Anomaly Detection Using Call Stack Information

Henry Hanping Feng[1], Oleg M. Kolesnikov[2], Prahlad Fogla[2], Wenke Lee[2], and Weibo Gong[1]

[1] *Department of Electrical and Computer Engineering*
University of Massachusetts
Amherst, MA 01003
{*hfeng, gong*}*@ecs.umass.edu*

[2] *College of Computing*
Georgia Institute of Technology
Atlanta, GA 30332
{*ok, prahlad, wenke*}*@cc.gatech.edu*

Abstract

The call stack of a program execution can be a very good information source for intrusion detection. There is no prior work on dynamically extracting information from call stack and effectively using it to detect exploits. In this paper, we propose a new method to do anomaly detection using call stack information. The basic idea is to extract return addresses from the call stack, and generate abstract execution path between two program execution points. Experiments show that our method can detect some attacks that cannot be detected by other approaches, while its convergence and false positive performance is comparable to or better than the other approaches. We compare our method with other approaches by analyzing their underlying principles and thus achieve a better characterization of their performance, in particular, on what and why attacks will be missed by the various approaches.

1 Introduction

A lot of research has focused on anomaly detection by learning program behavior. Most of the methods proposed were based on modeling system call traces. However, there has not been much improvement on system call based methods recently in part because system calls themselves only provide limited amount of information. Invoking system calls is only one aspect of program behavior. We can also use other aspects, such as the information contained in the call stack, for intrusion detection purposes.

There is prior work on using finite state automata (FSA) to model program behavior. Wagner et al. proposed to statically generate a non-deterministic finite automaton (NDFA) or a non-deterministic pushdown automaton (NDPDA) from the global control-flow graph of the program [17]. The automaton was then used to monitor the program execution online. Sekar et al. proposed to generate a compact deterministic FSA by monitoring the program execution at runtime [16]. Both methods were proposed as system-call-based. However, what is really appealing is that both implicitly or explicitly used the program counter information to construct states. The program counter (PC) indicates the current execution point of a program. Because each instruction of a program corresponds to a distinct PC, this location information may be useful for intrusion detection.

In addition to the current PC, a lot of information can be obtained about the current status and the history (or the future, depending on how it is interpreted) of program execution from the call stack, particularly in the form of return addresses. Thus, the call stack can be a good information source for intrusion detection. However, to the best of our knowledge, there is no prior work on dynamically extracting information from the call stack and effectively using this information to detect exploits.

In this paper, we propose a new anomaly detection method, called *VtPath*, that utilizes return address information extracted from the call stack. Our method generates the abstract execution path between two program execution points, and decides whether this path is valid based on what has been learned on the normal runs of the program. We also developed techniques to handle some implementation issues that were not adequately addressed in [16], using techniques that are much simpler than those described in [17].

Based on our understanding of the principles behind VtPath and the approaches in [17, 16], we believe the VtPath method can detect some attacks that cannot be detected by the other approaches. We developed several attacks in our experiments to verify that this is indeed the case. Our experimental results also show that the VtPath method has similar convergence and false positive performance as the deterministic FSA based approach.

Another contribution of this paper is that we attempt to compare the various approaches by analyzing their underlying principles and thus achieve a better characterization of their performance, particularly on what and why attacks will be missed by the various approaches.

The rest of the paper is organized as follows. Section 2 describes the related research. Section 3 presents the VtPath method. Section 4 discusses important implementation issues. Section 5 presents experimental evaluation results. Section 6 presents the comparison of the VtPath method to other approaches. Section 7 summarizes the paper and discusses future work.

2 Related Work

The *callgraph* model Wagner et al. proposed characterizes the expected system call traces using static analysis of the program code [17]. The global control-flow graph is naturally transformed to a NDFA. This automaton is nondeterministic because in general which branch of choices will be taken cannot be statically predicted. The NDFA can then be used to monitor the program execution online. The operation of the NDFA is simulated on the observed system call trace non-deterministically. If all the non-deterministic paths are blocked at some point, there is an anomaly. It was stated that there were no false alarms because all possible execution paths were considered in the automaton.

Wagner et al. pointed out that the callgraph model allows some impossible paths. Basically, if a function is called from one place but returns to another, the model will allow the impossible path, which should not occur in any normal run. We refer to this as the *impossible path* problem. To solve it, Wagner et al. proposed a complex pushdown automaton model, called the *abstract stack* model, in which the stack forms an abstract version of the program call stack. Namely, everything but the return addresses is abstracted away. We use a similar virtual stack structure for our method, but we avoid the complex generation and simulation of pushdown automata. In addition, our method dynamically extracts information from call stack at runtime, while both of the above models only dynamically monitor system calls.

One main problem of the above models is that the monitor efficiency is too low for many programs. The monitor overhead is longer than 40 minutes per transaction for half of the programs in their experiments [17]. This is because of the complexity of pushdown automata and the non-determinism of the simulation. Also, too much nondeterminism may impair the ability to detect intrusions. This problem is not well addressed in the paper. There may be scalability problem too because of the human efforts in refining models for some libraries.

Giffin et al. refined the ideas behind the above models [7]. Their approach applies static analysis on binary executables, so it is not dependent on any programming language, but on working platforms. They developed many optimization and obfuscation techniques to improve the precision and efficiency. In particular, "inserting null calls" is their main technique to largely decrease the degree of nondeterminism and help solve the impossible path problem, and consequently, increase the precision. This technique requires the rewriting of the executables and the change of the call name space. This may be appropriate for remote execution systems, which is the application context of their approach. However, this technique may be inappropriate or undesired under the common host-based anomaly detection environment. In addition, Giffin et al. reported high efficiency (low overhead) in their experiments. They added large delay per real system call to simulate network round trip time (RTT), and small delay (4 magnitudes lower than the simulated RTT delay) for each null call inserted. It is possible that most of the run time was spent on the simulated RTT delay, and the relative overhead appeared small even if many null calls were added. In particular, the network delay for thousands of null calls inserted is only comparable to the delay for one real system call. The relative overhead may not appear so small under the common host-based anomaly detection environment with no network delay involved.

The method proposed by Sekar et al. does not have the problems related to non-determinism. Instead of statically analyzing the source code or binary, the method (we call it the FSA method) generates a deterministic FSA by monitoring the normal program executions at runtime. Each distinct program counter at which a system call is made is a state. System calls are used as the labels for transitions. The FSA can then be used to monitor the program execution online. If the stack crashes, or a state or transition does not exist, there may be an anomaly. There are false positives also because some legal transitions or states may never occur during training. Because each transition is deterministic, the efficiency is high and the method will not miss intrusions due to non-determinism. The FSA method also suffers from the impossible path problem mentioned earlier in this section. This problem was not addressed in the paper. Also, some implementation issues were not adequately addressed. The way DLLs were handled is so simple that some intrusions on the DLLs may be missed. We will have a more detailed discussion on these issues later in the paper.

Ashcraft et al. proposed to use programmer-written compiler extensions to catch security holes [1]. Their basic idea is to find violations of some simple rules using system-specific static analysis. One example of these rules is "integers from untrusted sources must be sanitized before use". While we agree that their method or this kind of methods can be very useful in finding programming errors, we do not think it is a panacea that can solve all the problems. A lot of security requirements are subtle and cannot be described in simple rules. For example, their range checker can only guarantee "integers from untrusted sources are checked for range", but not "checked for the right range", because "the right range" is very subtle and too instance-specific to be developed for each instance of untrusted integers. As a result, sometimes we can decide whether an action should be permitted only by checking whether this action occurs before in normal situations. We think dynamic monitoring based anomaly detection methods, such as our method and the FSA method, are still important even if there are many static bug removers. In fact, these two kinds of approaches are good complements to each other. The static methods can remove many logically obvious bugs, and because we cannot remove all the bugs, dynamic monitoring can help detect the exploits on the remaining holes. Another problem with Ashcraft's approach is that the rules have to be system-specific because "one person's meat is another person's poison". The human efforts to develop these rules may not be as easy. If the rules developed are not precise enough to generate low false positives, the programmers will just think of some ways to bypass the rule checking.

There are many methods that only model system call traces. The N-gram method models program behavior using fixed-length system call sequences [8, 5]; data mining based approaches generate rules from system call sequences [12, 11]; Hidden Markov Model (HMM) and Neural Networks were used [19, 6]; algorithms originally developed for computational biology were also introduced into this area. In [20], Wespi et al. presented a novel technique to build a table of variable-length system call patterns based on the Teiresias algorithm. Teiresias algorithm was initially developed for discovering rigid patterns in unaligned biological sequences [14, 4]. This algorithm is quite time and space consuming when applied on long traces containing many maximal patterns. Wespi et al. announced that their method worked better than N-gram. However, N-gram generated the highest scores it could possibly generate on all their intrusion traces. This may suggest the attacks they chose are inherently easy to detect. So although Wespi's method generated higher looking scores, this does not necessarily mean it works better.

Cowan et al. proposed a method, called StackGuard, to detect and prevent buffer overflow attacks [2, 3]. StackGuard is a compiler technique for providing code pointer integrity checking to the return address. The basic idea is to place a "canary" word next to the return address on the stack, and check if this word was modified before the function returns. This is a good idea and may work well with buffer overflow attacks, but it is not effective in detecting many other kinds of attacks.

All methods described above have their advantages and disadvantages. In the next section, we will develop a new method that combines some advantages of the automaton based methods while avoiding their problems. Our method trains the model by monitoring at runtime, so it is closer to the FSA method.

3 The VtPath Model

Although closely related, our method has many properties that the FSA method does not possess. It uses call stack history as well as the current PC information. This can help detect more intrusions. It explicitly lists which function boundaries a transition traverses. This makes the model more precise. Our method is able to handle many implementation issues, such as signal handling. These issues were not considered for the FSA method. Also, our method handles DLL functions just like statically linked functions. This avoids the potential problems for the FSA method related to its unnecessary simplification. Our model is called VtPath because one main concept we use is called virtual path.

3.1 Background

Each instruction corresponds to a distinct program counter. However, it is neither necessary nor possible in efficiency to follow all these program counters. The FSA method records the program counter information at each system call. This is a good choice because system calls are where the program interacts with the kernel. In our approach, we also record program counter information at each system call. In the future, we may record information at other places as well, for example, when each jump or function call instruction is executed. We make the following assumption:

Assumption *The program counter and call stack can be visited with low runtime overhead when each system call is made.*

Using kernel-level mechanism to intercept system calls can achieve low runtime overhead. Our experiments later will show the overhead for pure algorithm execution is actually very low.

We will discuss how to handle DLLs later in the paper. In this section we assume all the functions that the program invokes are statically linked. We use relative program counters because the program may be loaded at different places for different runs, but the relative positions within program memory space will remain the same.

3.2 Virtual Stack Lists and Virtual Paths

As each system call is made, we extract the system call name and the current PC, as the FSA method does. In addition, we also extract all the return addresses from the call stack into a *virtual stack list* $A = \{a_0, a_1, \ldots, a_{n-1}\}$, where n is the number of frames in the call stack, and a_{n-1} is the return address of the function last called. The current

PC is then added into the list A as item a_n. For example, assume a function $f()$ is called within the $main()$ function. Then there are three elements in the virtual stack list when a system call in function $f()$ is made. a_0 and a_1 are the return addresses of $main()$ and $f()$, respectively; a_2 is the current PC. The virtual stack list denotes a history of all unreturned functions.

Our model uses a *virtual path* to denote a transition between two system calls. Assume $A = \{a_0, a_1, \ldots, a_n\}$ and $B = \{b_0, b_1, \ldots, b_m\}$ are the virtual stack lists for the current and the last system calls, respectively. Note that the two system calls may be called in different functions. We compare the lists A and B from the beginning, until we find the first subscript l so that $a_l \neq b_l$. As shown in Figure 1, the virtual path between the two system calls is defined as:

$$P = b_m \rightarrow Exit; \ldots; b_{l+1} \rightarrow Exit; b_l \rightarrow a_l; \\ Entry \rightarrow a_{l+1}; \ldots; Entry \rightarrow a_n \qquad (1)$$

where $Entry$ and $Exit$ are two specially defined PCs denoting the entry and exit points of any function.

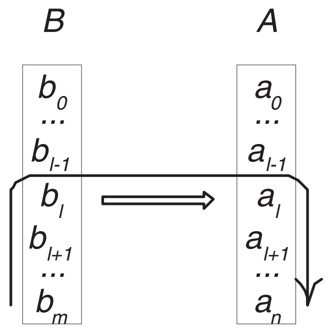

Figure 1. The virtual path from the last system call to the current system call, whose virtual stack lists are B and A, respectively.

The definition of the virtual path abstracts the execution between two system calls. The program sequentially returns from some functions (corresponding to the return addresses b_{m-1} to b_l), and then gradually enters some other functions (corresponding to the return addresses a_l to a_{n-1}). We traverse the virtual stack lists back to a common function (corresponding to the return address a_{l-1} and b_{l-1}, which are equal), below which both system calls are made.

For recursive functions, the control flows generally are very data-driven. The virtual stack lists obtained may be different for each distinct set of parameters, which results in a lot of distinct virtual paths. This could make training harder to converge or result in higher false positive rates. We modified our method to avoid this problem. A common property of recursion in virtual stack lists is that the same return address occurs repeatedly. When our method finds out that a pair of return addresses are the same, all the return addresses between them are removed from the virtual stack list, including one end of the pair. This reflects the fact that we only record each function at most once in the call history.

3.3 Training Phase

During training, we use a hash table, called RA (return address) table, to save all the return addresses ever occurred in the virtual stack lists of system calls. If the return address is the last item in a virtual stack list (the current PC item), the corresponding system call number is saved with it. Another hash table, called VP (virtual path) table, is used to save all the virtual paths. Virtual paths are denoted in a compact string form.

The return addresses and virtual paths are gradually added during many normal program runs. For each run, we assume there is one null system call with empty virtual stack list before the first real system call, and another one after the last real system call. The virtual path between the null system call and the first real system call, whose virtual stack list is $A = \{a_0, a_1, \ldots, a_n\}$, is:

$$P = Entry \rightarrow a_0; \ldots; Entry \rightarrow a_n \qquad (2)$$

The virtual path between the last real system call, whose virtual stack list is $B = \{b_0, b_1, \ldots, b_m\}$, and the null system call is:

$$P = b_m \rightarrow Exit; \ldots; b_0 \rightarrow Exit; \qquad (3)$$

3.4 Online Detection Phase

After training, we can use the hash tables to monitor the program execution online. As each system call is made, we record its virtual stack list. Like in the training, we also assume there are null system calls at the beginning and the end of the program run. There may be several types of anomaly phenomena:

- If we cannot get the virtual stack list, the stack must be corrupted. This is a *stack anomaly*. This kind of anomalies often happens during a coarse buffer overflow attack.
- Assume the current virtual stack list is $A = \{a_0, a_1, \ldots, a_n\}$. We check whether each item is in the RA table. If any return address is missing, this is a *return address anomaly*.
- If a_n does not have the correct system call, this is a *system call anomaly*.

- We generate the virtual path between the last and the current system call according to the equations (1), (2) or (3). If the virtual path is not in the VP table, this is a *virtual path anomaly*.

One problem for the FSA method is that the intruder could possibly craft an overflow string that makes the call stack looks not corrupted while it really is, and thus evading detection. Using our method, the same attack would probably still generate a virtual path anomaly because the call stack is altered. Our method uses and saves more information in training, so it is harder for attacks to evade detection.

3.5 Impossible Path Problem

Our method can help solve the impossible path problem mentioned before. Assume the attacker can somehow modify the return address within a function $f()$, so that the program enters function $f()$ from one call point and exits from another. This will not trigger an alarm for the FSA method because all the transitions are legal. Our experiments later will show carefully designed attacks exploiting this problem can fool callgraph and abstract stack methods as well. This kind of attacks can help the intruder because some critical part of the program could be jumped, for example, some permission checking code. The intruder can also use the technique to repeat the execution of some program part to create race conditions.

Our method will disallow the virtual path between the last system call before the call stack alteration point and the first system call after the alteration point. This is because in the call stack, the return addresses of function $f()$ for these two system calls will be different. These two return addresses will be included, resulting in an invalid virtual path.

4 Implementation Issues

Wagner et al. addressed some implementation issues for their statically generated models [17]. If not handled properly, these issues will also affect the effectiveness of the dynamic monitoring approaches. However, Sekar et al. only addressed one of the issues regarding DLLs [16]. Moreover, we believe that their method simplifies the behavior of DLLs so much that many intrusions on these DLLs may be missed. We find that some of these implementation issues are much easier to handle at runtime than at static analysis because some information is only available at runtime. Wagner et al. also pointed out this as the second principle in their paper [17].

4.1 Non-Standard Control Flows

For optimization and convenience, some non-standard control flows, such as function pointers, signal handlers and *setjmp()* function, are often used in programming. Wagner et al. stated that these features are always used in real applications of interests for intrusion detection [17]. They also found that function pointers and *setjmp()* are extensively used in some library functions.

Signals A signal handler is a function that is invoked when the corresponding signal is received. The program suspends the current execution and turn to the signal handler. It continues the execution from the suspended point after the signal handler returns. It is hard to consider signal handling in the model because a signal may occur anytime during the program execution. The problem is further complicated if signal handlers can be called within each other. If we treat signal handler calls as ordinary function calls in training, there will be false positives when signals occur at new places.

When the first system call in a signal handler is executed, we save the information about the last system call, including its virtual stack list. The last system call is then set to the null system call. When the signal handler returns, we restore the information about the last system call. This framework can be easily extended for the multi-level signal handler case. Each execution of signal handlers is treated like a program run. The same techniques used for training and online detection before can still be applied here with signal handlers.

For Linux, when a signal handler is called upon the receipt of a signal, a *sigreturn* system call is inserted into the stack frame. It will be called when the signal handler returns, to clean up the stack so that the process can restart from the suspended point. If we find a new *sigreturn* system call in the call stack when a system call is made, we know a signal handler was executed. If we encounter *sigreturn*, the signal handler just returned. Our method is simpler than Wagner's method because it does not need to monitor the signals received or signal handler registrations.

***setjmp()/longjmp()* calls and function pointers** The *setjmp()/longjmp()* library functions are useful for exception and error handling. The *setjmp()* call saves the stack context and environment for later use by *longjmp()*, which restores them. After the *longjmp()* call is finished, the execution continues as if the corresponding *setjmp()* call just returned. Function pointers are used to dynamically choose a function to call at runtime.

It is hard to handle them statically because it is hard to predict statically the value of a function pointer, or which *setjmp()* call is corresponding to a *longjmp()* call. Wagner et al. can only come up with some rough solutions that make the model more permissive than necessary or add more nondeterminism because their methods do not train at runtime. For our method, there is no such problem because it does not need to be aware of function pointers or the library calls. In detection phase, if a new function is called through a function pointer, or a new *longjmp()/setjmp()* pair appears, our method will generate an anomaly. It is reasonable to generate an anomaly here because some new situations have happened that never occurred before in training.

4.2 Dynamically Linked Libraries

One problem for both our method and the FSA method is related to dynamically linked libraries (DLLs). The difficulty is that the functions within DLLs may be loaded at different relative locations (comparing to the static portion) for different program runs, so the program counters may change from run to run. The methods Wagner et al. proposed do not have the above problem because they do not use PC information for online monitoring.

The FSA method tried to solve this problem by traversing the stack back to the statically linked portion. Using the virtual stack list concept, this means that the FSA method uses the last item in the list that is in the statically linked portion as the state. The behavior of a function in DLLs is simplified to a list of system calls that can be generated by this function and all functions it called. There will be states that have many transitions pointing to themselves labeled with these system calls. This simplifies the model for DLLs a lot. However, intrusions may also occur in the DLLs. For example, the intruder may install the Trojan version of a DLL. The FSA method may make the model for DLLs too simple to detect these intrusions. As detailed in a later section, for security critical daemon programs in our experiments, most system calls are actually made in DLLs.

We model the functions in DLLs just like any statically linked function. During training, we use a "block" lookup table to save the information for each executable memory block of each forked process, including the start address, the block length, the name (with full path) and the offset of the file from which the memory block was loaded. We use the block length, file name and offset but not the start address to match the same memory blocks in different runs. When we get a return address, we can use the block lookup table to decide which memory block it is in and what the relative PC within the block is. These two pieces of information together can uniquely distinguish a return address. Each return address is denoted by a global block index and an offset within the block.

There can be another kind of anomaly: *block anomaly*. This happens when we cannot match a memory block to any memory block occurred during training. This can be because the intruder is trying to load another DLL.

For Linux, there is a pseudo file named "maps" under the process's information pseudo file system "/proc". This file contains all the memory block information we need. There are structures containing similar information under other flavors of UNIX, such as Solaris.

Using the above approach, we can match a dynamically loaded code block to the same code blocks in other runs, although this block may be loaded to a different place. A return address can be uniquely distinguished, and the functions in DLLs can be modeled and checked just like statically linked functions.

4.3 Threads

Currently, there are many different ways to implement threads. As far as we can distinguish which thread generates a system call, there is no problem for applying our method on multi-threaded applications. For Linux, different threads actually have different process IDs, so we can distinguish threads by distinguishing their IDs. For other flavors of UNIX, we can try to find other ways to distinguish threads.

5 Experimental Evaluation

In this section, we present results from our experiments. We first describe the experiments on comparing VtPath with FSA in terms of convergence time, false positives, overhead, and detection of common root exploits. We then describe the experiments on evaluating the effectiveness of VtPath against some attacks, including impossible path exploits, that can evade several other detection models, and discuss the lessons learned.

5.1 Experiments on Comparing VtPath with FSA

Sekar et al. conducted experiments on normal data for some security-critical daemon programs [16]. They showed FSA uniformly worked better than N-gram in the sense of convergence speed, false positive rates and overhead. We conducted similar experiments to compare our VtPath method with FSA.

If all functions are statically linked, for the same program run, whenever there is a new transition for FSA, there is also a new virtual path for VtPath. This is because the virtual path contains all the information of the corresponding transition. So generally speaking, virtual paths are more specific than transitions. For VtPath, we should expect a slower convergence speed, a higher false positive rate and a higher detection rate. When DLLs are involved, the situation is somewhat complicated. As FSA simplifies DLL behavior model, it should have even faster convergence speed, fewer false positives, and lower detection rate. But there are situations where the simplification may also increase the convergence time and false positives. This is because one intra DLL function transition may map to different transitions at different DLL function call points, due to stack traverse to the statically linked portion. The situation becomes severe if some frequently called DLL functions have many intra-function transitions. For programs using a lot of signal handling, the convergence time and false positive rates of VtPath will benefit from its signal handling mechanism.

We conducted experiments on security critical daemon programs `ftpd` and `httpd`. We used the original FSA implementation from the authors, and compared VtPath with it. For fairness, all the comparison was based on data collected from the same program runs. The experiments were conducted on a RedHat 7.2 workstation with dual Pentium III 500MHz processors. We used WebStone 2.5 benchmark suite to generate HTTP traffic [13]. The files and visiting distribution were copied from our laboratory web server. For FTP experiments, we wrote some scripts using the "expect" tool. These scripts can execute commands that mimic some common user activities, such as downloading or uploading files and directories. The scripts were randomly executed to exercise the FTP server. The files were copied from one lab user's home directory.

We found out some bugs in the original FSA implementation, which contributed to higher false positive rates and slower convergence. We modified the programs and created our own FSA implementation. We will present our results for the VtPath implementation and both FSA implementations.

5.1.1 Convergence

The training process is considered as converged if the normal profile stops increasing (i.e., with no new behavior added). The convergence speed is important because the faster the training converges the less time and effort are needed. For FSA, the normal profile consists of states and transitions. There is always a new transition whenever there is a new state because the state is a part of the corresponding transition. The above statements are also true for VtPath if we use "return address" instead of "state" and "virtual path" instead of "transition". Therefore, we believe that the number of virtual paths or transitions is a good metric to measure convergence speed because the profile stops increasing if this number stops increasing. These numbers are plotted against the numbers of system calls used for training, which are presented in logarithmic scale.

We made a program to start the daemon program, and simultaneously record the traces for both methods. When the number of system calls made exceeds a preset limit, the program stops the daemon program. By this way, we generate traces with different lengths. We apply these traces incrementally starting from the shortest traces for training on both methods. Every time a trace is applied, the profile is copied and the convergence metric is calculated.

Figure 2 shows the results for `ftpd`. The solid line with star marks is for VtPath; the dashed line with square marks is for our FSA implementation; the dotted line with circle marks is for the original FSA implementation. The interesting thing is that the number of virtual paths actually increases more slowly than the number of transitions. This may be due to the DLL or signal handling related issues discussed at the beginning of this section. The original FSA implementation generates even more number of transitions. For VtPath and our FSA implementation, the profile increase stops after about 5M system calls are processed. The `httpd` experiments show similar results in terms of the comparisons between the methods.

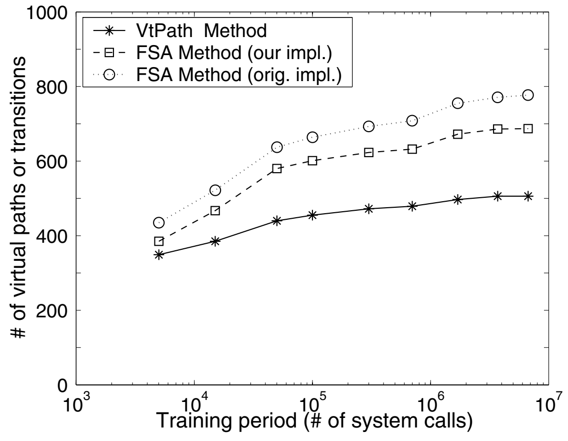

Figure 2. Convergence on `ftpd`

In our `httpd` experiments, less than 1% of the system calls are actually made in the statically linked portion. For more than 30% of the system calls, FSA has to go back at least 3 call stack frames to find a return address in statically linked portion. This means at least 3 levels of DLL functions are called for 30% of the system calls. These facts may suggest that DLLs are very important, and the simplification of DLL behavior by FSA may have severely impaired its detection capability.

5.1.2 False Positives

For `ftpd` experiments, we collect several normal testing traces ranging from 1M to several million system calls for each method, with a script execution distribution slightly different from what was used for the convergence experiments. As Sekar et al. argued in [16], this is to account for the fact that things may change between the training and detecting times. We use the profiles saved in the convergence experiments to analyze these testing traces. Like what Sekar et al. did in [16], each mismatched return address (state) or virtual path (transition) is counted as a false positive. The false positive rates are calculated as the number of false positives over the number of system calls in each trace, and averaged over the several testing traces for each method.

Figure 3 shows the relationship between the average false positive rate and the number of system calls used for training for `ftpd` experiments. Note both axes are in logarithmic scale. VtPath has almost the same false positive rates as our FSA implementation. Actually, using the profiles corresponding to more than 1M system calls, there is no false positives on all testing traces. The original FSA implementation generates much higher false positive rates at most points. Our `httpd` experiments show similar results in terms of comparisons between the methods.

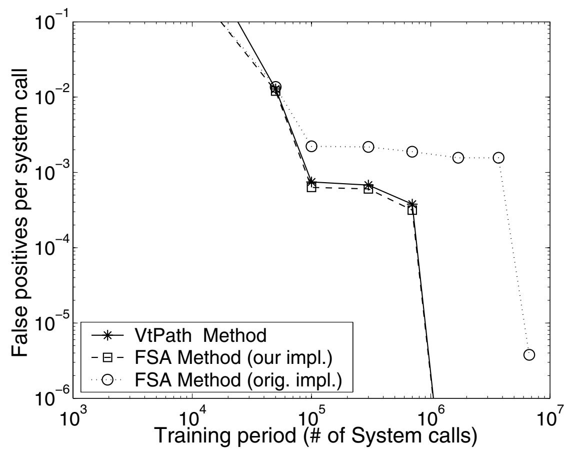

Figure 3. False positive rates on `ftpd`

5.1.3 Runtime and Space Overhead

We use the same user-level mechanism to intercept system calls as FSA. As pointed out by Sekar et al. [16], this mechanism incurs relatively high runtime overhead. They estimated that system call interception incurs 100% to 250% runtime overhead, while the overhead of their algorithm execution is only about 3%. For real applications, we want to use kernel-level mechanisms that incur much lower system call interception overhead. In this section, we only compare the algorithm execution overhead for both methods.

We use the average process time per system call stop to evaluate the algorithm runtime overhead. In our experiments, for FSA, the value is about 350 milliseconds for training and 250 milliseconds for detection. For VtPath, the value is about 150 milliseconds for both training and detection. It is interesting that the VtPath algorithm actually executes faster because, theoretically, it should be a little bit slower since it needs to do more work. The reason may be that we paid much attention on efficiency for our VtPath implementation.

The space overhead for VtPath, however, is higher than FSA. This is because it needs to save more information of call stack. For our `ftpd` experiments, the profiles that the FSA code creates are about 10K bytes, while the profiles that the VtPath code creates are about 30K bytes. These profiles should require less spaces when loaded into memory because the profiles saved on disk are human readable.

5.1.4 Detection of Common Exploits

We have also tested VtPath and FSA against several recent local and remote root exploits, namely efstool (local root), dhcpd v3.0.1rc8, and gdm v2.0beta1-4. Both VtPath and FSA successfully detected all of these exploits in our experiments.

5.2 Impossible Path Exploits and Beyond

We implemented two example attacks, including an impossible path exploit first introduced in [17], to demonstrate the effectiveness of the VtPath approach. The attacks were realized and tested on a RedHat 7.3 machine.

We evaluate the implementation of our approach as well as related approaches such as abstract stack, N-gram, callgraph, and FSA, under the same conditions to determine how effective the approaches are against the test attacks we develop. In our experiments, we use working implementations of N-gram and FSA we received from the authors (the bugs we found do not impair the detection ability of FSA for our test attacks). For abstract stack and callgraph, we do not have access to the implementations and thus we do all the tests manually using the algorithms described in [17].

Our approach is able to detect both attacks 1 and 2 described below. None of the other approaches we analyze detect either of them. We have also tested our approach against the mimicry attacks described in [18]. We find that our approach as well as FSA is able to detect the mimicry attacks. However, we find a way to improve the mimicry attacks to make them invisible to FSA. We achieve this by manipulating the stack contents that are analyzed by FSA implementation in such a way that FSA will mistakenly trace back to a legitimate system call in the text segment. The *masked mimicry attack* we develop will not be detected by the FSA implementation. VtPath, however, will still be able to detect both the original and the masked mimicry attack. We plan to describe our improvements of mimicry attacks to evade IDS in a separate paper.

In Section 5.2.3 and Section 5.2.4, we will present our critique of the attacks 1 and 2 we implemented and consider some general ideas behind the possible attacks against the detection models discussed in this paper.

5.2.1 Attack 1

As mentioned earlier, due to the lack of precision of many program execution models, it may be possible for an attacker to jump a security-critical section of a program code without being detected by IDS. We refer to the class of attacks that exploits this vulnerability as impossible path exploits (IPEs). The attack 1 we implement belongs to the IPE class. To the best of our knowledge, this is the first working implementation of the IPE attack.

Attack description. The attack works as follows. Consider *login_user()* function, shown in Figure 4. There are two possible execution paths in this function because of the *if()* statement. If *is_regular(user)* returns true, path number one is followed. Otherwise, path number two is followed. Suppose the function read_next_cmd() called at (I) contains an overflow at the *strcpy()* statement. Then, an attacker can substitute the return address of the function so that the *read_next_cmd()* returns to (II), the address where the other read_next_cmd() would otherwise return.

```
void   read_next_cmd(){
  uchar input_buf[64];
  umask(2);                    // sys_umask()
  ...
  // copy a command
  strcpy( &input_buf[0], getenv( "USERCMD" ));
  printf( "\n" );              // sys_write()
}
void login_user(int user){
  if( is_regular(user)){
    // unprivileged mode
    read_next_cmd();  // (I), this function will
                       // be overflowed
    ...
    // handle commands allowed to a regular user
    return;
  }
  // privileged mode
  read_next_cmd();     // (II), this function call
                       // will be skipped
  // --> this is where the control will be
  // transferred after a ret in read_next_cmd() at (I)
  seteuid(0);
  system( "rsync /etc/master.passwd ok@aeou.com:/ipe" );
  // and other privileged commands accessible only to
  // superuser
}
```

Figure 4. Pseudo code for attack 1

None of the existing models except VtPath will be able to differentiate between the *sys_write()* called when *read_next_cmd()* at (I) is called and the *sys_write()* called when *read_next_cmd()* at (II) is called. Consequently, because of imprecision of the models, including the ones for N-gram, abstract stack, callgraph, and FSA, after the jump an IDS would not detect an anomaly. The IDS would think the program has followed a legitimate execution path number two.

VtPath can detect the attack since in addition to verifying program counters and state transitions, it also sees stack context for both invocations of *read_next_cmd()*. More specifically, it can see an invalid virtual path from *sys_umask()* to *sys_write()* in *read_next_cmd()* at (I), as the return address of *read_next_cmd()* is changed by the overflow in *strcpy()*.

5.2.2 Attack 2

Attack description. This attack works as follows. *f()*, shown in Figure 5, is called from main() twice for the following two operations - checking a user name and checking a password. *f()* selects which operation to perform based on its parameter. The parameter is saved in a variable, *mode*. The variable is modified by an attacker when the adjacent local buffer, *input*, is overflowed. The local buffer is overflowed with a valid username and trailing zeros so that when *f(1)* is called, the value of mode is changed to zero. Under attack, instead of checking a user name and then checking a password, *f()* checks a user name twice. As a result, an attacker obtains access without knowing a password.

This attack will be detected by VtPath because it will see an invalid path between the *sys_close()* when *f(1)* is called and the following *sys_write()* in *main()*. N-gram, abstract stack and callgraph models will not be able to detect the attack because both branches in *f()* have the same system calls and the system call sequence stays unchanged during the attack. FSA will miss the attack because the transition from *sys_close()* to *sys_write()* is a valid FSA transition.

5.2.3 Observations

Based on the two attacks we described above, we can make the following general observations. First, both attacks require a way to change the control flow of a program. For our sample attacks we use buffer overflows. We realize that buffer overflows are not always possible and will eventually become a less significant threat. However, we believe our choice is justified given that over two-third of CERT's advisories in recent years were buffer overflows [15].

Second, programs that are vulnerable need to have a specific structure allowing, for example, a critical section to be jumped. In attacks described above, we show two examples of the possible program structures that can be exploited, namely a security-critical *if()* or a function whose argument controls execution and can be overflowed. For the IPE in Attack 1, it is also necessary that there be a function that is called from more than one point in a program.

```
f(int arg){
  int  mode = arg;              // this variable is overflowed
  char input[10];
  fopen();                      // sys_open(), open passwd file
  // overflow, changes 'mode' variable => execution flow
  scanf("%s", &input[0] );
  if( mode == CHECK_UNAME ){ // check username?
   fread();                     // sys_read(), read from passwd file
   fclose();                    // sys_close()
   if( is_valid_user(input) ) ret = 1; else ret = 0;
  }
  else if( mode == CHECK_PASSWD ){ // check password?
   fread();                     // sys_read(), read from passwd file
   fclose();                    // sys_close()
   if( is_valid_pass(input) ) ret = 1; else ret = 0;
  }
  return ret;
}
void main(){
  printf( prompt );         // sys_write()
  ret=f(0);                 // (I), read/check username
  if( ret ) ret = f(1);     // (II), read/check password
                            // if username was correct
  printf( "Authenticated\n" ); // sys_write()
  if( ret )
   execve( "/bin/sh" );         // superuser mode
}
```

Figure 5. Pseudo code for attack 2

```
f(){
 ...

 // read in some large string z, syscalls are fine here
 f0();

 // important: f1() has no system calls
 // it copies z to x, z is larger than x, so x is overflowed;
 // after the ret instruction, the overflow code can
 // jump anywhere within f(), as long as it is between
 // f1() and the next system call;
 // for example, the code can jump to IP1
 f1();

 if( cond ){
  // regular user privileges
  ...
  return;
 }
...
 IP1:
 // superuser privileges
 execve( "/bin/sh" );
}
```

Figure 6. Pseudo code for granularity attack

When the control flow of a vulnerable program is changed as in Attack 1, the function is exploited and a jump occurs.

5.2.4 Generalizations

The attacks we describe here have a common property in that they take advantage of the inherent limitations, or the insufficient level of granularity, of the IDS model. The information (or audit data) as well as the modeling algorithm used by an IDS model can be inadequate in a such a way that some attacks do not manifest as anomalies. For instance, attackers can exploit the fact that many anomaly-based IDS only check their program behavior models at a time of a system call [16, 17, 8]. Consider the example in Figure 6. This attack will not be detected by any of the approaches we described so far. VtPath will also be unable to detect the attack unless the IP1 is somewhere else in the program at a different level of nestedness so that there is an anomaly in the stack contents that can be detected.

As [10, 9] proposed and [17] pointed out, it is important that the intended behavior of a program is taken into account in a model. If a program comes with a complete specification of its intended behavior, any attack that causes the program to behave differently or violating the specification can be detected, provided that an IDS can check the program behavior against the specification precisely. For our purposes, such an IDS will be considered to have a *maximal* level of granularity because it can detect all attacks that cause the program to deviate from its intended behavior. In most cases, an IDS has an inadequate level of granularity and thus there are always attacks on the program that can evade detection.

5.2.5 Importance of IPEs

We recognize that a successful execution of the attacks we described above is contingent upon quite a few variables and may not always be possible. It can be tempting to dismiss the problem of IPEs altogether as having little relevance since finding an existing piece of code that is exploitable may not be easy. Besides, as with many other attacks, the attacker is constrained by the need to perform reconnaissance and to have access to the details of the environment on the attacked host, particularly the IDS and other protection tools used.

We must point out, however, that instead of looking for vulnerable code, attackers can introduce IPE-vulnerable code into open source products in the form of innocent improvements or legitimate bug fixes. In contrast to other security flaws that attackers may attempt to inject, changes needed for IPEs can be made very subtle, which makes them less likely to be detected by code inspection. One of the reasons is that it is typically the structure of the code that makes it vulnerable to IPE, not the actual commands. Furthermore, it seems natural to assume that attackers will do everything in their power to disguise the IPE-vulnerable code. This can be done, for example, by gradually shaping the structure of a program code over series of patches.

6 A Comparison of System Call Based Anomaly Detection Approaches

In this section, we compare several anomaly detection methods based on their underlying principles. These methods include N-gram, FSA, VtPath, callgraph, abstract stack, and the method Wespi et al. proposed [20] (We call it Var-gram because it uses variable length N-gram patterns). The principles of the methods proposed in [7] are the same as those of callgraph and abstract stack; thus, our analysis on callgraph and abstract stack can also be applied to these methods. Our comparison is based on the algorithmic approaches of the models as well as the types of information they use. We analyze their performance characteristics in terms of false positives, detection capability, space requirement, convergence time, and runtime overhead.

We also realize that the performance of the methods can vary a lot due to their implementation details, such as issues regarding signals, DLLs and system call parameters. For example, some detection approaches are equipped with mechanisms to predict static system call parameter values. These mechanisms can also be applied to other detection approaches with appropriate modification, either through static analysis or dynamic monitoring. We can also develop appropriate mechanisms regarding other implementation issues for each approach. In this section, we ignore all the implementation issues, and focus on the underlying principles.

State based approach and information captured. We can model the execution of a program using a state diagram. At the start of the program, the system is in the start state. At each event occurrence, the system transits from one state to another. At any point, it is in a valid state if and only if the start state was valid and all the intermediate transitions were also valid. Consider an instantiation of the monitored program. To capture the normal behavior, the model tries to capture the valid states and valid state transitions by monitoring the behavior of the program at different event points. The model should also ignore the variables that are specific to that particular run of the program. It tries to learn the behavior of program by generalizing the observed instances of the program. However, it is not feasible to monitor the program at every event. For the approaches we study here, the states of the system are recorded only at the point of system calls. The decision to monitor only at system calls is justifiable because many attacks can manifest at the system call level.

Possible variables which could be considered while defining the states of the system include "contents of data heap", "registers", "code segment", "program stack", "system call and its arguments" and other system variables. The objective of a model is to record only the relevant state variables. Using the state transition diagram of each run during the training period, we would like to build a generalized state transition diagram which represents the normal behavior of the program. Data heap and register values are highly specific to that particular run of the program and do not generalize well, so we can ignore them. Code segment might be useful in some cases. System calls and their arguments are certainly useful. Although some arguments of some system calls are worth recording, many arguments can have many possible values, resulting in a model with slow convergence and high overhead. Call stack is important for learning the flow of program. In general, using more information to develop the intrusion detection model helps in attaining better detection rate. But it may also cause higher runtime overhead and more false positives.

N-gram and Var-gram choose to record only the system calls. N-gram records fixed-length sequences of system calls that occurred in the training data. Var-gram extracts variable-length sequences of system calls that occur repeatedly. FSA chooses to store the current system call along with its PC. The involvement of PCs makes it possible to distinguish system calls with the same name but called at different locations (location sensitive). VtPath keeps additional entries from the call stack, which further distinguishes system calls called in different contexts (context sensitive). N-gram can achieve some characteristics of location or context sensitive by using larger N. We believe that VtPath has better tradeoff considering the performance of N-gram, FSA, and VtPath in experiments. Although they are state based, abstract stack and callgraph models use a different approach of learning the behavior of the program by statically analyzing the source code. They only concern about system calls at detection time as N-gram and Var-gram do.

False positives False positives depend on how well the model captures the normal behavior of a program while ignoring the information that does not generalize well. Callgraph and abstract stack models do not have any false positive because they are statically derived from the source code, and all possible paths of execution are encoded in the grammar of the model. N-gram and Var-gram record sequences of system calls that occur in the training data. Any path which is not covered in the training set may produce a new sequence, thus raising a false positive. For N-gram, the probability of the alert depends largely on the size of N.

The larger N is, the higher is the probability that new paths will generate new N-length sequences. FSA tries to model programs more accurately by taking into account the locations of system calls. This is logical because the location of a system call determines what system call will be executed. The model may generate a false positive if any valid system call location or any valid transition between system call locations is not covered in training. VtPath on the other hand models the program more strictly because valid transitions must have valid return address combinations as well. So it should generate a little bit more false positives than FSA. Both FSA and VtPath essentially use diagrams. Comparing to N-gram, the location or context sensitive property will increase false positives, but on the other hand the digram property will decrease false positives when comparing to N-gram with large N.

Detection capability In Section 5.2, we presented a few specific attacks which will be missed by some detection approaches while detected by others. Detection capability of an IDS depends on its granularity, which in turn is determined by the amount of relevant information the IDS is storing and its modeling technique. An IDS with more granularity should have better detection capability.

All the approaches we study here try to model the system call behavior of the program. Any attack that introduces a new system call or causes very noticeable changes in the system call sequences (e.g., common buffer overflow attacks) should be detected by all the approaches. It is easier for FSA and VtPath to find Trojan horses because program counters for system calls and return addresses for function calls will probably change with the change in the code, while system call sequences may not. FSA and VtPath can also detect all the attacks where any system call is made from invalid points. All other approaches will miss these attacks if the system call sequences do not change. VtPath provides another level of protection because it is hard to jump or skip to another place in the program by changing return addresses while avoiding detection. Attacks which have no effect on system call sequences and return addresses will evade all the approaches discussed here (if no frequency or parameter value information is used).

For N-gram and Var-gram, the detection capability depends on the statistical regularity of normal data and also the properties of attack, in particular, how much the attack sequences deviate from that normal behavior. However, there is no concrete research done on what types of attack can be detected by N-gram and Var-gram. Due to the context-insensitive treatment of function calls, callgraph model allows IPE. As a result, all attacks that follow any of these IPEs will go undetected. Abstract stack model tries to remove this imprecision by including some context-sensitive information. However, our experiments showed that carefully designed IPEs can still evade detection by it. The non-determinism may impair the detection capability for both callgraph and abstract stack models. FSA checks the transition between the PCs of two system calls. It suffers from the same problems of the context-insensitive property as callgraph. In particular, IPE can evade FSA. VtPath stores all the system call points and all the allowed virtual execution paths. It can be evaded if an attack changes the call stack but somehow changes the virtual path to another valid one.

Space requirement For N-gram and Var-gram, the main space requirement is to store the system call sequences. That depends on the number of different sequences and also on the data structure used for storage. For example, storing sequences in the form of array generally takes more space, whereas tree structure takes less. For callgraph and abstract stack models, the space requirement is proportional to the number of NDFA transitions or the size of context free grammars (CFGs), which is proportional to the number of positions where function calls or system calls are made in the program code. For FSA, the memory requirement is proportional to the number of transitions in the automaton. The upper bound on number of transitions is proportional to the square of the number of places system calls are made in the program code. But in general, the number of transitions should be comparable to that of callgraph. For VtPath, the space requirement is driven by the number of virtual paths. In the extreme case, the number of virtual paths that pass function boundaries can be exponential to the number of function calls in program code. However, in general, the number of virtual paths is at the same level as the number of transitions for FSA or callgraph.

Convergence time By convergence time, we mean the amount of training time or data required to have a stable model. N-gram converges when most of the possible sequences are encountered in the training data set. This depends on the value of N. As N increases, the size of required training data increases, possibly exponentially. Var-gram converges when most of the "wanted" patterns appear repeatedly and are extracted. The Teiresias algorithm Var-gram uses is not suitable for incremental training usage, so we can only check the convergence by training on data sets with different sizes separately and comparing the resulting patterns. For FSA, we need to cover most of the possible states and possible transitions. It is not necessary to go through each path of execution. It therefore needs less data and time to form the stable model. Abstract stack and callgraph models do static analysis of the source code, so they do not require any training data. Also they need just one pass of the program. VtPath converges when most of the possible virtual paths are covered. This will require a somewhat larger data set than FSA. But as it is essentially based on diagrams with call stack attached, it should take less training data and time than N-gram with large N.

Runtime overheads Runtime overhead of IDS is due to system call interception and processing time of the IDS model. Because system call interception overhead is similar for all the models, here we discuss only the processing time of a model for each system call. N-gram and Var-gram need to check if there are matches in the database for the system call sequences starting from the current point. Using a trie structure, this can be done in time linear to the sequence length. For FSA (or VtPath), at each system call we need to check if it has a valid state (or valid return addresses) and there is a valid transition (or valid virtual path). Using a hash table this will take constant time.

Non-determinism will aggravate the runtime overhead for callgraph and abstract stack. In the callgraph model, there could be multiple paths from the current state. Using efficient techniques, we can cover all the next possible valid states in the time proportional to the number of states. So for each system call, the upper bound of time overhead is proportional to the number of states. In abstract stack model, for each system call we need to go through each possible path in the CFG to determine the possible next states and the stack contents. This may take exponential time in some cases.

7 Summary

Call stack can be very useful for intrusion detection purposes. In this paper, we developed a new method that can dynamically extracts return address information from the call stack and use it for anomaly detection. Our experiments show that this method is effective in terms of both detection ability and false positive rates. We also compared various related approaches to achieve a better understanding on what and why attacks will be missed by these approaches.

The main advantages of FSA and VtPath are that they are deterministic and location (context) sensitive. The main advantages of callgraph and abstract stack are that they can remove all false positives and do not require training. We may be able to combine these methods together and have all these advantages. Using binary analysis techniques similar to those in [7], we can extract and generate all the possible system calls (with the corresponding PCs), return addresses, and virtual paths from executables. The profile generated can then be used to dynamically monitor program executions. We can avoid false positives because the profile is generated by techniques similar to other static analysis techniques compared in this paper. The determinism and location sensitive properties are also kept. We will conduct more research on this subject in the future.

ACKNOWLEDGEMENTS We would like to thank Yong Huang for providing some related material and discussions. We would also like to thank John Levine for suggestions and corrections. The work reported in this paper is supported in part by ARO contract DAAD19-01-1-0610 and NSF contracts CCR-0133629 and CCR-0208655. The contents of this work are solely the responsibility of the authors and do not necessarily represent the official views of the U.S. Army and NSF.

References

[1] K. Ashcraft and D.R. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes", *IEEE Symposium on Security and Privacy*, Oakland, CA, 2002.

[2] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, "Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", *7th USENIX Security Symposium*, San Antonio, TX, 1998.

[3] C. Cowan, P. Wagle, C. Pu, S. Beattie and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", *DARPA Information Survivability Conference and Expo*, Hilton Head Island, SC, 2000.

[4] A. Floratos and I. Rigoutsos, "On the Time Complexity of the TEIRESIAS Algorithm", *Research Report 98A000290*, IBM, 1998.

[5] S. Forrest, S.A. Hofmeyr, A. Somayaji and T.A. Longstaff, "A Sense of Self for Unix Processes", *IEEE Symposium on Computer Security and Privacy*, Los Alamos, CA, pp.120-128, 1996.

[6] A. Ghosh and A. Schwartzbard, "A study in using neural networks for anomaly and misuse detection", *8th USENIX Security Symposium*, pp. 141-151, 1999.

[7] J.T. Giffin, S. Jha and B.P. Miller, "Detecting Manipulated Remote Call Streams", *11th USENIX Security Symposium*, 2002.

[8] S.A. Hofmeyr, A. Somayaji, and S. Forrest, "Intrusion Detection System Using Sequences of System Calls", *Journal of Computer Security*, 6(3), pp. 151-180, 1998.

[9] C. Ko, "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach". *PhD thesis*, UC Davis, 1996.

[10] C. Ko, G. Fink and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring", *10th Computer Security Applications Conference*, Orlando, Fl, pp.134-144, 1994.

[11] W. Lee and S. Stolfo, “Data Mining Approaches for Intrusion Detection”, *7th USENIX Security Symposium*, San Antonio, TX, 1998.

[12] W. Lee, S. Stolfo, and P. Chan, “Learning Patterns from Unix Process Execution Traces for Intrusion Detection”, *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, 1997.

[13] Mindcraft, “WebStone Benchmark Information”, http://www.mindcraft.com/webstone/.

[14] I. Rigoutsos and A. Floratos, “Combinatorial Pattern Discovery in Biological Sequences: The TEIRESIAS Algorithm”, *Bioinformatics*, 14(1), pp. 55-67, 1998.

[15] B. Schneier, “The Process of Security”, *Information Security*, April, 2000, http://www.infosecuritymag.com/articles/april00/columns_cryptorhythms.shtml.

[16] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors”, *IEEE Symposium on Security and Privacy*, Oakland, CA, 2001.

[17] D. Wagner and D. Dean, “Intrusion Detection via Static Analysis”, *IEEE Symposium on Security and Privacy*, Oakland, CA, 2001.

[18] D. Wagner and P. Soto, “Mimicry Attacks on Host-Based Intrusion Detection Systems”, *ACM Conference on Computer and Communications Security*, 2002.

[19] C. Warrender, S. Forrest, and B. Pearlmutter, “Detecting Intrusions Using System Calls: Alternative Data Models”, *IEEE Symposium on Security and Privacy*, pp. 133-145, 1999.

[20] A. Wespi, M. Dacier and H. Debar, “Intrusion Detection Using Variable-Length Audit Trail Patterns”, *3rd International Workshop on the Recent Advances in Intrusion Detection* , LNCS 1907, Springer, pp. 110-129, 2000.