original IS state after it terminates, it can be treated as a single basic block. Duplication of the loop in each parallel thread generated for commands that follow it may also be feasible in some cases. However, the analysis involved to guarantee that the execution of such a non-parallelizable loop in more than one thread resulting in a correct and safe state is very complicated and difficult.

5.2.3 Variable Dependence

So far, we have considered that information obtained in a basic block is independent of other basic blocks. Two basic blocks are dependent only if the issuable commands in the basic blocks are dependent. Therefore, script variables used or modified in one basic block are not referenced in the others. Under certain circumstances, such as the operand introduced by the if-conversion, the algorithm may need to handle the variable dependence across basic blocks.

5.3 Optimization and Transformation

After generating parallel threads from the dependence graph, several possible optimizations can be performed on each thread. One possible optimization is to log in as a different user in each thread. Consider the password-cracking example presented in Section 2. Figure 2 shows eight parallel threads generated from the dependence graph shown in Figure 1 by the algorithm. As shown in Figure 2, each thread can *login* as a different user unless user1 is *root* in the sequential script (Figure 1). If different users are used in parallel threads, we must ensure that all referenced files are accessible by all of these users. This can be achieved either by setting the permission mode of these shared files explicitly by the users, or by assigning one thread to change the permission modes of these shared files.

Another possible optimization is to remove redundant commands executed in a thread. For example, the last parallel thread in Figure 2 removes the temporary directory crack; however, two redundant *cd* commands are executed before *rmdir*. Obviously, they can be removed from this thread without altering its intrusive behavior.

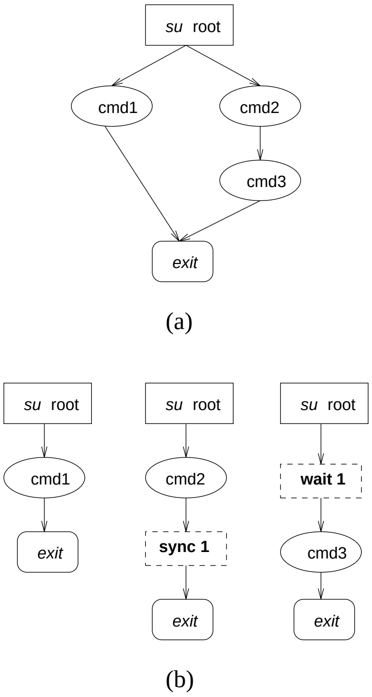

Figure 5: Transformation example.

As the parallel threads generated contain minimal activity, a login session in the sequential intrusion may be broken down into multiple login sessions in the concurrent intrusion. For example, Figure 5(a) shows a subtree of the dependence graph representing one part of the intrusion and Figure 5(b) shows the three parallel threads generated to perform the activity in Figure 5(a). Although these parallel threads perform individual root login session independently, three root logins may be more suspicious to an IDS than the one root login in the sequential intrusion. In this case, parallelization of this subtree may not be beneficial. One possible transformation in this example is to combine the threads into one thread to avoid creating suspicious login sessions. Therefore, Figure 5(a) is used for that part of the intrusion after the transformation.

The parallel intrusion generated by our current approach uses the same working space as the sequential intrusion. For instance, in Figure 1, all files created or added by this sequential cracker intrusion are placed under the directory ~user1/crack in machine host1, and they will be removed at the end of the intrusion. Although different parallel threads of the concurrent cracker intrusion can log into different users and have their own working space, they access and use the same working space as in the sequential intrusion, e.g., the directory ~user1/crack. With further analysis on the semantics of the script, it might be desirable to transform the threads to use their own working space if possible.

6 Discussion and Future Work

This section discusses several aspects of parallelizing an intrusive script: various forms of parallel scripts generated by our algorithm, some practical difficulties, and generalization of our algorithm.

We have presented an algorithm for parallelizing a sequential intrusive script into one possible parallel intrusive script in Section 5. The intrusive activity is basically performed by several parallel threads concurrently. Each parallel thread carries minimal intrusive activity. In fact, other possible parallel intrusive scripts can easily be generated by assembling parallel threads together in different ways to form different combined threads. Thus, an intrusion can be systematically mutated from a sequential form into various different parallel forms, which can be used in testing an IDS.

Parallelizing an intrusive script is difficult in practice because of the rich set of shell-level commands, and various constructs supported by the script language. However, we will deal with these difficulties and search for their possible solutions in the future. Five practical difficulties are discussed below.

Domain and range analysis of shell-level commands. Our algorithm relies on the assumption that we have the knowledge about the domain and range of every Unix command. The domain of a command is the set of file system objects it reads from and the set of IS state attributes it uses, while the range is the set of file system objects it writes to and the set of IS state attributes it changes. However, they are difficult to obtain systematically. First, the domain and range of a command may differ with different arguments. Sometimes, the meaning of a command changes substantially with different option arguments, and so do the domain and range. For instance, the command "*cp* a b" copies the file a to a file b, whereas the command "*cp -r* a b" copies all files under the directory tree rooted by a to a directory b if a is a directory. In this example, the domain and range of these two *cp* commands may be different due to the "*-r*" option. In addition, the user is able to redirect I/O to and from files as well as redirect the output of one command as input to another command using pipes. For example, the command "*ls -l* > file" changes the file system state while the command "*ls -l*" alone does not. In Unix, command aliases, hard links, and symbolic links are allowed to be created. The command *cp* at one time may refer to the program */bin/cp*, but at another time, it may refer to */bin/ls* if the user has made an alias named *cp* of this *ls* command. Similarly, references on a hard link or a symbolic link made by a command may actually refer to another file. Apart from standard Unix commands, a user can also execute a user-defined command which can be a program making system calls. To determine the domain and range of such a program requires detailed analysis of the program source.

Complexity of text editor commands. Our algorithm cannot completely handle text editor commands because of their complexity. For example, *vi* can edit different files before exiting the editor session. It can also start a new user session via shell escape. Activities performed within an editor session are very hard to analyze from a script. We currently handle *vi* and *emacs* as I-commands that only access and modify one file specified as a parameter and do not have other side effect.

Side effect of additional open user sessions. S-commands and T-commands may be duplicated in parallel threads. The execution of the duplicated S-commands on parallel threads opens additional user sessions which may affect the current state of the computer system, such as the list of users currently on the system, the number of active processes, and the last login time of an attacked account. In other words, execution of S-commands may affect the output of some I-command in a script. Our algorithm does not currently handle an intrusive script containing an I-command which depends on the system state. For example, one of the actions performed by an intruder is to discover the last user who logs into machine A. When an intruder has logged into A, the user whom the intruder wants to find becomes the second last user who logs into A. The command "*last* -2" can be used to get this information in a sequential intrusive script, but it is not necessarily correct when used in a parallel script.

Interprocedural Analysis. Our algorithm handles control-flow constructs used in a script, such as if-then-else and loops. Procedures are another construct that complicates our dependence analysis. However, by analyzing the effect of a procedure call, including which parameters changed on return, what global variables are used and modified, and other side effects, we can determine whether the procedure presents a constraint on an intrusive script's parallelization. In some situations, a procedure can be expanded in-line by substituting the formal parameters with actual parameters, and renaming local variables. However, in-line expansion is not applicable to recursive procedures.

Suspension and resumption of open user sessions. Currently, our algorithm does not handle commands that suspend or resume a user session. We believe that these commands are rarely used in intrusions. Our algorithm can easily be extended to handle them. The command that suspends a user session can be treated similar to an R-command. Instead of just restoring a previous IS state, those commands whose executions reflect the suspended IS state are kept so that this suspended IS state can be restored when this suspended user session is resumed.

Our algorithm described in this paper focuses on parallelizing a sequential intrusive script of Unix shell commands. However, it can be generalized to apply to other system platforms, e.g., VMS. The major difference in the intrusive script transformation on different system platforms is the definition of the intrusive session state on which the dependence analysis depends, but same kinds of analysis can be used. In the future, we will study the dependence relations of commands on various platforms and thus our work can be applied to testing other IDSs that run on other system platforms.

7 Conclusion

In this paper, we have presented an automated mechanism for parallelization of intrusive scripts for testing an IDS. Being able to simulate an intrusion in different forms is very important for testing the ability of an IDS to detect intrusions. Parallelizing an intrusive script has some similarities with parallelizing a program; however, they differ in some aspects mainly due to the shell-level commands involved in a script and their additional constraints on parallelization. By modeling shell-level commands and intrusive scripts, we can adapt the methodologies used in program parallelization for parallelizing intrusive scripts.

The transformation of an intrusive script allows us to generate other possible parallel forms so that an IDS can be thoroughly tested. We conducted some experiments on testing an IDS with both sequential and parallel intrusive scripts. The initial results showed that the detection mechanism of an IDS could be defeated when an intruder distributes the intrusive activity over concurrent sessions. We believe that our work is especially useful to the developers of IDSs in testing their products. Besides, a system administrator can also evaluate and compare the effectiveness of the detection mechanism of an IDS with the help of the intrusive scripts transformation. We expect that our work constitutes a major part in testing IDSs.

In the future, we will deal with the practical problems in parallelizing intrusive scripts that we have discussed. We will also conduct some experiments on testing operational IDSs with both sequential and parallel intrusive scripts to obtain further results. Finally, we would like to mention that our work is used for testing IDSs rather than for launching new intrusions.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools.*, Addison-Wesley, 1986.

[2] J.R. Allen, et al., "Conversion of Control Dependence to Data Dependence", *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983.

[3] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Boston, Mass., 1988.

[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua, "Automatic Program Parallelizaton", *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211-43, February 1993.

[5] M. Burker and R. Cytron, "Interprocedural Dependence Analysis and Parallelization", *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pp. 17-22, June 1984.

[6] D.E. Denning, "An Intrusion Detection Model", *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 222-232, February 1987.

[7] L.T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A Network Security Monitor", *Proc. 1990 Symposium on Research in Security and Privacy*, pp. 296-304, May 1990.

[8] D. J. Kuck, et al., "Dependence Graphs and Compiler Optimization", *SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 207-218, January 1981.

[9] Z. Li, P. Yew, and C. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers", *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 26-34, January 1990.

[10] D. Libes, "Expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

[11] S. Midkiff and D. Padua, "Compiler algorithm for Synchronization", *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1485-1495, 1987.

[12] B. Mukherjee, L.T. Heberlein, and K.N. Levitt, "Network Intrusion Detection", *IEEE Network*, vol. 8, no. 3, pp. 26-4l, 1994.

[13] N. Puketza, B. Mukherjee, R.A. Olsson, and K. Zhang, "Testing Intrusion Detection Systems: Design Methodologies and Results from an Early Prototype", *Proc. 17th National Computer Security Conference*, vol. 1, pp. 1-10, October 1994.

[14] S.E. Smaha, "Haystack: An Intrusion Dection System", *Proc. IEEE 4th Aerospace Computer Security Applications Conference*, December 1988.

[15] P. Wang, *An Introduction to Berkeley Unix*, Wadsworth Publishing Company, Belmont, California.

A Appendix: Algorithm for Parallelizing Intrusive Scripts

Parallel Threads Generation Phase:

S = empty stack `/* S-commands stack */`
$M = \emptyset$ `/* resulting parallel threads */`
For each command $C_i \in$ an intrusion I do
  if C_i is an S-command with uid $= U$ then
    ∘ push(S, C_i); push($S, cd \sim U$)
  if C_i is a T-command then
    ∘ push(S, C_i)
  if C_i is an I-command then
    ∘ g = gen_graph(S); append C_i to g
    ∘ $M = M \cup \{g\}$
  if C_i is an R-command then
    ∘ find the S-command C_s in S that corresponds to C_i (i.e., the top S-command in $S = C_s$)
    ∘ for each graph g containing C_s do
      ⋄ append C_i to g
    ∘ pop(S) until C_s is pop

gen_graph(S) generates a new thread containing all commands in the stack S.

Threads Synchronization Phase:

Perform a BFS on the dependence graph G and perform the following when visiting a node of command C_i:

if C_i is an I-command then
  ∘ find the graph g in M containing C_i
  ∘ for each successor of C_i in G, say C_j
    ⋄ for each graph g' in M containing C_j
      ∘ if input of C_j depends on output of C_i then
        ⋄ insert command in g after C_i (to send data to C_j in g')
        ⋄ insert command in g' before C_j (to receive data from C_i in g)
      ∘ else
        ⋄ insert sync command in g after C_i
        ⋄ insert sync command in g' before C_j