1-1-2005

Combining Static Analysis and Dynamic Learning to Build Context Sensitive Models of Program Behavior

Zhen Liu

COMBINING STATIC ANALYSIS AND DYNAMIC LEARNING TO BUILD CONTEXT SENSITIVE MODELS OF PROGRAM BEHAVIOR

By

Zhen Liu

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2005

COMBINING STATIC ANALYSIS AND DYNAMIC LEARNING TO BUILD

CONTEXT SENSITIVE MODELS OF PROGRAM BEHAVIOR

By

Zhen Liu

Approved:

Name: Zhen Liu

Date of Degree: December 9, 2005

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Susan M. Bridges

Title of Study: COMBINING STATIC ANALYSIS AND DYNAMIC LEARNING TO BUILD CONTEXT SENSITIVE MODELS OF PROGRAM BEHAVIOR

Pages in Study: 175

Candidate for Degree of Doctor of Philosophy

This dissertation describes a family of models of program behavior, the Hybrid Push Down Automata (HPDA) that can be acquired using a combination of static analysis and dynamic learning in order to take advantage of the strengths of both. Static analysis is used to acquire a base model of all behavior defined in the binary source code. Dynamic learning from audit data is used to supplement the base model to provide a model that exactly follows the definition in the executable but that includes legal behavior determined at runtime. Our model is similar to the VPStatic model proposed by Feng, Giffin, et al., but with different assumptions and organization. Return address information extracted from the program call stack and system call information are used to build the model. Dynamic learning alone or a combination of static analysis and dynamic learning can be used to acquire the model. We have shown that a new dynamic learning algorithm based on the

assumption of a single entry point and exit point for each function can yield models of increased generality and can help reduce the false positive rate.

Previous approaches based on static analysis typically work only with statically linked programs. We have developed a new component-based model and learning algorithm that builds separate models for dynamic libraries used in a program allowing the models to be shared by different program models. Sharing of models reduces memory usage when several programs are monitored, promotes reuse of library models, and simplifies model maintenance when the system updates dynamic libraries.

Experiments demonstrate that the prototype detection system built with the HPDA approach has a performance overhead of less than 6% and can be used with complex real-world applications. When compared to other detection systems based on analysis of operating system calls, the HPDA approach is shown to converge faster during learning, to detect attacks that escape other detection systems, and to have a lower false positive rate.

DEDICATION

To my parents and brother.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of many people. My terrific adviser, Dr. Susan M. Bridges, has taught me to think and work as an effective researcher. She has helped me with endless patience and has given me invaluable feedback. I am indebted to her help and encouragement throughout my Ph.D. study.

I would like to thank Dr. Rayford Vaughn for his great security class that broadened my mind and his wonderful fiction collection that brought me much fun. I am thankful for the financial support and the research opportunities that he and Dr. Bridges have made available for me in the past years. I thank Dr. Julia Hodges for giving me the opportunity to pursue my Ph.D. degree in this wonderful country. I am also grateful to Dr. Yoginder Dandass and Dr. Eric Hansen who have served as my committee members and provided helpful advice.

I would like to thank the members in th Center for Computer Security Research at Mississippi State University. Special thanks are due to Wei Li, German Florez-Larrahondo, Ambareen Siraj, and Robert Wesley McGrew for their many helpful discussions.

Finally, I would like to acknowledge the National Science Foundation for supporting my research work at Mississippi State University (grant No. SCI04354-04090852).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Detailed models of program behavior have proven to be effective methods for detection of attacks that exploit software vulnerabilities and conduct malicious operations on the compromised systems. Because a program's access to low-level system resources is restricted to the system call interface, models based on the sequence of system calls issued by a program are a rich source of information about critical program behavior. Two basic approaches have been used to build models of program behavior based on system calls. The first approach, introduced by Forrest et al. [34] and studied by many others, uses audit histories to "learn" a model of normal sequences of system calls. The second approach, introduced by Wagner and Dean [76], derives a model of legal sequences of system calls from the program code. We refer the first approach as dynamic learning and the second as static analysis. Models acquired by dynamic learning from audit data often suffer significant problems with false positives. Models acquired with static analysis typically do not generate false positives, but these methods have been restricted to statically linked programs and they use approximation to acquire behavior defined at runtime. This dissertation describes a new family of models of program behavior based on systems calls. The models can be acquired by dynamic learning or by a combination of static analysis and dynamic learning and can therefore take advantage of the strengths of both approaches. A method is also provided for acquiring behavior of dynamically linked libraries by static analysis and for combining these models with models of other program components.

Although researchers have developed techniques for detection of vulnerabilities in programs prior to deployment [17, 18, 36, 52, 72, 74, 75], large software systems continue to be deployed with vulnerabilities. In addition, patches distributed by software vendors are often not installed properly or at all on deployed systems. Therefore, systems for run-time detection and confinement of attacks continue to be necessary to enhance global security [23]. Many intrusion detection systems monitor programs and detect attacks by comparing the runtime behavior of a program with a predefined model of program behavior. This method has proven to be effective for detecting attacks such as buffer overflow, Trojan horse, and format string that are often destructive and difficult to detect with other monitoring methods (e.g., monitoring user command behavior and system state) [16, 30, 31, 34, 39, 42, 47, 48, 50, 53, 58, 60, 67, 71, 76, 79, 80]. This type of attack is very common and poses serious risks to software systems. For example, buffer overflow vulnerabilities account for about 50% of all vulnerabilities reported by CERT [77].

Unlike the behavior of a human user or patterns of network traffic, the behavior of a program is exactly defined in its binary executable. Unless the executable image is modified on disk or in memory, program behavior should not change without the administrator's knowledge. “Thus, if intrusions can be detected as deviations from normal program behavior, such an intrusion detection technique would be free from false alarms caused by changes in user behavior patterns, and free as well from missed intrusions caused by attackers that mimic benign users" [60]. The goal of this dissertation is to develop an approach that combines the high accuracy of static-based models and the practical aspects of dynamic learning.

1.1 Models of normal program behavior based on system calls

The operating system kernel is the central component for computer resource management. In most current operating systems such as LINUX, UNIX, and Windows, user programs do not have the right to access hardware resources directly; instead, a privileged program called the kernel runs with full access to the hardware, and regular programs must ask the kernel to perform tasks on their behalf. Current operating system kernels use a system call interface to provide services such as file I/O, network I/O, and user privilege control. This system call interface is the only way that a user space program can communicate with the kernel and conduct low-level operations. Therefore, monitoring system call related behavior has several advantages: a) it provides a rich information source about the resource usage of a user program, b) no malicious operations can bypass the system call interface, and c) the techniques can be used in most current operating systems.

The two approaches for acquiring models based on system calls invoked by the program differ in the source of data used to construct the model. Dynamic learning builds a model of normal behavior by observing the runtime behavior of the program while static analysis builds a model of normal behavior by analyzing the binary executable or source code of the program. Many system call modeling methods such as those by Forrest et al. [34] and Wagner et al. [76] consider only sequences of system calls. The order relation between system calls is the key information used to identify anomalies in their methods. Sekar et al. [71] included the program counter as supplementary information for system call analysis so that system calls with the same name could be differentiated when invoked at different program positions. Feng et al. [32] and Gao et al. [37] further utilized information from the call stack to capture the context of function calls.

1.2 Motivation

Most applications that run on computer systems are too complex and complicated to be completely understood and thoroughly tested. These software systems almost always contain exploitable bugs that may lead to unauthorized access even when a trusted software development process has been used [10]. Even if the software does not have bugs, attackers can take advantage of memory errors to obtain unauthorized control of the system. Govindavajhala and Appel [44] have presented an attack which sends the Java Virtual Machine (JVM) a malicious Java program designed so that almost any memory error in its address space will allow it to take control of the JVM. In their experiments the faults were generated by heat. However, many other factors such as electromagnetic interference could cause the same type of problem. Monitoring of program behavior is required to detect such abnormal operations.

The goal of all approaches for system-call-based program monitoring is to confine the program execution so that the sequences of system calls invoked by a program conform to the execution flow defined in the program executable. The design of a good program model must balance four factors: effectiveness, efficiency, scalability, and flexibility. An effective model should be capable of detecting as many anomalies as possible with almost no false alarms and should be difficult for attackers to evade. It should be possible to apply the model efficiently so that it imposes a small performance overhead on the monitored programs. The learning algorithm must be scalable in order to handle large real-world applications and must be able to accommodate dynamically linked programs and other features of modern programming environments. The model should be sufficiently flexible to allow it to be adapted for different uses such as intrusion detection and anomaly detection.

Many program models are susceptible to both mimicry attacks [38, 78] and impossible paths [32, 76]. A mimicry attack takes advantage of the characteristics of an IDS that only considers the sequence of system calls. The attacks are conducted by embedding the malicious system calls into a sequence of system calls that is valid for the program model. Some of the system call arguments are manipulated to conduct the real malicious operations and others are nullified to allow the system calls to act as no-ops. All system-call-based IDSs described in [34, 39, 53, 60, 76, 79] are vulnerable to this type of attack. Models that take advantage of call stack information make it much more difficult to implement mimicry attacks [38].

The Non-Deterministic Finite Automata (*NDFA*) model proposed by Wagner et al. [76] is the first efficient model that captures most program control flow. However, this model is susceptible to the impossible path problem, an execution path that never appears in legal program execution but is acceptable by the program model. This problem occurs when returning from a function call where the function is called at least twice in the program. The *NDFA* model has no concept of function call context and cannot determine which of the possible return sites it should select as the next state.

Wagner and Dean proposed using a Push Down Automata (*PDA*) model to resolve the impossible path problem [76]. The *PDA* uses a stack to maintain the function call context. However, the *PDA* approach is too computationally expensive to be used in practice [41, 76]. More recent research including *Dyck* [42], *VPStatic*, [31], and *IAM* [43] has significantly decreased these overheads. However, with systems such as *Dyck* and *VPStatic* that monitor system calls, some monitored programs still incur overhead penalties of up to 50% when the models are enabled. The *IAM* approach is based on monitoring library calls and thus is easily bypassed by attacks that invoke system calls using a system interrupt directly.

The *NDFA* and *PDA* models, *Dyck*, *VPStatic*, and *IAM* are all acquired using a static analysis-based approach. Although static learning allows acquisition of nearly all possible execution paths defined in the binary executable, modern programming languages introduce a number of mechanisms such as virtual functions, exception handling, and dynamic linked libraries that make it difficult to capture of all execution flow by static analysis [41, 42, 71, 76]. Moreover, the complexity of performing static analysis on x86 binaries restricts the applicable targets of current tools [59, 66, 68, 70]. “This complexity stems from difficulties in code discovery and module discovery, with numerous contributing factors, including: variable instruction size; hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms; and indirect branch instructions such as *call/jmp reg32* that make it difficult or impossible to identify the target location” [38]. All static analysis-based approaches approximate some execution flows to overcome these difficulties. However, the approximations introduce new impossible paths. Previous static analysis approaches have not addressed scalability issues. If these approaches are to be used in modern computational environments, they must have the capability to handle large real-world programs with low overhead and be easily updated when the execution environment changes.

Although in many papers, the terms anomaly detection and intrusion detection are used interchangeably, these terms actually represent two different tasks. In some cases, it may be appropriate to consider some legal program behavior as non-normal. For example, in safety critical situations, program behavior that handles unusual faults probably should be considered anomalous. Dynamic learning is more suitable than static analysis for building models for this type of anomaly detection. Any behavior that does not appear in the training data will be considered an anomaly. Static analysis, on the other hand, is more appropriate for intrusion detection tasks where one of the requirements is to keep false alarms to a minimum. An anomaly detection approach can also potentially be used for finding hidden code in software. The widespread use of COTS software and outsourcing of software development has increased the potential for hidden ”back door” vulnerabilities. These risks are particularly important in computing environments where security is a high priority, e.g., military applications, control systems for critical infrastructure, etc. The size of modern software systems makes it physically and economically impractical to conduct a line by line inspection to determine if a back door has been inserted in a system. However, it is possible to build a model of normal behavior during black-box evaluation of the product and then to treat the problem of detecting hidden functionality as an anomaly detection problem. Previous approaches do not provide a single modeling framework that can be adapted for either intrusion detection or anomaly detection.

This dissertation describes a new family of program behavior models that can be acquired using dynamic learning or a combination of static analysis and dynamic learning. The following is a summary of the design goals of our work.

- Incorporation of a call stack information into the model: Return addresses on the program call stack provide a rich information source for intrusion detection, and models based on analysis of this information are more effective in detecting attacks and more difficult to evade than methods based only on sequences of system calls [32, 38, 71]. Previous approaches do not provide an acceptable trade off between effectiveness and efficiency. The *NDFA* method is efficient but is susceptible to the impossible path problem. The *PDA* approach can develop a model that almost exactly matches program control flow, but it is too computationally expensive. Although other approaches reduce the computational overhead, the overhead is still too high for the methods to be applicable in real-world applications. In this dissertation, return address information is incorporated into an automata based model to improve effectiveness and efficiency. We provide an analysis of mapping between execution flow and the model and have designed new epsilon reduction and anomaly recovery algorithms.

- Static and dynamic learning for model acquisition: Static analysis can be used to attain nearly all execution paths defined in the binary executable. However, modern programming languages use some constructs that change program flow and that cannot be captured by static analysis. These include signals, function pointers, *setjmp()* primitives, virtual functions, and exception handling. Although approximations have been developed to eliminate the false positives introduced by the above difficulties, these approximations also impair the detection ability of the model. Static analysis cannot capture the program behavior mentioned above because the programming language defines them as late-binding. For example, C++ defines virtual functions as a late-binding technique to implement polymorphism that determines the behavior of an object at runtime rather than compile time. Therefore it is impossible for a static learning algorithm to acquire this behavior. Such behavior can be captured by observing the runtime behavior of a program through dynamic learning. A system that can build models by dynamic learning from runtime behavior, or by a combination of static analysis and dynamic learning has the potential to produce models that exhibit a good trade off between false positives generated and detection ability and that can be configured for different purposes [57].

- Handling dynamic linked libraries: Dynamic linked libraries are widely used in modern operating systems. A dynamic library does not need to be linked with an executable image at link time; instead, it may be loaded dynamically at run time. This is a powerful concept that permits a properly written executable to link itself with almost any library without reconfiguration. A behavior model for one dynamic library can potentially be shared among the programs which make use of the same library. We have developed a new model construction framework that allows models of dynamic libraries to be constructed separately by static analysis or dynamic learning and then to be shared by many program models. A method is provided for combining appropriate component models to produce a complete model at compile time. Use of our new component-based model will reduce the size of the model produced for an individual program. Moreover, less training data will be required to build an accurate behavior model for a program since a large portion of the behavior exhibited by the dynamic library may have already been learned and can be shared by several program models. Because of the separation of models, the program models do not need to be completely relearned when a system updates dynamic libraries.

1.3 Hypothesis

The hypothesis of this dissertation is *that an automata model of program behavior supplemented with call stack information can be acquired by dynamic learning or by a combination of static analysis and dynamic learning and that the acquired model will have better accuracy (higher detection and lower false alarm rates) than similar models acquired only through static analysis or dynamic learning. A model construction framework that separates the program model into models for each execution component is effective for handling dynamic libraries and is applicable with real world applications.*

1.4 Contributions

The primary contribution of this dissertation is the development of a new family of program models that can be acquired by dynamic learning, or by a combination of static analysis and dynamic learning. The main contributions can be specifically described as follows:

- A new model that can be acquired by a combination approach: A novel program model, the hybrid pushdown automata (HPDA), has been developed. This model is a flow- and context-sensitive model that is as accurate as a PDA model. This model can be acquired by a combination of static analysis and dynamic learning. Static analysis, dynamic learning, and combination algorithms have been designed for this model.
- Component based model: A component-based model (CBHPDA) for dynamically combining models during runtime has been developed to handle dynamic linked libraries used in modern operating systems. The CBHPDA separates the program model into separate models for each execution component for the program.
- Analysis of the mapping between execution flow and models: This research provides an analysis of execution flow and defines the concept of an *execution region* that is the basis for mapping the models to program execution flow.

- Epsilon reduction algorithm: The *execution region* concept is used as the basis for designing a new epsilon reduction algorithm that removes unnecessary epsilon transitions while retaining the properties of the HPDA.
- Anomaly recovery mechanism: A new anomaly recovery mechanism is developed that prevents a detection system from generating consecutive false alarms after it encounters an anomaly.
- Analysis of program models based on the assumption of single function entry and exit points: This research provides a dynamic learning algorithm that relies on the assumption of single function entry and exit points. Experimental results demonstrate improvement in the false positive rate with the new algorithm. The extent of the violation of this assumption in real world applications was also investigated.
- Empirical evaluation of operations for methods based on call stack information: This research provides the first reported detailed empirical evaluation of operations that are used by the HPDA and other methods based on supplementing system call sequences with call stack information. The results demonstrate that these operations introduces negligible or low overheads.

Our basic modeling approach can be used anywhere there is a programming interface. Although the implementation reported in this dissertation works with system calls, it is possible to build similar systems that work with library calls, object method invocations, or even microprocessor instructions. Our model is a context sensitive model that is able to capture the function call context and is not vulnerable to the impossible path problem described in section 1.2.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter II gives a survey of related work on methods for intrusion detection and intrusion prevention. Chapter III introduces the proposed models and the static analysis and dynamic learning algorithms for each model. Chapter IV provides the experimental results that demonstrate the detection capability and improvement on false positive rates of our approaches. The overhead of our prototype system is measured and the applicability of our system with real world applications is demonstrated. Finally, Chapter V states the conclusions of this research and presents future work.

CHAPTER II

LITERATURE REVIEW

Intrusion detection has been the focus of a great deal of research during the past twenty years [23,31,34,37,40,42,46,53,66]. Attacks that misuse system resources can be detected by observing anomalies in the system calls issued by a program. This dissertation investigates the development of an approach for detecting deviations from normal sequences of system calls defined in the program executable using a combination of static analysis of the program executable and dynamic learning based on audit data. This approach can be used to detect and prevent attacks such as buffer overflow, format string, or viruses and worms that modify the execution paths defined in the original program executable. Previous research efforts that attempt to mitigate or eliminate this type of attack by learning models of program behavior based on sequences of system calls will be discussed in section 2.1. Static analysis of source or executable code has previously been used for both intrusion detection and for detection of software flaws prior to delivery. Section 2.2 will provide a review of previous work in static analysis of program code. In addition to intrusion detection approaches, researchers have also investigated methods for preemptively preventing misuse of system resources. Methods for prevention that modify the operating system architecture or the execution environment are discussed in section 2.3 and those that use access control-based methods are discussed in section 2.4.

2.1 Intrusion detection via sequences of system calls

A widely studied technique for detecting attacks on programs is monitoring the sequence of system calls invoked by programs. System calls provide the interface between programs and the operating system and are the mechanism that programs use to access system resources. Attacks that misuse system resources modify the normal sequence of system calls invoked by the program. Formally, the traces of system calls are strings S drawn from a language L over an alphabet $\Sigma = \{exit, open, close, read, ...\}$ consisting of possible system calls used by a program. An example sequence of system calls produced over time is presented in Figure 2.1 where (A, B, C, D, E, F) represent symbols in Σ.

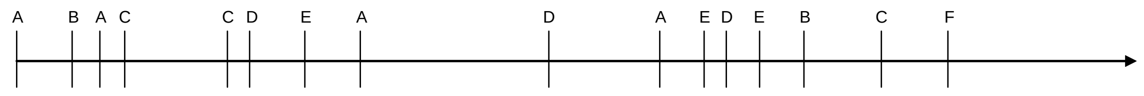


Figure 2.1

A trace of system calls produced over time

An accurate model of program behavior as reflected in system calls can be used for intrusion detection. New sequences of system calls generate by a program that are not accepted by the model will be treated as attacks. Methods for obtaining those models have been developed using either dynamic learning from audit data of system call sequences or static analysis of source code or the binary executable.

2.1.1 Dynamic model learning based on sequences of system calls

Several methods in the machine learning area can be used for building models from sequences of discrete events. These include sliding window methods, recurrent sliding windows, hidden Markov models, conditional random fields, and graph transformer networks [26].

Forrest, Hofmeyr, and Somayaji [34] described a method called sequence time-delay embedding (*stide*), in which a profile of normal behavior is built by enumerating all unique, contiguous sequences of a predetermined, fixed length *n* that occur in the training data using a sliding window method. For example, suppose a normal trace consisting of the following sequence of calls:

execve, brk, open, fstat, mmap, close, open, mmap, munmap

and a window size of 4 is used. A window is moved across the trace and, for each system call encountered, the calls that precede it at different positions within the window are recorded, numbering the calls from 0 to window size - 1, with 0 being the current system call. The trace above yields the following instances:

position 3	*position 2*	*position 1*	*current*
			execve
		execve	*brk*
	execve	*brk*	*open*
execve	*brk*	*open*	*fstat*
brk	*open*	*fstat*	*mmap*
open	*fstat*	*mmap*	*close*
fstat	*mmap*	*close*	*open*
mmap	*close*	*open*	*mmap*
close	*open*	*mmap*	*munmap*

Forrest et al. made use of a tree structure to record every *n-gram* appearing in normal runs of a program. During detection, the new trace is divided into *n-gram* sequences as described above and the *n-gram* sequences are compared with those recorded in the tree to decide if they have been seen before. If a sequence cannot be found in the tree, an alarm is generated. The tree structure speeds the searching for an *n-gram* sequence and reduces the size of the profile. However, the tree structure only records sequences that have appeared in the normal run of a program and does not support any generalization. Since any sequences not recorded in the tree will cause an alarm, a training dataset that does not cover most program behavior will result in a model that may raise many false alarms during the detection phase.

Many researchers have extended Forrest's *n-gram* approach by applying machine learning algorithms to learn the profile from *n-gram* sequences instead of merely enumerating them [39, 53]. Ghosh, Schwartzbard and Schatz have used neural networks to analyze system call log data and demonstrated that neural networks have better generalization on the data than the *stide* approach [39]. Lee and Stolfo applied RIPPER, a rule-based classifier on the classification of *n-gram* sequences [53].

Michael and Ghosh [60] developed a method to learn a non-deterministic automaton (NFA) model from audit data consisting of system call sequences. During training, the audit data was split into sub-sequences of size $n + l$ using a sliding window. For example, with $n = 2$ and $l = 1$, the first two windows created from the sequence a, b, c, d, e would be a, b, c and b, c, d. The first n elements of the window are used to define a state, and the last l elements are used to label a transition coming out of that state. The construction of the automaton proceeds by deciding where the outgoing transitions will lead. This decision is made by referring to the first n audit events in the next window; these states define the next state that the automaton should enter for this particular training sequence. The learned finite automaton represents a language that accepts all the system call traces in the training data. However, since the learning process is based on the *n*-gram scheme, the learned language is only for short sequences with the window size $n + l$ and cannot represent long-term characteristics in the trace. Michael and Ghosh report that their results indicate that the detection performance of this method is worse than that of *stide* [34], although the convergence speed for learning is faster.

All of the above approaches utilize a window size with fixed length which raises a question: what is an appropriate window size? The longer the *n*-gram sequence, the lower the probability that a pattern would match part of an event sequence generated on behalf of an attack. Therefore, longer window sizes result in higher detection ability. However, a long window size will be more process-specific than short ones and will make the problem of the requirement of sufficient training data more serious. Moreover, it is desirable to have a short window size because it reduces the amount of computation need for the detection process and shrinks the size of the behavior profile. Tan and Maxion [73] have presented a framework for analysis of data sets and have shown that a window of size 6 is appropriate to detect intrusions in a dataset from the University of New Mexico [63]. They also demonstrated that characteristics of the attack behavior impact the most appropriate window size. Since attackers are free to change the system call sequences used in their attacks, determining a correct window size a priori remains a difficult if not impossible problem.

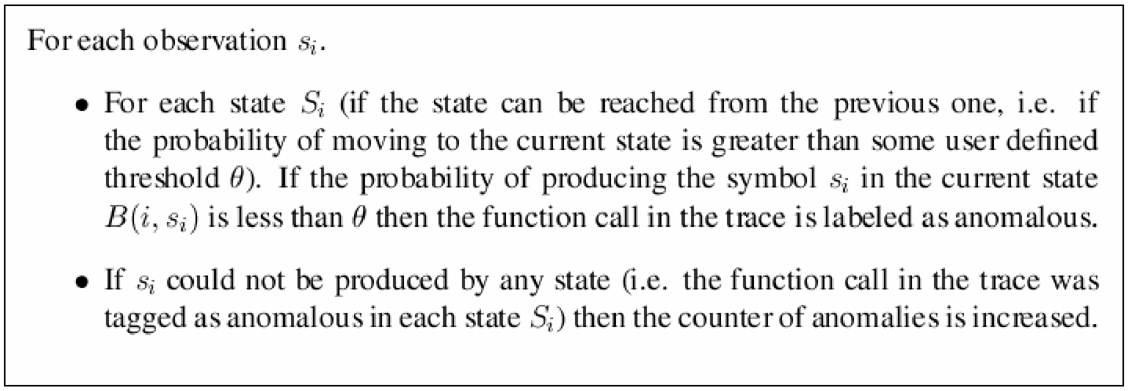
Building a model of program behavior with variable length windows can significantly improve the performance of detection [80]. A sub-function in a program usually invokes the same sequence of system calls and these should not be divided in the behavior model. Variable length sequences are considered a more natural way to model program behavior than fixed length sequences [80].

The *n-gram* based methods can capture the correlations between clustered system calls within the length of sequences in the model. However, none can capture long-distance interactions such as a function that invokes an $open$ at its start and a $close$ at the end. Although Wespi et al.'s method [80] with its variable length sequences can partially capture long-distance interactions, the method used to generate the variable-length patterns will miss long-distance interactions that do not have common intervening system call behavior.

Many researchers have employed HMMs (hidden Markov models) to analyze sequences of system calls and string arguments for web-based applications [50, 51, 79]. These research efforts considered the sequence of events as the output of a probabilistic grammar with probabilities assigned to each of its productions [50]. An HMM can be considered to be an automaton that has probabilities associated with its symbol emissions and its transitions [69]. After learning an HMM, the standard way to classify a new sequence $S(s_1, s_2, ..., s_n)$, is to compute the likelihood of S using the learned HMM model. The output of the Markov model consists of the probability of all paths from its start state to its terminal state. The probability value assigned to any test sequence S is the sum of the probabilities of all distinct paths through the automaton that produce S (as shown in equation 2.1). The probability of a single path is the product of the probabilities of the emitted symbols $ps_i(s_i)$ and the taken transitions $p(t_i)$.

$$p(S) = p(s_1, s_2, ..., s_n) \sum_{(path\ P\ to\ generate\ S)} \prod_{(states\ in\ P)} ps_i(s_i) \times p(t_i) \tag{2.1}$$

Since the probabilities of all possible sequences that are accepted by an HMM sum to 1, even legitimate input that has been regularly seen during the training phase may receive a very small probability value. Therefore, researchers have chosen different schemes to classify the new sequences. Kruegel and Vigna [50] had their model return a probability value of 1 if the word is a valid output from the Markov model and a value of 0 when the value cannot be derived from the given grammar. Warrender etc. [79] used a more complicated method as shown figure Figure 2.2. Florez [33] has proposed an incremental HMM for anomaly detection.



For each observation s_i.

- For each state S_i (if the state can be reached from the previous one, i.e. if the probability of moving to the current state is greater than some user defined threshold θ). If the probability of producing the symbol s_i in the current state $B(i, s_i)$ is less than θ then the function call in the trace is labeled as anomalous.
- If s_i could not be produced by any state (i.e. the function call in the trace was tagged as anomalous in each state S_i) then the counter of anomalies is increased.

Figure 2.2

Detection algorithm using HMM used by Warrender et al. [79]

Researchers have demonstrated that HMMs that consider the entire system call trace result in better detection performance than methods derived from *n*-grams because *n*-grams cannot capture long range dependencies.

2.1.2 Enhancing models based on sequences of system calls

All of the approaches discussed in the previous section consider only system calls and their relative order. That is, during the detection phase, at the invocation point of a system call, only the system call number is taken into consideration. Researchers have shown that the call stack information is a rich source of additional information for detecting intrusions [32, 71].

Sekar et al. [71] first introduced use of the program counter in the call stack accompanying the system call to create a deterministic FSA model. A kernel mechanism (*ptrace*, a kernel interface for debugging and monitoring that is provided by most UNIX-like systems) was used to collect the program counter information associated with each system call. The program counter indicates the location in the program where the operating system request was made.

Upon the inclusion of the program counter, the sequences of events are as follows:

$$\frac{execve}{0xFA10}, \frac{brk}{0xFAB0}, \frac{open}{0xFAB4}, \frac{fstat}{0xFAFB}, \frac{mmap}{0x45FC}, \frac{close}{0xFB10}, \frac{open}{0x03B9}, \frac{mmap}{0x03C0}, \frac{munmap}{0x03CB}$$

Each distinct value of the program counter corresponds to a different state of the FSA. The system calls correspond to transitions in the FSA. Both the current pair of $\frac{SysCall}{PC}$ and the previous pair $\frac{PrevSysCall}{PrevPC}$ are used to construct the transitions. The invocation of the current system call $SysCall$ results in the addition of a transition from the state $PrevPC$ to PC that is labeled with PrevSysCall. The construction process continues through many different runs of the program, with each run possibly adding more states and/or transitions.

Since every “system call” invocation is actually made from the codes within a library function in *libc*, if the program counter value at which the actual system call is invoked is used, the automaton will capture no useful information about the structure of the main program. Instead it models only the structures in library calls. As a result, the automaton that is learned will remain similar across different programs which use the same libraries. Sekar and his colleague provided a solution which goes through the call stack and fetches the program counter that is in the statically linked portion of the executable. This solution performed well with dynamic libraries. However, if the libraries are statically linked or wrapper functions have been introduced for portability, the approach will not work satisfactorily [71]. In those cases, the FSA will learn the location within the library where a system call is made.

Feng et al. [32] extended the approach to create the *VtPath* between two system calls. A call stack that is a collection of function return addresses is extracted when the system is invoked. Figure 2.3 illustrates the organization of activation records in a program stack. The saved *EBP* values connect the activation records and can be used to extract the return addresses. The *VtPath* is the difference between the call stacks for two consecutive system calls that represents the exit and entry of function calls during the execution between the two system calls. During training, two hash tables are used to record the information. One, called the RA (return address) table, is used to save all the return addresses associated with the system calls that have occurred. The other, called the VP (virtual path) table, is used to save all the virtual paths.

The learning algorithm uses a special mechanism for recursive functions. If two different runs of the program have different parameters which result in different depth of the recursion, they will lead to distinct virtual paths. This could make it more difficult for training to achieve convergence. Therefore, when a pair of return addresses are the same, all return addresses between the pair are removed from the virtual stack list, including one of the pair.

Both Sekar [71] and Feng [32] have demonstrated that adding information from the call stack improves detection ability and reduces the convergence time so that less training

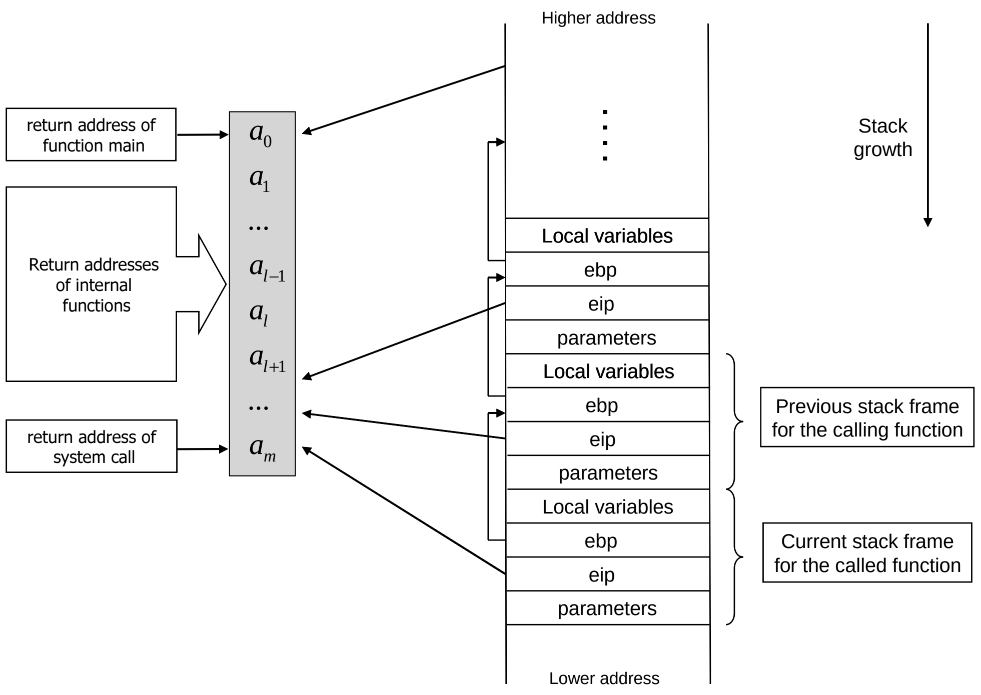

Higher address
return address of function main
Return addresses of internal functions
return address of system call
a_0
a_1
…
a_{l-1}
a_l
a_{l+1}
…
a_m
Stack growth
Local variables
ebp
eip
parameters
Local variables
ebp
eip
parameters
Local variables
ebp
eip
parameters
Previous stack frame for the calling function
Current stack frame for the called function
Lower address

Figure 2.3

Organization of activation records in the program stack data is required to obtain an accurate model than with the *n*-gram methods introduced in section 2.1.1.

2.1.3 Models capturing program control flow

All aforementioned models do not consider the underlying control flow defined in the program code. With the inclusion of additional information from the program stack, the *VtPath* model described in section 2.1.2 is able to partially capture the execution path because the return address from the stack represents the execution flow. However, since the models used in this approach do not directly model the control flow, extra operations must be taken to handle recursive function calls. In this section, several approaches that target capture of control flow are discussed.

2.1.3.1 Static analysis

Wagner and Dean proposed using static analysis of the program source code to develop an automaton model [76]. The control flow graphs implied from source code explicitly define the states and transitions which can be used to create an automaton. In Wagner's research, a non-deterministic finite state automaton (*NDFA*) or push-down automaton (*PDA*) is built by analyzing the source code of the program. The states of the automata implicitly indicate the program counter (PC) of the program and one transition is associated with one system call or move. During detection, whenever a system call is invoked by the program, the *NDFA* moves into a new state associated with the call. If there is not a corresponding transition with this system call from the current state, an anomaly is detected. Due to the limited capability of an *NDFA*, function call context is not captured and the method is susceptible to the impossible path problem [76,78]. Figure 2.4 shows an example *NDFA* with a bold line indicating an impossible path. Wagner and Dean's *PDA* model uses an extra stack to provide information about inter-procedure function calls. The *PDA* approach captures all the call sequences that can potentially appear in a program run and does not allow any sequences that a legitimate program cannot generate. This resolves the false positive problem associated with all approaches that require a learning phase from past audit data.

However, the *PDA* approach is too computationally expensive to be used in practice [41,76]. Giffin, Jha, and Miller have extended this method to learn the model from binary code instead of source code and solve the problem of tracking function call context by executable editing [41]. Their approach inserts two NULL calls (a new system call), one before the invocation of each function call and the other after. The NULL calls provide the function call context for the system. Furthermore, Giffin proposed an enhanced model named the *Dyck* model in which an identifier is passed to the detector through the NULL call [42]. The detector maintains a stack which records the identifier passed in by a NULL call. An entry NULL call will cause the detector to push the identifier onto the stack and an exit NULL call will check for a matching identifier and pop the identifier from the top of stack. This process makes the model more sensitive to function call context.

These *NFSA* and *PDA* models [41,42,76] effectively capture the long distance interactions between system calls defined in the program. Since methods from Wagner and Giffin are based on static learning, the behavior model will not raise any false alarms during de-

```
int one(int x)
{
1             return getuid();
}

int two(int x)
{
2             if(x) getuid();
3             else geteuid();
4             return 1;
}

main()
{
              int i=1;
5             open("foo", O_RDONLY);
6             if(i)
7                           read(0, NULL, 255, 1);
8             read(1, NULL, 255, 1);
9             if(i) two(0);
10            else one(0);
11            close(0);
12            two(1);
13            exit(0);
}
```

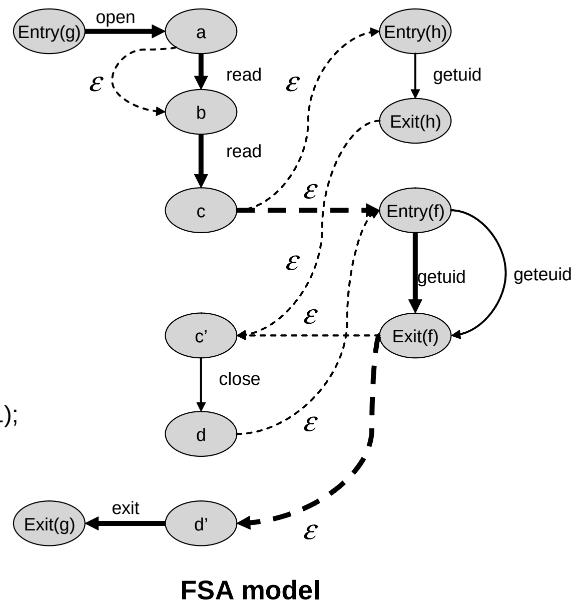


Figure 2.4

Example code and corresponding *NDFA*

tection. By using some approximations to handle unusual program behavior (e.g. signals, indirect calls, *setjmp()* primitives), the authors claim that their models acquired by static analysis of source or binary code can capture most program behavior. However, these approximations reduce model accuracy and thus some attacks can circumvent detection (see section 4.3.2 for detail). These methods also do not handle dynamic linked libraries that are commonly used in modern programming environments.

Recently, Feng, Giffin et al. [31] extended the static learning method to build a PDA model named *VPStatic* with the inclusion of call stack information. Their detector maintains an extra stack like the original *PDA* model proposed by Wagner. However the alphabet of symbols on the extra stack is the same as the symbols that could appear in the system maintained stack. By utilizing the call stack information in the system, they proved that their models, *VPStatic* and *Dyck* (see section 2.1.3), are deterministic and will thus have less computational overhead than the original *PDA* model which is non-deterministic.

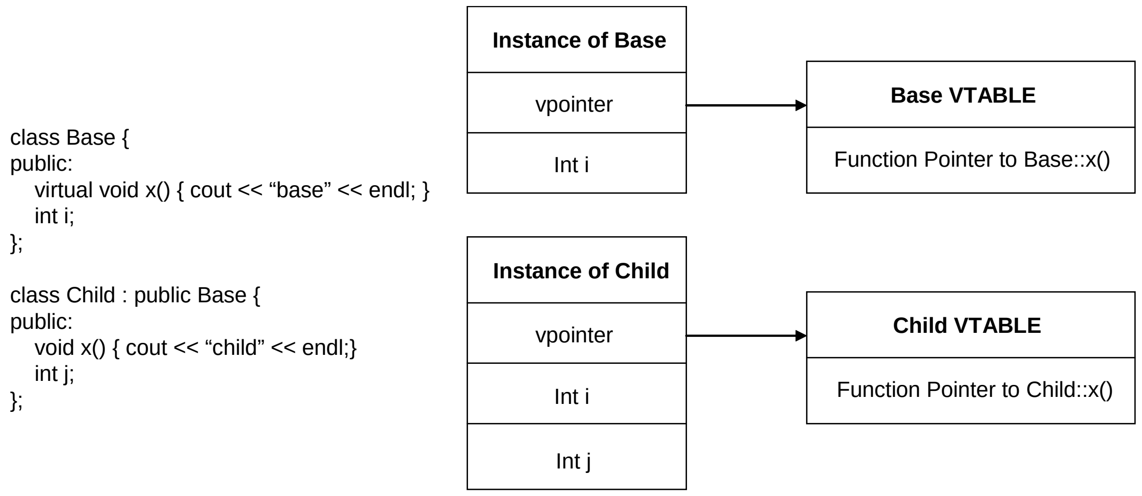
More recently, Gopalakrishna and Spafford et al. [43] proposed a new inlined automaton model (*IAM*). *IAM* uses a representation of program behavior similar to the *NDFA* model [76]. The difference is that *IAM* inlines all function calls in the program and thus the model is context sensitive to the function call flow. However inlining all function calls also results in state explosion. The authors developed methods to compact the state space and data structures to efficiently store the model in memory.

2.1.3.2 Issues in static analysis

Although static learning can capture most program behavior, it is impossible for static learning to acquire behavior which is dynamically determined by the program. Examples include virtual functions in C++, function pointers in C, exception handlers in C++ etc. Exception handlers will cause an abrupt change in the program execution flow. In C, *setjmp()* and *longjmp()* functions are available that can simulate the exception handler in C++. These non-standard execution flows are difficult to handle by static analysis.

Indirect calls — Function pointers have been extensively used in many libraries written in C to emulate an object-oriented programming style or make the implementation flexible to use. One of the essential features in object-oriented programming is polymorphism which is implemented in C++ with virtual functions. The typical compiler implements virtual functions by creating a single table (called *VTABLE*) for each class that contains virtual functions. The compiler places the addresses of the virtual functions for that particular class in the *VTABLE*. Each instance of the class will have a pointer named *vpointer* that points to the *VTABLE* for the class. When a virtual function call is made, the correct function address is fetched through the *vpointer* and *VTABLE*. A graphical representation of this structure is presented in Figure 2.5. The compiler will initialize the value of *vpointer* in the constructor of each class at runtime.

Both function pointers and virtual functions are a type of indirect call. Although sophisticated algorithms for pointer analysis could help to obtain the actual function invoked by the indirect calls, most runtime binding cannot be discovered using static learning.



```
class Base {
public:
    virtual void x() { cout << "base" << endl; }
    int i;
};

class Child : public Base {
public:
    void x() { cout << "child" << endl;}
    int j;
};
```

Figure 2.5

Memory structure for virtual functions

One possible solution used by Wagner is to assume that every pointer could refer to any function whose address has been taken [76]. However, this mechanism adds much non-determinism to the model and the model that is learned is inaccurate and susceptible to attacks such as those described in section 4.3.2.

The *setjmp()* primitive and exception handlers — The *setjmp()* primitive and exception handlers present another difficulty. ANSI C provides a form of non-local control flow that is sometimes used to provide a crude form of exception handling or error recovery. The *setjmp()* primitive saves the stack pointer and other registers and then *longjmp()* may be called by a subroutine to roll the registers and stack back to its saved state. The usage of the *setjmp()* primitive is like a non-local *goto* across functions. Wagner presented a method for handling *setjmp()* and *longjmp()* for the *PDA* model. During the runtime monitoring, the monitor agent maintains a running list of all call stacks that were possible when some *setjmp()* call was visited earlier in this execution trace [76]. Each *longjmp()* call can be emulated by adding this accumulated list to the automaton's current set of states.

Exception handlers in C++ makes the situation more complex. The effect of an exception handler is much like the *setjmp()* primitive except that, before rolling back the registers and stack, the destructors for the objects in current scope will be invoked automatically. Therefore, a static learning algorithm cannot determine how many and which destructors are called when an exception is thrown. If system calls are used within the destructors, it is impossible to predict and record a running list of call stacks as used in Wagner's method.

Signals — Many operating systems allow applications to register a signal handler to be executed upon reception of a signal. The model for the signal portion of the program could occur at any point in the whole model. Therefore, it is difficult to integrate the model for the signal with the model for other parts of program. Interception of runtime signals can potentially be used to determine when a signal has occurred and to switch to the corresponding model for the current signal.

Dynamic libraries — Many general purpose operating systems support dynamic linked libraries. Instead of linking the code in library modules into the binary program image, the library will be dynamically loaded into the process memory space and thus can be shared with different processes which use the same library. This will reduce the size of the program image both in disk and main memory.

There are two issues raised by the use of dynamic libraries. First, function pointers are used extensively to enable the dynamic linking. Second, it may not be possible to know which version of the dynamic library is used in the target system. If a model is created with the wrong version, false alarms can be generated by the model.

Self-modifying code — Self-modifying code is software which achieves its goal by rewriting itself as it executes. It can be used for runtime code generation, or specialization of an algorithm at runtime or load time (which is popular in the domain of real-time graphics). Windows software also uses this method to confuse crackers and to protect copyright. However, this kind of unusual programming pattern imposes a problem for static analysis. Since the code is generated during runtime, static analysis algorithms cannot capture its behavior.

Overlays — Overlays are a solution when a process is larger than the amount of memory allocated to it. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer required. Several programming languages provide mechanisms to help programmers writing overlayed codes. Though this solution is seldom used in current computer systems which usually have plenty of memory, many embedded systems with limited memory still employ this method. Current static analysis algorithms do not handle this case.

2.1.3.3 Dynamic learning from audit data

By only considering the sequences of system calls, it is difficult to learn an automaton that captures control flow because the sequences of system calls do not provide any direct information about internal states of the automaton [71]. For instance, if a sequence of system calls S_0, S_1, S_2, S_0, S_2 is observed, it is impossible to know if two occurrences of S_0 came from the same automaton state or not.

In work with some similarities to ours, Gao et al. [37] proposed a model named *execution graph* that is also extracted from the call stack log. The generation of the *execution graph* is based on dynamic learning and the model conforms to the control flow graph of the program. Gao et al. [37] recognized that the first two different return addresses between two consecutive call stacks can be treated as two call points within the same function. In their paper, they provided a formal definition of the program control flow graph and a proof that the *execution graph* conforms to the portion of the control flow graph exercised during the training executions.

2.1.4 Attacks circumventing behavior-based detection systems

Researchers have presented evidence that behavior-based detection systems can detect attacks such as buffer overflow, format string, and Trojan horse attacks. However, if an attacker is aware of the existence of a behavior-based detection system, a special version of the exploit can be crafted to escape detection [78]. The basic idea of the exploit is that after a successful buffer overflow exploit, the attack is constructed to generate behavior that is accepted as “normal” by the detection system. In most situations, attackers are free to make any number of system calls so that they can identify the most favorable path of execution and then synthetically construct the sequence of system calls that would be executed by this path, inserting the malicious sequence at the appropriate point with the no-op calls between. It has been shown that most system calls can be used as no-op calls. Examples include, open a non-existent file, open a file and then immediately close it, reading 0 bytes from an open file descriptor, or call $getpid()$ and discard the result [78]. Therefore, any system call sequence in a normal program model can be crafted as an attack by faking the arguments for the system call. Wagner and Soto [78] call this type of attack a mimicry attack.

Given a model of the intrusion detection system and a malicious sequence of system calls, a mimicry attack is able to compute a trace of system calls that is accepted by the IDS (without triggering any alarms) and yet contains the malicious sequence. Let Σ denote the set of system calls, and Σ^* the set of sequences over the alphabet Σ. Let $N \subseteq \Sigma^*$ denote the set of system call traces allowed by the IDS, i.e.,

$$N \overset{def}{=} \{T \in \Sigma^* : T \textit{ is accepted by the IDS}\}.$$

Let $M \subseteq \Sigma^*$ denote the set of traces that achieve the attacker’s goals. Then the generation of the mimicry attack is to compute $M \cap N$. The attacker is able to conduct a mimicry attack on a detection system if and only if $M \cap N \neq \oslash$.

Additional information incorporated into the program model can make these attacks more difficult by forcing attackers to emulate not only the system calls but also the incorporated information. For example, in Sekar's model [71] the program counter needs to be emulated and in Feng's model [31, 32] the call stack needs to be crafted to circumvent the detection. However, researchers have determined that it is still possible to develop mimicry attacks that circumvent these detection approaches [38].

2.2 Static analysis of source code for discovering programming bugs

Ideally, software is bug-free before delivery. However, this is rarely the case. Several projects have been undertaken that have the goal of discovering bugs by static analysis of the program source code. Wagner et al. [77] first introduced a method to discover buffer overrun vulnerabilities in C programs. The method formulates detection of buffer overruns as an integer range analysis problem and checks if the inferred allocated buffer size is at least as large as the inferred maximum buffer length. Both the information of inferred allocated size and maximum size are collected by analysis of the program source code. Larochelle and Evans [52] used constraints on buffer operations similar to those employed by Wagner. However, their methods exploit semantic comments added to source code to enable local checking of inter-procedural properties. Their methods also handle non-string buffers that are ignored by Wagner's approach. Recently Dor et al. [27] and Ganapathy et al. [36] have developed methods that are context and control flow sensitive and thus are able to detect more buffer overrun vulnerabilities than the previous two.

Static analysis has also been used to detect other vulnerabilities. Shankar et al. [72] developed a constraint-based type-inference engine that can detect format string vulnerabilities in a program during compile-time. A token-based scanning of source code is employed by Viega et al. [74] to identify many kinds of attacks such as race condition and buffer overrun that are defined in a vulnerability database. Recently, model checking has been proven to be an effective approach for discovery of violations of temporal safety properties [5, 6] and security properties [17, 18]

Static analysis methods significantly reduce the workload of manual code auditing and thus are able to detect many unknown attacks that exist in the software and are ignored by the programmers and the testers. However, these methods, with the exception of model checking based methods, are susceptible to false positives. These false alarms must be further analyzed to identify the real vulnerabilities. Moreover, although static analysis of the source code is able to discover many vulnerabilities, these approaches are not sound (i.e., they are not guaranteed to find out all the possible vulnerabilities) [17]. Even if the model checking approach itself is sound, approximations in the process used to extract a PDA model from source code can result in attacks that are not detected [17].

2.3 Guarding against exploits based on memory errors

Memory overflow will lead to several kinds of exploits such as stack smashes [64], heap overflows [19], printf format vulnerabilities [20], and other errors [3, 25].

Solar Designer wrote the first non-executable stack patch for Linux [24]. Although all stack-based buffer overflow attacks [64] cannot be successfully executed with this patch, other kinds of buffer overflow exploits can evade the protection of this patch [82]. The

Pax team further developed a patch that makes the heap non-executable [65]. Libsafe provides a safe wrapper for unsafe C library calls that usually cause buffer overflow [7]. Other approaches extend the compiler to add protection to the stack [22], pointers [21], and format strings [20] to prevent the attack code from executing.

Forrest et al. [35] first presented the idea of diversity in the operating system or execution environment to make the successful exploitation of a vulnerability difficult or even impossible. They provided a compiler-based prototype to randomize the amount of memory allocated on a stack frame to prevent simple buffer overflow attacks. More recently, researchers have also proposed methods for instruction set randomization [8, 45] and process memory space randomization [12, 28, 65, 86]. Randomization thwarts the attackers from implementing useful exploitations even if the program has buffer overflow vulnerabilities.

2.4 Hypervisor and access control for applications

It is impossible to guarantee that a system is free of attacks. Therefore, a mechanism is needed that will prevent attacks from having a malicious effect on the system. Mitchem et al. [61] first developed an architecture named hypervisors to conduct access control at the system call level. Other similar systems have been designed and developed to enforce policies at the system call level [11, 16]. All of these systems are based on the loadable kernel module (LKM) in Unix-like systems. However, the ability to intercept the system call interface using an LKM are disabled in some modern operating system kernels such as the current Linux kernel. Other researchers have investigated building the access control points into the kernel so that fine-grained access control can be implemented [2, 67, 84]. The Linux Security Module (LSM) is a recent project that defines an infrastructure to implement very fine-grained access control within the Linux kernel [83]. LSM identifies a number of points within the kernel code where resources are accessed and defines relevant callouts from these points so that other modules are able to implement the real access control using these calls. Currently SeLinux is one module that implements a mandatory access control policy using the infrastructure of LSM [2].

Unlike access control that prevents malicious operations, program isolation delays the operations and lets users to decide if the operations are allowed [54]. All modification operations of an untrusted process are saved in a "modification cache" and then users are given an opportunity to examine the modification operation list and decide if the operations are allowed to take effect on the system. Liang et al. [54] presented a system-call interposition based approach for program isolation.

In recent research, Xu et al. have proposed a mechanism called *Waypoint* to control the access of system calls in a function [85]. The set of system calls within each function execution domain is collected by static analysis of the binary code and defined as the *Waypoint*. Binary rewriting has been used to insert a new system call before and after each function call to allow the control system to determine the *Waypoint* of the current execution. When the system call is invoked, the control system checks if the system call is allowed in the current *Waypoint*.

CHAPTER III

CONTEXT SENSITIVE MODELS OF PROGRAM EXECUTION FLOW

In this chapter, we will describe a new automata-based model called the hybrid push down automata (HPDA) that models program execution flow according to system calls invoked by a program and return addresses on the program call stack. The use of the program-maintained call stack gives the HPDA the ability to capture context between function calls. The definition and properties of the HPDA will first be given. Then we describe the use of the HPDA for online intrusion detection. The correspondence between components of the HPDA and program execution flow is discussed. Static analysis and dynamic learning methods for building the models from binary executable and call stack audit logs will be presented. We will explain that both methods effectively develop a model that matches program execution flow. We will then describe an extended HPDA model called the Component-Based Hybrid Push Down Automata (CBHPDA) that separates the models for dynamic libraries used in a program. It will be followed by the presentation of learning algorithms for the CBHPDA and the algorithm for run-time combination of CBHPDA models for intrusion detection.

3.1 Hybrid Push Down Automata (HPDA) model

The HPDA model utilizes the return addresses extracted from the program stack (shown in Figure 2.3) along with system call information to build the profile model. The call stack log is a sequence of system calls associated with a snapshot of return addresses at the point of system call invocation. A sample call stack log is presented in Figure 3.1.

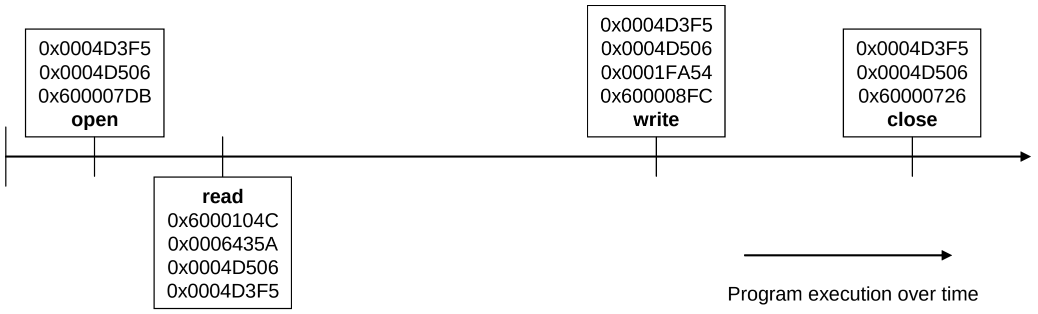


Figure 3.1

A sample call stack log

3.1.1 *HPDA Definition*

Definition 1 *(Hybrid Push Down Automata): An HPDA is a 5-tuple:* (S, Σ, T, s, A) *where* S *is a finite set of states and* Σ *is a finite set of input symbols. In the HPDA model the set of input symbols is defined as* $\Sigma = (Y \times Addr) \cup \epsilon$ *where* $Y = \{Entry, Exit, Syscall\}$ *and* $Addr$ *is the set of all possible addresses defined in the program executable.* T *is the set of transition functions* $(S \times \Sigma \rightarrow S)$, s *is the start state* $(s \in S)$, *and* A *is the set of accept states* $(A = S)$. *The inputs symbols are of four types.*

1. *Entry is a function call entry point. The associated address is the return address for this function.*
2. *Exit is a function call exit point. The associated address is also the return address for this function.*
3. *Syscall is the invocation point of a system call, where the associated address is the return address for this system call.*
4. *ϵ is the empty string.*

For intrusion detection, all states in the learned HPDA model are accept states that indicate the trace is in a normal condition. We can assume there is an explicit state that represents an anomalous status of program execution and any unaccepted transitions lead to this state. Figure 3.2 shows an example HPDA for a sample code.

Although the definition of the HPDA appears to be similar to that of an FSA, it should be noted that the addresses that are incorporated into the input strings come from the system maintained program stack. This gives our model the same capability as Wagner's PDA and thus the name, hybrid push down automata (HPDA). Like the PDA model, the HPDA is also sensitive to function call context. Although the form of our model is similar to the *VPStatic* model described by Feng et al. [31], the HPDA model is based on a different set of assumptions and we have established a set of properties for the HPDA model that support a more robust dynamic learning algorithm allowing us to combine static analysis and dynamic learning. Feng et al. [31] used only static analysis for model acquisition. A new set of algorithms has been developed to acquire an HPDA model by static analysis of the program binary executable that assures that the model will satisfy the properties of

```
int one(int x)
{
1           return getuid();
}

int two(int x)
{
2           if(x) getuid();
3           else geteuid();
4           return 1;
}

main()
{
            int i=1;
5           open("foo", O_RDONLY);
6           if(i)
7                       read(0, NULL, 255, 1);
8           read(1, NULL, 255, 1);
9           if(i) two(0);
10          else one(0);
11          close(0);
12          two(1);
13          exit(0);
}
```

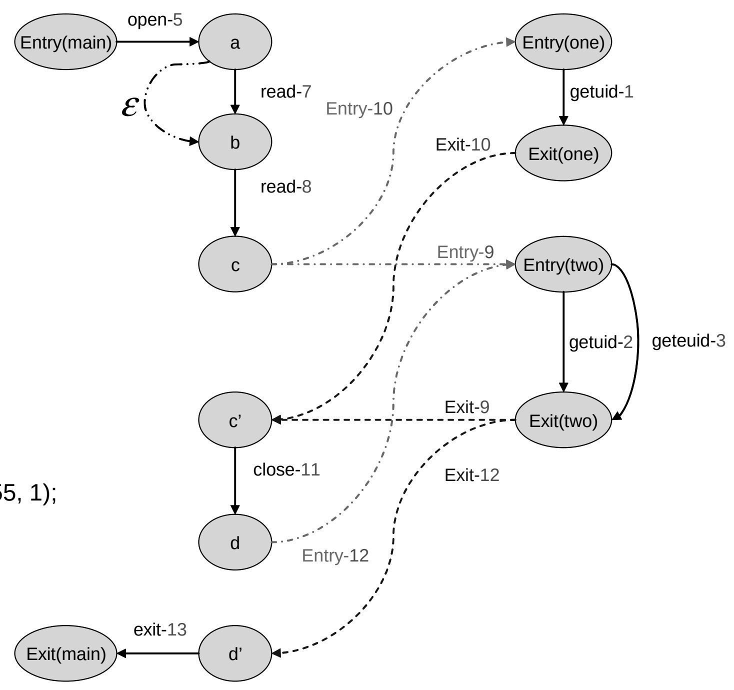


Figure 3.2

A sample C code and its corresponding HPDA model an HPDA model. A dynamic learning algorithm is used to supplement static analysis to model that part of behavior that is difficult to obtain using static analysis.

In a 32-bit operating system, the address of an instruction in program memory space is a 32-bit value that indicates the position within the memory space. A sample address is shown in Figure 3.1. Dynamic libraries have been widely used in modern operating systems. In most of current OS implementations, dynamic libraries can be loaded in arbitrary locations of the program virtual memory space. Therefore after unloading a library and then loading a new library, different instructions may appear at the same address. Moreover, during different runs of a program, the same dynamic library may be loaded at a different start address and thus the code invoking a system call in one dynamic library may be located at a different address from that for other runs. To ensure that our model can handle this dynamic characteristic of system call location, a transformed paired address has been developed for use in the HPDA model.

Definition 2 *(HPDA address): The $Addr$ used in the HPDA model is a pair consisting of the identity of the execution component and the relative address within the component, i.e. exe_id–ra.*

The use of an HPDA address also allows us to separate the models of different execution components of a program in our component based HPDA (for details see section 3.6.2). Notation used to define the properties and in algorithms for the HPDA model are defined below.

Definition 3 *(Notation): The letters $m,\ n,\ and\ o$ will be used for a single symbol in alphabet Σ and the letters $M,\ N,\ and\ O$ will be used for sets of symbols. The letters $x,\ y,\ and\ z$ will be used for a single transition in T and the letters $X,\ Y,\ and\ Z$ will be used for sets of transitions.*

- $type(m)$ $represents\ the\ type\ value\ of\ symbol\ m\ and\ type(m) \in \{Entry, Exit, Syscall, NULL\}$
- $addr(m)\ is\ the\ address\ value\ of\ symbol\ m$
- *Given a transition $x \in T$ where $x = \{s_i, m\} \longrightarrow s_j$*
  - $source(x) \overset{def}{=} s_i$
  - $dest(x) \overset{def}{=} s_j$
  - $symbol(x) \overset{def}{=} m$
- *Given a set of transitions $X \subseteq T$, $symbols(X)$ contains all the symbols appearing in any transitions in X. $symbols(X) \overset{def}{=} \{m : \forall x \in X, symbol(x) = m\}$*
- *An alternative notation for a transition $\{s_i, m\} \longrightarrow s_j$ is $s_i \xrightarrow{m} s_j$*
- *$s_i \xrightarrow{\epsilon^*} s_j$ indicates state s_i can reach state s_j through consecutive epsilon transitions. $s_i \xrightarrow{\epsilon^*} s_j$ holds if and only if $\exists s_k, s_l, ...s_n, transitions\ s_i \xrightarrow{\epsilon} s_k, s_k \xrightarrow{\epsilon} s_l, ...s_n \xrightarrow{\epsilon} s_j \subset T$*
- *$\widehat{s}$ represents the set of states that can be reached from state s through consecutive epsilon transitions. $\widehat{s} \overset{def}{=} \{\widehat{s} \subset S : \forall s_i \in \widehat{s} \Rightarrow s \xrightarrow{\epsilon^*} s_i \in T\}$*
- *Given a set of states S', $out_T(S')$ represents the set of transitions with source states that belong to S'. $out_T(S') \overset{def}{=} \{out_T(S') \subseteq T : \forall x \in out_T(S') \Rightarrow \exists s \in S',\ source(x) = s\}$*
- *$s_{current}$ is the current state of the HPDA.*
- *I (input symbol sequence) —- a sequence of input symbols for the HPDA. All symbols are in the format $type - addr$.*
- *$pop(I)$ —- returns the first symbol in the sequence and removes it from the sequence.*

3.1.2 Properties of the HPDA

In most real-world programs, the instructions in a program image are stationary. In other words, the program file image and the in-memory executable image will not be modified. Therefore, we assume that addresses used in an HPDA model and instructions of the corresponding program have a one-to-one relationship. This assumption fixes the execution flow defined by the instruction in an HPDA address and leads to the development of several important properties of the HPDA model.

Properties of the HPDA:

1. *For each Entry-Addr in Σ, there is a corresponding Exit-Addr in Σ with the same address value (each Entry-Addr is paired with an Exit-Addr).* $\forall m \in \Sigma\ (type(m) = Entry) \Rightarrow \exists m_1 \in \Sigma\ (type(m_1) = Exit \wedge addr(m_1) = addr(m))$.
2. *All symbols in Σ have a unique address value except the paired Entry-Addr and Exit-Addr symbols.* $\forall m, m_1 \in \Sigma\ (type(m) = type(m_1) \vee (type(m) = Entry \wedge type(m_1) \neq Exit) \vee (type(m) = Exit \wedge type(m_1) \neq Entry)) \Rightarrow addr(m) \neq addr(m_1)$.
3. *All symbols in Σ can appear once and only once in any transition. This means all symbols in Σ must be associated with one and only one transition. One exception is the ϵ transition.* $\forall m \in (\Sigma - \epsilon) \Rightarrow \exists y \in T(symbol(y) = m) \wedge \neg\exists y, y_1 \in T(symbol(y) = symbol(y_1))$.

In practice, programs employing overlay and self-modifying code (described in section 2.1.3.2) may not satisfy the above assumptions. Since these two techniques may result in different instructions appearing at the same address, property 2 of the HPDA will not hold. However, such techniques are seldom used in modern commodity applications and thus the HPDA approach is applicable to most applications. Moreover, since the detection and learning algorithms are based on property 3, the algorithms can still work with programs that contains overlay and self-modifying code. Property 2 ensures that an HPDA model matches the execution flow defined in the program executable.

3.1.3 Online detection using the HPDA model

The detection system is activated when a program invokes a system call. Using a method similar to that of Feng et al. [32], the difference between the current call stack and the last call stack is used to generate the input symbols for the HPDA. The simulated HPDA uses these symbols to determine if the current system call invocation is allowed. The call stack represents all currently active routines that have been called but have not yet returned to their respective caller. This information is maintained by each running process.

Using a notation similar to [32], we record the last call stack CS_{last}, the current call stack $CS_{current}$ and the current system call number $SyscallID$. For example, suppose the system is activated by an invocation of a system call with $SyscallID$ and

$$CS_{last} = (a_0, a_1, \ldots, a_l, a_{l+1}, \ldots, a_m)$$

$$CS_{current} = (b_0, b_1, \ldots, b_l, b_{l+1}, \ldots, b_n)$$

where $a_0 = b_0, a_1 = b_1, a_l = b_l, a_{l+1} \neq b_{l+1}, a_m$ is the return address of the last system call and b_n is the return address of the current system call. In the actual program that is running, the following actions have occurred in the period between CS_{last} and $CS_{current}$: 1) the process returns from the last system call that has return address a_m, 2) the process returns from functions that have return addresses $(a_{l+1} \ldots a_{m-1})$, 3) the process invokes functions that have return addresses $(b_{l+1} \ldots b_{n-1})$, and 4) the process invokes the system call that has return address b_n. These events are illustrated in Figure 3.3.

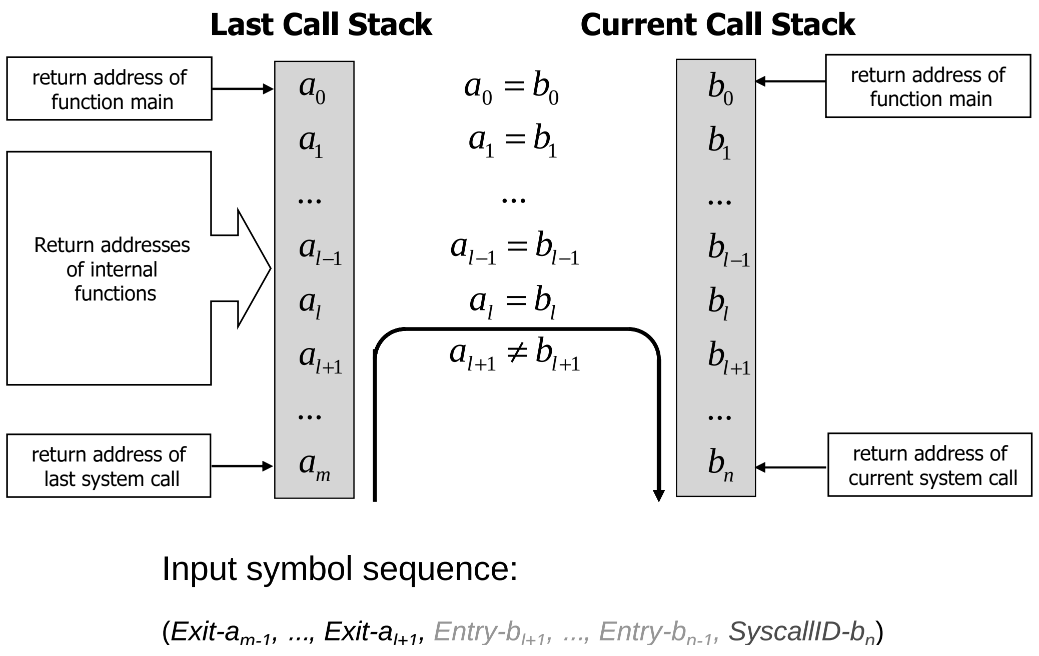

Last Call Stack
Current Call Stack
return address of function main
Return addresses of internal functions
return address of last system call
return address of function main
return address of current system call
a_0 a_1 … a_{l-1} a_l a_{l+1} … a_m
$a_0 = b_0$
$a_1 = b_1$
…
$a_{l-1} = b_{l-1}$
$a_l = b_l$
$a_{l+1} \neq b_{l+1}$
b_0 b_1 … b_{l-1} b_l b_{l+1} … b_n
Input symbol sequence:
(Exit-a_{m-1}, ..., Exit-a_{l+1}, Entry-b_{l+1}, ..., Entry-b_{n-1}, SyscallID-b_n)

Figure 3.3

Input symbol extraction (adapted from [32])

The input symbols for the HPDA are generated from the difference between CS_{last} and $CS_{current}$. In this example, they are:

$$(\textit{Exit-a}_{m-1}, \ldots, \textit{Exit-a}_{l+1}, \textit{Entry-b}_{l+1}, \ldots, \textit{Entry-b}_{n-1}, \textit{SyscallID-b}_n)$$

The HPDA is used to determine if there is a valid transition for each input symbol starting from the current state using Algorithm 1.

Algorithm 1 Detection using HPDA

```
▷$I$ is the ordered list of the input symbols generated from the difference between
$CS_{last}$ and $CS_{current}$

1: procedure DETECTION($I$, $s_{current}$)
2:     while $I$ is not null do
3:         $m \leftarrow pop(I)$
4:         $S_1 \leftarrow \widehat{s}_{current}$
5:         $X \leftarrow out_T(S_1)$
6:         $M \leftarrow symbols(X)$
7:         if $m \in M$ then
8:             $x \leftarrow x \in X \wedge symbol(x) = m.$
9:             $s_{current} \leftarrow dest(x)$
10:        else
11:            raise an alarm
12:            ANOMALY_RECOVERY($s_{current}, m$)
13:        end if
14:    end while
15: end procedure

16: procedure ANOMALY_RECOVERY($s_{current}$, $m$)
17:     if $\exists x \in T, symbol(x) = m$ then
18:         $s_{current} \leftarrow dest(x)$
19:     end if
20: end procedure
```

Figure 3.4

Algorithm1: Detection using HPDA

The HPDA can be acquired by dynamic learning or a combination of static analysis and dynamic learning. It is possible for a model acquired by dynamic learning to produce false alarms during detection because all behaviors may not be represented in the training data. A false alarm generated by the HPDA may lead to a continuous sequence of false alarms. For example, in Figure 3.2, suppose the ϵ transition between state a and b is not learned. When the HPDA is in state a and a system call *read-8* is invoked, a false alarm will be generated and a remains the current state. Therefore, all succeeding normal calls will generate alarms. The following mechanism is used to recover from an anomaly in a graceful fashion. When an anomaly occurs, the program will determine if this symbol is in the HPDA. If there is a transition associated with the symbol (there is at most one according to property 3 of the HPDA), the destination state of this transition will be the new current state. Therefore, subsequent normal symbols will not generate alarms. This process is incorporated in Algorithm 1.

3.1.4 Advantages of the HPDA model

Impossible path problem — Since the HPDA model uses call stack information maintained by the program itself, it is not susceptible to the impossible path problem. For example, consider the impossible path shown in Figure 2.4. Suppose that at state $exit(two)$ with $CS_{last} = (0x9, 0x2)$ an attacker tries to exploit an impossible path and make the execution flow go to d', the $exit$ system call is invoked instead of the expected $close$. The current stack will be $CS_{current} = (0x13)$. This is not allowed by the HPDA. Due to the inclusion of system call stack information, the HPDA model accepts the same set of system call sequences as a PDA model, while the set of sequences allowed by the FSA is a superset of those allowed by the PDA and HPDA.

Mimicry attacks — Mimicry attacks were introduced by Wagner and Soto [78]. This type of attack takes advantage of the characteristics of IDSs that only consider the sequence of system calls. The attacks are conducted by manipulating the sequence of system calls after exploiting a vulnerability to produce a sequence of system calls that is allowed by the IDS. Some of the system call arguments are manipulated to conduct the real work. The arguments can be nullified to allow the system calls to act as no-ops. A detailed description can be found in [78]. The authors state that all system call based IDSs described in [34, 39, 53, 60, 76, 79] are vulnerable to this attack.

Since the HPDA also checks the address values on the stack in addition to the sequence of the calls, it is more resistant than the FSA and PDA models to mimicry attacks. When an attacker conducts a mimicry attack, he must manipulate not only the sequence of system calls but also the exact address at which the system call is used and the call stack context. It is much more difficult for an attacker to create such a complicated attack [38].

Non-determinism — The HPDA removes the non-determinism inherent in the FSA and PDA models and thus reduces the overhead required to simulate the automata. Non-determinism occurs when there are two instances of a system call that can be reached from the current state by a transition. In Figure 2.4, if the current state is a, after the program invokes the system call $read$, neither the FSA nor the PDA can know whether state b or c

should be the next state and thus it is necessary to simulate two copies of the automaton. The same problem occurs when $getuid$ is invoked.

As mentioned in the properties of the HPDA model, all symbols in Σ can appear exactly once in the transitions. Although the HPDA also has ϵ transition, these transitions will not lead to an ambiguity. In the above example, the HPDA model knows which $read$ is invoked by checking its return address. Although the *Dyck* model from Giffin et al. [42] also removes this ambiguity, *Dyck* requires executable editing and creates additional overhead by invoking NULL before and after each function call. Moreover, the *Dyck* model cannot be built dynamically.

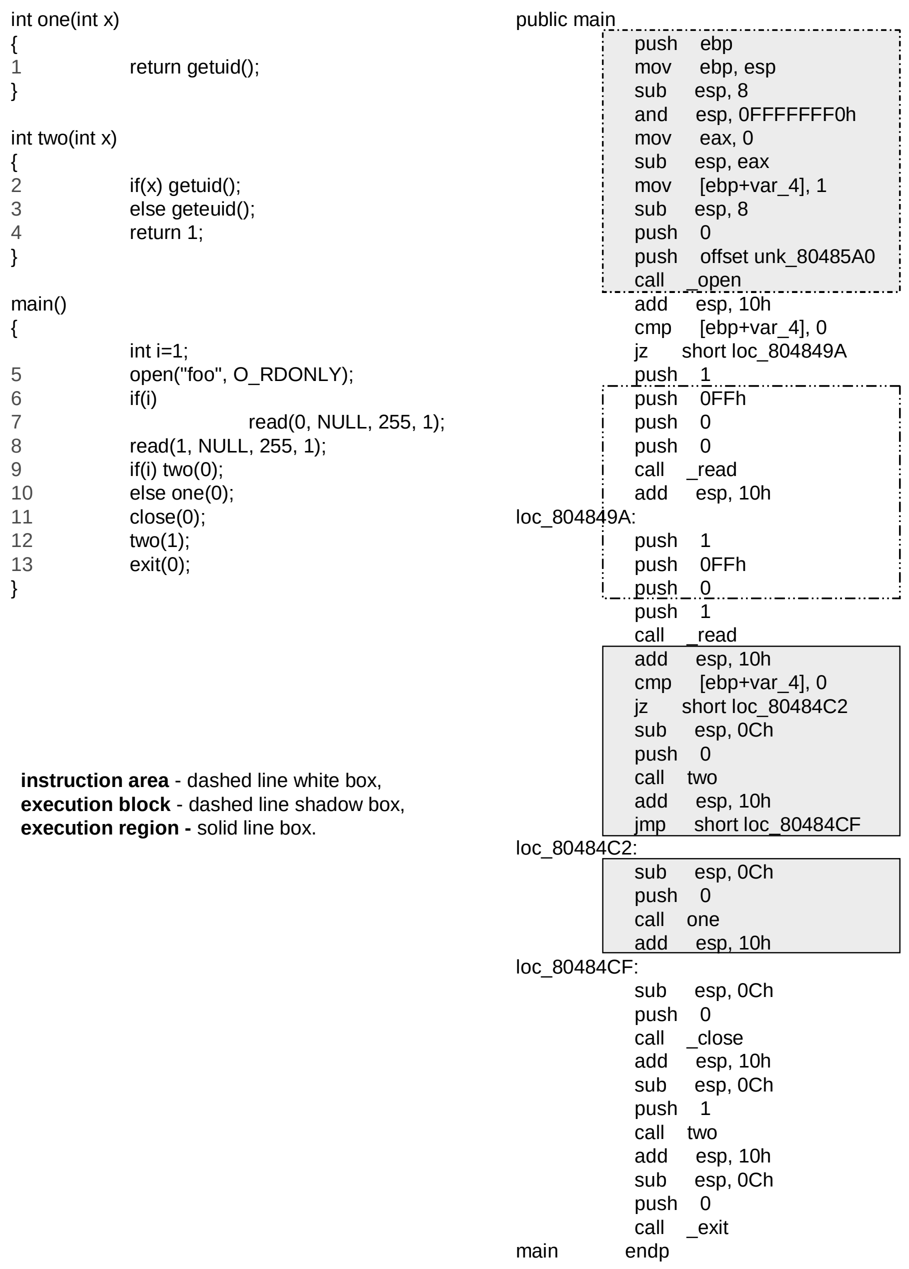
Although it might appear that it would be acceptable to remove all ϵ transitions in the HPDA created by static analysis by creating new transitions from $source(\epsilon)$ to the states that are the destination states of transitions leading out from state $dest(\epsilon)$ [41], this can lead to violation of the properties of the HPDA. For example, if the ϵ transition in Figure 3.2 is removed by adding transition $\{a, read-8\} \rightarrow c$, the address symbol pair associated with the new transition already exists in the HPDA model and addition of the new transition leads to a violation of HPDA properties 2 and 3. If this removal of ϵ transitions is done after static analysis, the dynamic learning algorithm described in a later section cannot be applied to this base model because the algorithm requires that all properties of the HPDA be satisfied. We have, however, developed an algorithm for ϵ reduction (see section 4.8), that can be used to remove a large proportion of the ϵ transitions and the model after ϵ reduction does not violate the HPDA properties. In section 4.8, the experiments demonstrate that the number of ϵ transitions contains only a small percentage of the total transitions, which means the remaining ϵ transitions will not essentially affect the performance. Therefore we do not consider removal of ϵ transitions in our algorithms.

3.2 Instruction execution space

Program execution follows the instructions defined in the program executable. Instructions are executed sequentially unless one of three kinds of instructions is encountered: an unconditional jump (jmp), a conditional jump (jz, jnz, je, etc.), or a function call (call, ret). In this section, we will define the terms *execution block* and *execution region* that divide the instruction space into clusters. The static analysis and dynamic learning algorithms make use of the properties of these clusters.

Two concepts used in the following definitions are *instruction area* and *entry point*. An *instruction area* is an aggregate of instructions and an *entry point* is the place where execution can begin in an *instruction area*. The dashed white box in Figure 3.5 indicates an example of an *instruction area*. The area is composed of a set of instructions and the instructions may or may not be executed sequentially. The *entry points* of this *instruction area* are the instructions "*push 0FFh*" and "*push 1*". A call site is the place where a function call is made using instruction *call*.

Definition 4 *Execution Block: An execution block is an instruction area that has only one entry point and all instructions in this area are executed sequentially from the*



```
int one(int x)
{
1               return getuid();
}

int two(int x)
{
2               if(x) getuid();
3               else geteuid();
4               return 1;
}

main()
{
                int i=1;
5               open("foo", O_RDONLY);
6               if(i)
7                               read(0, NULL, 255, 1);
8               read(1, NULL, 255, 1);
9               if(i) two(0);
10              else one(0);
11              close(0);
12              two(1);
13              exit(0);
}
```

instruction area - dashed line white box,
execution block - dashed line shadow box,
execution region - solid line box.

```
public main
            push    ebp
            mov     ebp, esp
            sub     esp, 8
            and     esp, 0FFFFFFF0h
            mov     eax, 0
            sub     esp, eax
            mov     [ebp+var_4], 1
            sub     esp, 8
            push    0
            push    offset unk_80485A0
            call    _open
            add     esp, 10h
            cmp     [ebp+var_4], 0
            jz      short loc_804849A
            push    1
            push    0FFh
            push    0
            push    0
            call    _read
            add     esp, 10h
loc_804849A:
            push    1
            push    0FFh
            push    0
            push    1
            call    _read
            add     esp, 10h
            cmp     [ebp+var_4], 0
            jz      short loc_80484C2
            sub     esp, 0Ch
            push    0
            call    two
            add     esp, 10h
            jmp     short loc_80484CF
loc_80484C2:
            sub     esp, 0Ch
            push    0
            call    one
            add     esp, 10h
loc_80484CF:
            sub     esp, 0Ch
            push    0
            call    _close
            add     esp, 10h
            sub     esp, 0Ch
            push    1
            call    two
            add     esp, 10h
            sub     esp, 0Ch
            push    0
            call    _exit
main        endp
```

Figure 3.5

Assemblies extracted from IDA Pro for the sample code in Figure 3.2

beginning to the end of the area. An execution block can have at most one call site inside the area.

Definition 5 *Execution Region: An execution region is an instruction area such that every call site in the region must be reachable from every entry point of this region and the region cannot contain two call sites that can be called consecutively.*

Examples of an *execution block* and an *execution region* are shown in Figure 3.5. The two solid line boxes represent one *execution region*. The definition of *execution block* is similar to the definition of *basic blocks* used in the compiler theory for compiler optimizations (a basic block is a straight-line piece of code without any jumps or jump targets in the middle). However, the definition of *execution block* also includes a constraint on call sites.

In the definition of *execution block*, one *entry point* implies that no jumps, jump targets, or call sites exists in the middle of the area. Only the last instruction of the block could be a jump or call site. If the first instruction in an *execution block* is executed, all the instructions belonging to this block are executed. The *execution region* can contain jumps, jump targets, or calls out of its area. Several function call instructions may belong to one *execution region* but only one will be called during each execution of the *execution region*. In other words, during the period from the execution flow into an *execution region* to the execution flow out of this region, at most one call site can be executed. For the example shown in Figure 3.5, the *execution region* contains function calls *one* and *two*, but they cannot be invoked consecutively when the execution starts from the entry point of this *execution region*. The "reachable" property of *execution region* indicates that from every entry point the execution can flow to every call site that the region contains. It should be noted that the *execution region* does not have to contain instructions in continuous memory space and the *execution block* is a special kind of *execution region*.

In the HPDA model, each node implicitly represents an *execution region* and transitions represent the instructions that lead the execution from one region to another. The static analysis and dynamic learning algorithms are based on the concept of the *execution region*. The goal of both methods is to develop an HPDA model that is able to match the execution flow defined in the program executable.

3.3 Static analysis of the executable

A base HPDA model is extracted by analysis of the program executable. Dynamic learning described in next section is used to supplement the base model by acquiring those parts of behavior that are difficult to obtain using static analysis. For the implementation for this dissertation, a commercial disassembler IDA Pro (http://www.datarescue.com/idabase) was used to disassemble the binary executable. A plugin for IDA Pro was developed to extract the statically defined portion of the HPDA model from the assemblies. The IDA Pro Disassembler and Debugger is an interactive, programmable, extensible, multiprocessor disassembler hosted on the Windows platform. IDA Pro supports many binary formats including Windows PE and Unix ELF. The ELF executable in Linux was chosen for analysis in our experiments. The steps used to build an HPDA model using static analysis are shown in Figure 3.6.

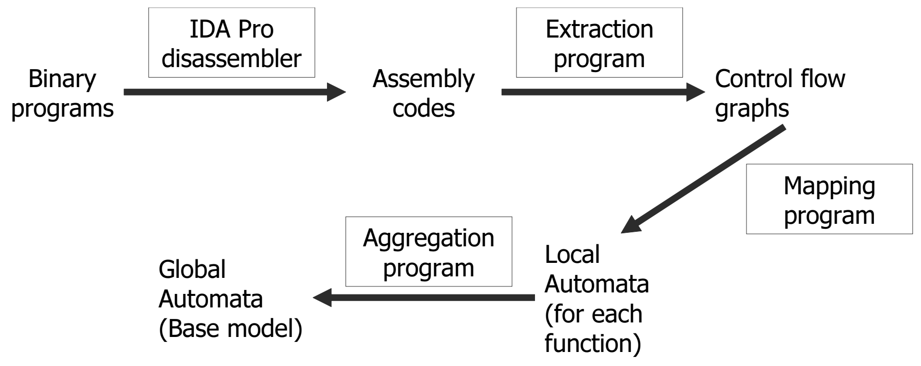


Figure 3.6

Steps for building an HPDA model by static analysis

The static analyzer contains three components:

1. *Extraction program*: Reads the binary executable of the program and constructs a control flow graph for each function in the program. The control flow graph represents all the possible control flows in a function.
2. *Mapping program*: Each control flow graph is converted into a non-deterministic finite state automaton (NFSA) that models all correct system call sequences that the function could produce. The return address for each call site is also extracted and associated with the appropriate non-ϵ transition. The algorithm is designed to reduce extra ϵ transitions.
3. *Aggregation program*: The collections of local automata are composed to form a single global automaton modeling the entire program. Each function call transition points to the correct local automata.

3.3.1 Call graph extraction

Call graphs for each function are extracted from the binary executable using Algorithm 2. Algorithm 2 makes use of a function LEARN_FUNC_FLOW defined in Algorithm 3. A call graph G is composed of blocks and edges connecting them. The notation $G =< B, E >$ represents a call graph, where B is a set of blocks and E is a set of edges. Every call graph has a single entry point called *StartBlock* and a single exit point *ExitBlock* that is the destination of all blocks containing a *ret* instruction. Every block learned in the algorithm is an *execution block.*

The call graph for the sample code in Figure 3.5 is shown in Figure 3.9. The recursive function LEARN_FUNC_FLOW examines every instruction in a function and build the blocks and edges. The function will build a new block whenever a *jump*, *call*, *ret*, or system call instruction is encountered. In the ELF executable for Linux, the instruction "$int\ 80$" is used to invoke a system call and the value of register EAX stores the system call identification number for the current invocation. Our extraction program examines the instructions in reverse order from the system call invocation point and finds the last assignment of the value of EAX. The value assigned to EAX is the system call id. The variable $syscall\_ID$ in Algorithm 3 represents the value obtained by the above process. The extraction program also records the return address of the blocks in which a function call or system call instruction is used. This address information is omitted in Algorithm 3 and Figure 3.9 for simplicity. The two *_read* calls on the edges in Figure 3.9 actually have different addresses and are two distinct symbols for the HPDA model. In IDA Pro, the portion of code that is shared by several functions is called a tail function. We have ignored the analysis of tail functions in our current implementation.

A block that contains no system call or function call is called an empty block. If an empty block has only one in-edge and one out-edge, it can be combined with its preceding block. The function REDUCE_EMPTY_BLOCK is used to reduce the number of empty blocks that have no meaning when building the HPDA. Figure 3.10 illustrates the application of REDUCE_EMPTY_BLOCK that generates block 1 for Figure 3.9. It might also appear that block 2 and 3 shown in Figure 3.9 can be combined. However, the combination will lead to a violation of HPDA property 3 after the mapping from the call graph to the HPDA model described in next section. According to the mapping method, two transitions associated with the same symbol ($\_read - addr$) are created in the HPDA model, but property 3 of the HPDA requires that any symbol can only appear on only one transition in the model. Therefore, all blocks containing a system call or function call have only one out edge in the call graph generated by our extraction program.

Although Giffin et al. and Feng et al. [31, 41, 42] also generate call graphs, no algorithms are mentioned in their papers. From the examples shown in their papers, it is not clear if our algorithm will always result in the same call graphs as theirs.

3.3.2 Local automata conversion

The call graph for each function is then converted to a local HPDA that represents all possible sequences of calls the function can generate. The conversion method is very

Algorithm 2 Call graph extraction

```
    ▷$Blocks \leftarrow EMPTY$
    ▷$Edges \leftarrow EMPTY$

 1: procedure EXTRACT_CALL_GRAPH(function_start_addr)
 2:     $Blocks \leftarrow Blocks \cup StartBlock \cup ExitBlock$
 3:     LEARN_FUNC_FLOW($StartBlock, function\_start\_addr$)
 4:     REDUCE_EMPTY_BLOCK()
 5: end procedure

 6: procedure REDUCE_EMPTY_BLOCK
 7:     for all $block \in Blocks$ do
 8:         if $(block.contain\_call\_site \neq true) \wedge (num\_in\_edges(block) = 1) \wedge (parent(block).contain\_call\_site = true) \wedge (num\_out\_edges(block) = 1)$ then
 9:             $Blocks \leftarrow Blocks - block$
10:             $Edges \leftarrow Edges - in\_edges(block)$
11:             $Update\ the\ source\ of\ the\ out\_edges(block)\ to\ parent(block)$
12:         end if
13:     end for
14: end procedure
```

Figure 3.7

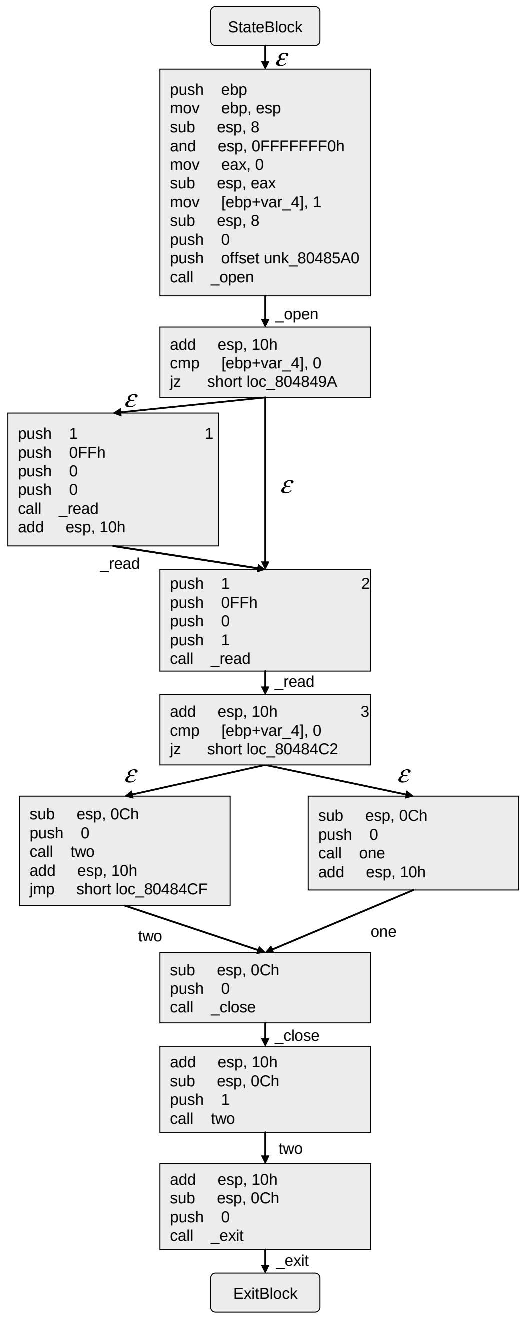
Algorithm 2: Call graph extraction

Algorithm 3 Learn function flow for call graph extraction in Algorithm 2

```
1: procedure LEARN_FUNC_FLOW(pre_block, start_addr)
2:     $block \leftarrow block \in Blocks \wedge start\_addr = block.start\_addr$
3:     if $exists\ block$ then
4:         $Edges \leftarrow Edges \cup \{pre\_block, block\}$
5:     else
6:         $block \leftarrow new\ block$
7:         $block.start\_addr \leftarrow start\_addr,\ current\_addr \leftarrow start\_addr$
8:         $Edges \leftarrow Edges \cup \{pre\_block, block\}$
9:         loop
10:            $inst \leftarrow current\_instruction(current\_addr)$
11:            if $inst\ is\ an\ unconditional\ jump$ then
12:                LEARN_FUNC_FLOW($block,\ destination\ of\ the\ jump$)
13:                $return$
14:            else if $inst\ is\ a\ conditional\ jump$ then
15:                LEARN_FUNC_FLOW($block,\ destination\ of\ the\ jump$)
16:                LEARN_FUNC_FLOW($block,\ next\_addr(current\_addr)$)
17:                $return$
18:            else if $inst\ is\ a\ call\ command$ then
19:                $block.contain\_call\_site \leftarrow true$
20:                LEARN_FUNC_FLOW($block,\ next\_addr(current\_addr)$)
21:                $return$
22:            else if $inst\ is\ a\ syscall\ command$ then
23:                $block.contain\_syscall\_site \leftarrow true$
24:                $block.syscall\_ID = syscall\_ID$
25:                LEARN_FUNC_FLOW($block,\ next\_addr(current\_addr)$)
26:                $return$
27:            else if $inst\ is\ a\ ret\ command$ then
28:                $Transitions \leftarrow Transitions \cup \{block, ExitBlock\}$
29:                $return$
30:            else if $there\ is\ jump\ or\ call\ refer\ to\ the\ next\_addr$ then
31:                LEARN_FUNC_FLOW($block,\ next\_addr(current\_addr)$)
32:                $return$
33:            end if
34:            $current\_addr \leftarrow next\_addr(current\_addr)$
35:        end loop
36:    end if
37: end procedure
```

Figure 3.8

Algorithm 3: Learn function flow for call graph extraction in Algorithm 2



```
push    ebp
mov     ebp, esp
sub     esp, 8
and     esp, 0FFFFFFF0h
mov     eax, 0
sub     esp, eax
mov     [ebp+var_4], 1
sub     esp, 8
push    0
push    offset unk_80485A0
call    _open
```

```
add     esp, 10h
cmp     [ebp+var_4], 0
jz      short loc_804849A
```

```
push    1
push    0FFh
push    0
push    0
call    _read
add     esp, 10h
```

```
push    1
push    0FFh
push    0
push    1
call    _read
```

```
add     esp, 10h
cmp     [ebp+var_4], 0
jz      short loc_80484C2
```

```
sub     esp, 0Ch
push    0
call    two
add     esp, 10h
jmp     short loc_80484CF
```

```
sub     esp, 0Ch
push    0
call    one
add     esp, 10h
```

```
sub     esp, 0Ch
push    0
call    _close
```

```
add     esp, 10h
sub     esp, 0Ch
push    1
call    two
```

```
add     esp, 10h
sub     esp, 0Ch
push    0
call    _exit
```

Figure 3.9

Call graph example for the code shown in Figure 3.5

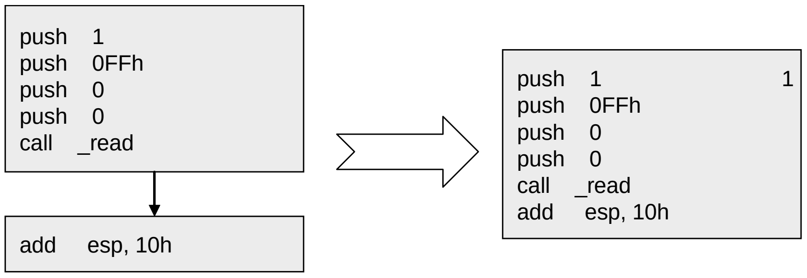


Figure 3.10

Example of operation of REDUCE_EMPTY_BLOCK

similar to the one described by Giffin [41] except that address information is associated with every non-ϵ transition. The method is restated here for completeness.

A call graph $G =< B, E >$ is converted into an HPDA by the following mapping:

$S = B$

$\Sigma = \{m = ID - addr | \exists b \in B, b\ contains\ a\ call\ labeled\ ID\ with\ address\ addr\}$

$s = StartBlock$

$A = S$

$$T = \bigcup_{s \to t \in E} \begin{cases} s \xrightarrow{\epsilon} t & \text{if no call at } s \\ s \xrightarrow{ID-addr} t & \text{if } s \text{ contain } ID \text{ associated with } addr \end{cases}$$

3.3.3 Epsilon reduction

After conversion, every state in the HPDA model implicitly represents an *execution block* in the call graph. The HPDA generated from call graphs follows the HPDA definition and satisfies the HPDA properties. To reduce space requirements and simulation efficiency, each HPDA is ϵ-reduced. Giffin et al. [41] proposed an algorithm that removes

Algorithm 4 ϵ–reduction

```
1:  procedure EPSILON_REDUCTION
2:      for all $s \in S$ do
3:          if $\exists s_i \in S, s \neq s_i \wedge out_T(\hat{s}) = out_T(\hat{s}_i)$ then
4:              $combine\_state(s, s_i)$
5:          end if
6:      end for
7:      for all $x \in T$ do
8:          if $(type(x)\ is\ \epsilon)$ then
9:              if $\forall y \in T, y \neq x \rightarrow dest(y) \neq dest(x)$ then
10:                 $combine\_state(source(x), dest(x))$
11:             end if
12:         end if
13:     end for
14: end procedure

15: procedure COMBINE_STATE(sr, dt)
16:     for all $(x \in T) \wedge (source(x) = dt)$ do
17:         $source(x) = sr$
18:     end for
19:     for all $(x \in T) \wedge (dest(x) = dt)$ do
20:         $dest(x) = sr$
21:     end for
22:     $T \leftarrow T - \cup(x \in T \wedge source(x) = dest(x) = sr \wedge type(x) = \epsilon)$
23:     $S \leftarrow S - dt$
24: end procedure
```

Figure 3.11

Algorithm 4: ϵ–reduction

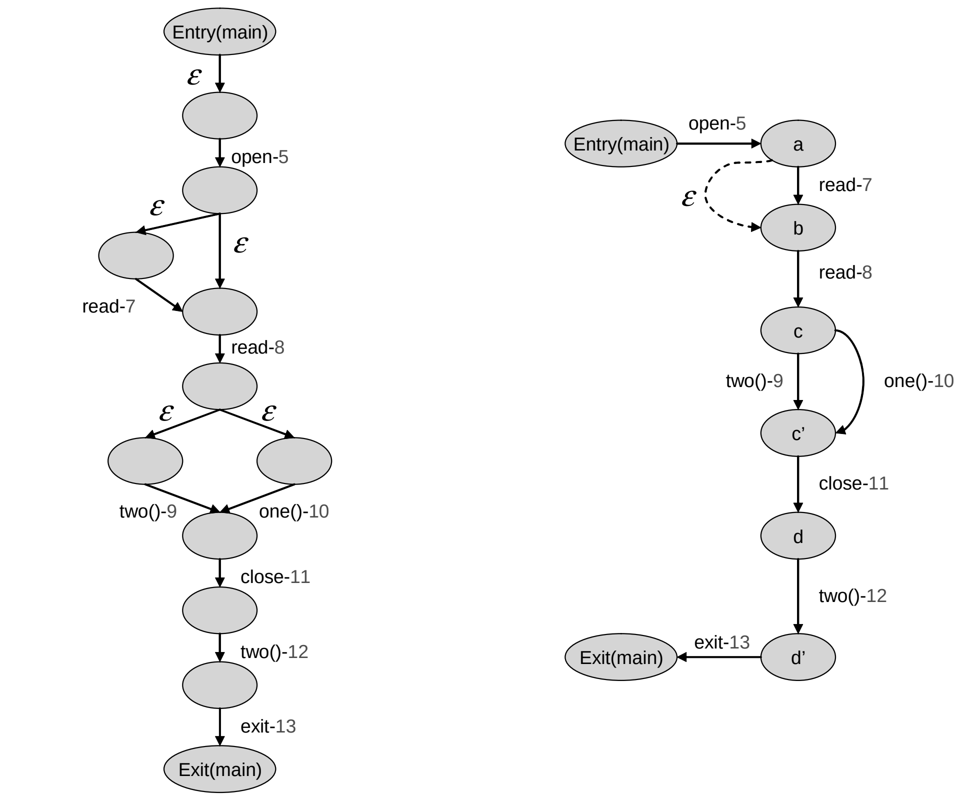
all ϵ transitions. However, their algorithm generates an HPDA that violates property 3 of the HPDA. Therefore, a new ϵ-reduction algorithm is proposed as shown in Algorithm 4.

The basic idea is to maximize the size of each *execution region* by combining the nodes (representing *execution blocks* after conversion) to form a new node representing an execution region. The first *forall* part in the ϵ-reduction algorithm combines all nodes from which the same set of non-ϵ transitions can be reached through zero or several ϵ moves. This process combines several *execution blocks* with different entry points into one *execution region*. The second *forall* enlarges the range of each *execution region* by removing each ϵ-transition where the ϵ-transition's destination state only has this ϵ-transition as its in transition. An example of a local HPDA after conversion and ϵ-reduction for the sample code in Figure 3.2 is shown in Figure 3.12.

3.3.4 Aggregation to global automata

The local HPDAs are composed to form one global HPDA by *call site replacement*. Every transition representing a function call is replaced with the HPDA modeling the callee. This method is also used by Giffin et al. [41, 42] and Wagner and Dean [76]. However, our algorithm differs from theirs, because instead of adding an ϵ transition to the entry node in the callee HPDA, an *entry* transition is added with the address information incorporated in the transition.

The final global HPDA is used for intrusion detection. An example of a global HPDA is shown in Figure 3.14.



(a) Automata converted from call graph (b) Automata after ε reduction

Figure 3.12

Local HPDAs converted from call graphs in Figure 3.9 and after ϵ-reduction

Algorithm 5 Aggregation of global automata

1. Add an $entry - addr$ transition from the source state of the call transition to the entry state of the called HPDA. The $addr$ is the address associated with the call transition.
2. Add an $exit - addr$ transition from the exit state of the called HPDA back to the destination state of the call transition.
3. Remove the original call transition.

Figure 3.13

Algorithm 5: Aggregation of global automata

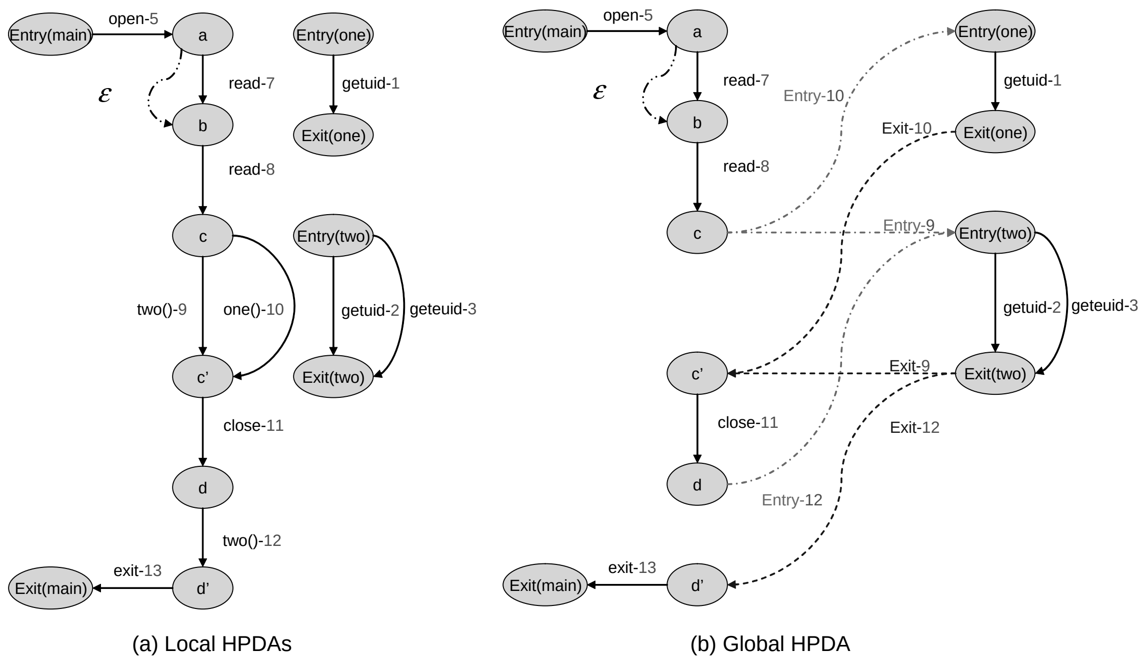


Figure 3.14

Local HPDAs and the corresponding global HPDA after aggregation

3.4 Dynamic learning of execution flow

Because of the existence of indirect jumps and the presence of non-executable data—jump tables, alignment bytes, etc.—in the instruction stream, it is difficult to correctly convert a binary executable to its corresponding disassembly. The popularity of dynamic libraries and position-independent code (PIC) makes the case even worse [70]. In addition, new methods to protect software privacy have been designed to obfuscate the executable code to improve resistance to static disassembly [55]. Static analysis based on the disassemblies also cannot capture many execution flows that may appear during runtime. In this section, a dynamic learning algorithm is proposed for acquisition of an HPDA model by learning from audit logs that record the program's runtime behavior. The dynamic learning algorithm can be used alone or to supplement a base model learned by static analysis. The dynamically learned HPDA adheres to the definitions described in section 3.1.1.

3.4.1 Basis of dynamic learning algorithm

As previously stated, each state of an HPDA implicitly represents an *execution region*. During static analysis, the *execution blocks* are extracted from the binary code by analyzing the control flow graph. Then ϵ-reduction algorithm combines the *execution blocks* into *execution regions* that are the basic components of the HPDA. The purpose of dynamic learning is to build regions and their transitions that correspond to the regions defined in the program executable. Unlike static analysis that proceeds from *execution blocks* to *execution regions*, dynamic learning algorithm first learns a big region and then divides it into small regions or blocks when new instances arrive. Since the algorithm learns *execution blocks*, the ϵ-reduction algorithm should be applied to the learned HPDA model to improve the efficiency of simulation when the model is used for real-time intrusion detection.

3.4.2 Dynamic learning algorithm

Input symbols are generated using the method described in section 3.1.3. Since all *Entry*, *Exit*, or *Syscall* symbols indicate a potential execution flow, they are not distinguished in the algorithm. Algorithm 6 shows how dynamic learning receives a symbol as input and updates the HPDA model if necessary.

The algorithm must deal with three cases. A diagram illustrating the handling of each case is presented in Figure 3.16. Gray nodes represent current states. Case 1 and case 3 are straightforward and will be discussed first. In case 1, a transition labeled with the input symbol exists from the current state $s_{current}$. Therefore, the transition has already been learned and does not need to be learned again. The only action is to change the current state to the target state of the transition.

In case 3, no transition with the current input symbol exists in the model, so a new state is added and a new transition for the current symbol is created from the current state to the new state. The current state is then changed to the new state.

Case 2 deals with situations where a transition x labeled with symbol m exists in the model but not from the current state. Property 3 of the HPDA forbids transitions with the same symbol, so a new transition with the same symbol should not be added.

Algorithm 6 Dynamic learning of HPDA

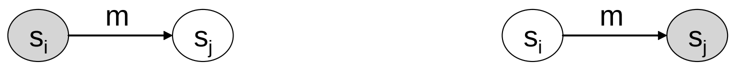
```
1: procedure DYNAMIC_LEARNING($s_{current}$, $m$)
2:     $X \leftarrow out_T(\widehat{s}_{current})$
3:     $M \leftarrow symbols(X)$
4:     if $m \in M$ then                                          ▷ Case 1
5:         $x \leftarrow x \in X \wedge symbol(x) = m.$
6:         $s_{current} \leftarrow dest(x)$
7:     else
8:         if $\exists x \in T \wedge symbol(x) = m$ then                    ▷ Case 2
9:             if $\exists y \in T \wedge dest(y) = source(x) \wedge type(y) \neq \epsilon$ then    ▷ Case 2.1
10:                $s_{new} \leftarrow new(state)$
11:                $s_{source} \leftarrow source(x)$
12:                $T \leftarrow T \cup \{s_{source}, s_{new}, \epsilon\}$
13:                $T \leftarrow T \cup \{s_{current}, s_{new}, \epsilon\}$
14:                $source(x) \leftarrow s_{new}$
15:            else                                               ▷ Case 2.2
16:                $T \leftarrow T \cup \{s_{current}, source(x), \epsilon\}$
17:            end if
18:            $s_{current} \leftarrow dest(x)$
19:        else                                                   ▷ Case 3
20:            $s_{new} \leftarrow new(state)$
21:            $T \leftarrow T \cup \{s_{current}, s_{new}, m\}$
22:            $s_{current} \leftarrow s_{new}$
23:        end if
24:    end if
25: end procedure
```

Figure 3.15

Algorithm 6: Dynamic learning of HPDA

Case 1:

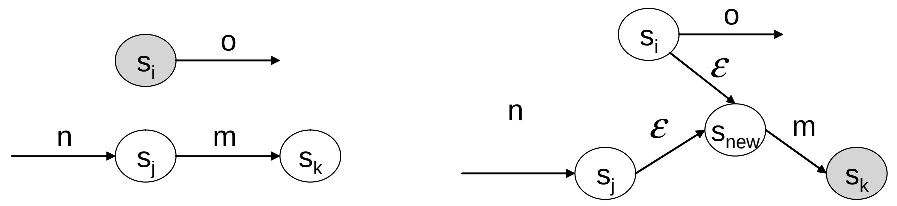
- Current state has the transition labeled with current symbol *m*



Case 2:

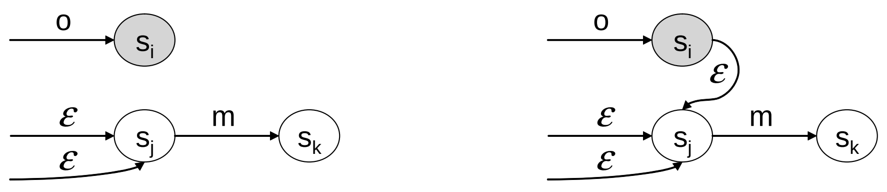
2.1:

- A transition labeled with current symbol *m* exists in the model.
- s_j has at least one non-ε in-transition



2.2:

- A transition labeled with current symbol *m* exists in the model.
- s_j does not have non-ε in-transition



Case 3:

- No transition labeled with current symbol *m* exists in the model

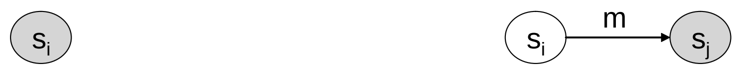


Figure 3.16

The dynamic learning processes (gray nodes represent current states)

It might seem intuitive to simply add an ϵ transition from the current state to the source state of transition x so that the execution flow is able to reach the transition x from the current state without consuming an input symbol. However, this simple solution has the potential of introducing an impossible execution path shown in Figure 3.17. When the ϵ-transition is added as illustrated in Figure 3.17, the new automaton allows execution flow from the current state to every out-transition of the source state of the existing transition x. However, the only new flow information that should be introduced is that the execution flow can reach transition x labeled with m from the current state.

A more complex process must be used to ensure that no impossible paths are introduced. There are two subcases that must be dealt with. In case 2.1 a non-ϵ in-transition exists for the source of the transition x. To deal with this case, a new state s_{new} is created. This state represents a minimum execution block that contains instructions for the existing transition x. Two ϵ transitions are added. One connects $s_{current}$ to the new state s_{new} and the other connects the source state of x to the new state s_{new}. The source state of x then is changed to the new state s_{new}. In case 2.2 the in-transitions of the source state for transition x are all ϵ transitions (i.e. the state represents a minimum execution block). Since the execution block is the smallest element in an HPDA model, it does not need to be divided and a new ϵ transition can be added from the current state to the source state of x. This ϵ transition can be treated as a *jmp* instruction that leads execution from the current state to the entry point of the source state of x.

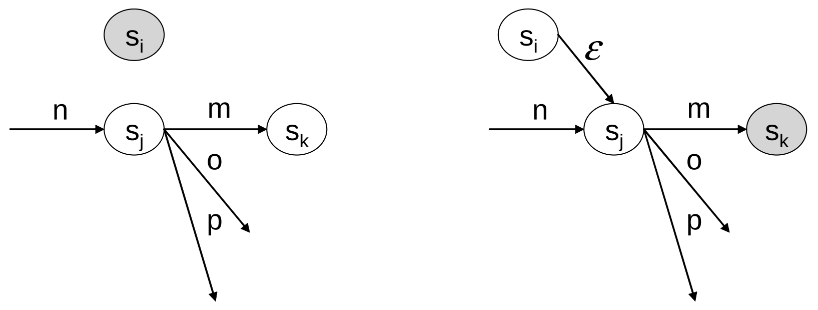


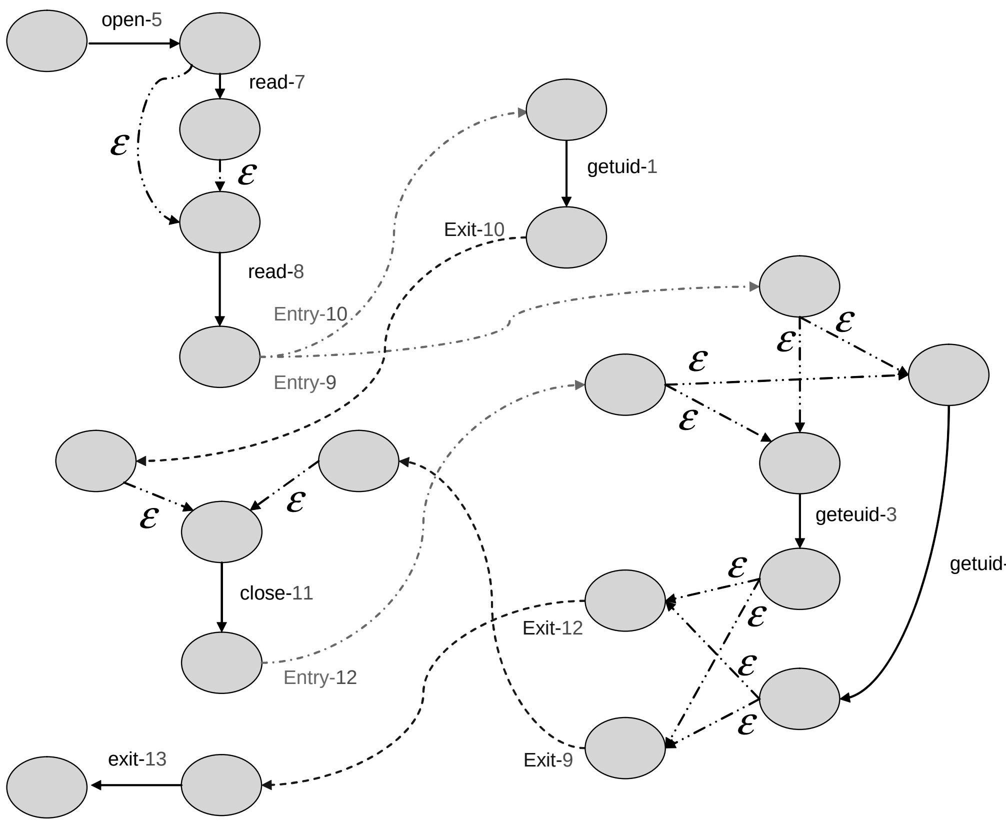
Figure 3.17

Diagram illustrating incorrect handling of case 2

Figure 3.18 shows an example of a dynamically learned HPDA model learned from the training data generated by the example code from Figure 3.2. Since the dynamic learning algorithm divides the *execution regions* into small *execution blocks*, the dynamically learned HPDA model may have more states and ϵ transitions than the final HPDA that is needed for intrusion detection. Applying the ϵ-reduction algorithm on the dynamically learned HPDA will yield the HPDA shown in Figure 3.2.

3.4.3 *Dynamic learning with generalization entry and exit point*

In the previous section a dynamic learning algorithm for the HPDA model was presented. This algorithm learns every execution path that appears in the training data. Further generalization of the execution flow can be achieved if one assumes that every function has a single *Entry* and *Exit* point. For example, in Figure 3.18 the target state of *entry-9* and *entry-12* should be one state even if the epsilon transitions between them are not learned. An algorithm with generalization will require less training data than the one without it.



open-5, read-7, read-8, getuid-1, close-11, getuid-2, exit-13
open-5, read-7, read-8, getuid-1, close-11, geteuid-3, exit-13
open-5, read-7, read-8, getuid-2, close-11, getuid-2, exit-13
open-5, read-7, read-8, geteuid-3, close-11, geteuid-3, exit-13
open-5, read-8, geteuid-3, close-11, geteuid-3, exit-13

Training data

Figure 3.18

Dynamically learned HPDA model given example training data

In this section, an alternate dynamic learning algorithm is presented that is based on the assumption that every function has a single *entry* and *exit* point and each instruction only belongs to one function. Although this assumption holds in most programs, in some optimized programs or programs written in assembly language, a function may jump to an address in the middle of another function. In such cases, the dynamic learning algorithm presented in the previous section more effectively captures the program execution flow. In the remainder dissertation, the algorithm with generalization *entry* and *exit* point is referred using symbol DA_HPDA.

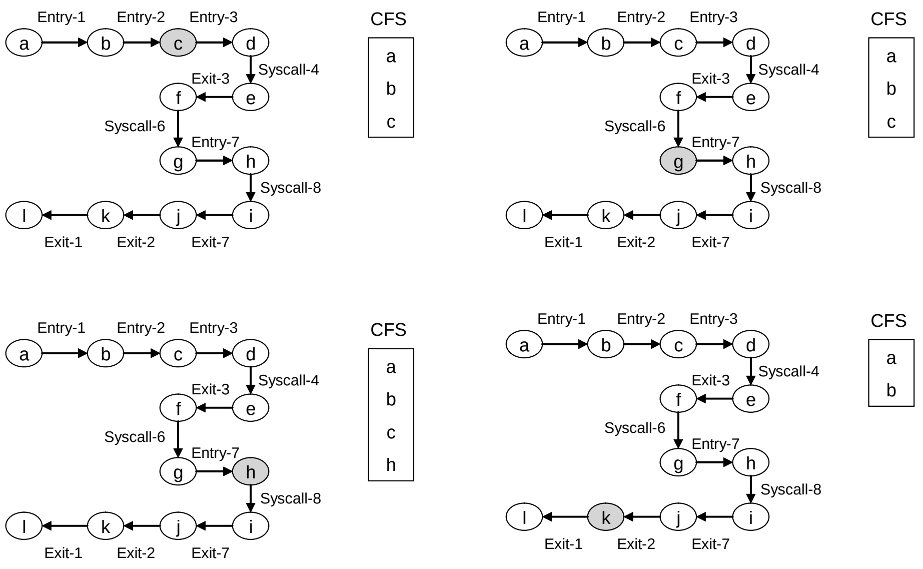


Figure 3.19

Example of CFS status during simulation of the HPDA

Algorithms 7 to 9 give the dynamic learning algorithm based on the assumption that every function has a single *entry* and *exit* point. Two issues that must be answered in the algorithm are: 1) how are functions identified, and 2) how does one determine which function a call site (indicated by a return address) belongs to. In our algorithm, the function is identified by its entry node. The algorithm maintains a stack of states (*CFS*) that stores the entry states of functions that have not returned. This stack is used to track which function the execution is currently within. Function MANIPULATE_FUNC_STACK in Algorithm 8 changes the status of *CFS* according to the input symbols. Examples shown in Figure 3.19 demonstrate how the function MANIPULATE_FUNC_STACK works. Gray node represents current state. Every state belongs to the function identified by the state at the top of current *CFS*.

Each state in an HPDA model is an entry or exit state of a function or an internal state within a function. For each state s, the algorithm stores a link to track the entry state (stored in cf_entry) of the function that s belongs to. Variable $true\_entry$ and com_entry are also used to track the entry state of a function. Since after combination of two entry states m and n (suppose n is removed from the HPDA model), the states that have a cf_entry pointing to n should now have a cf_entry value that points to m. To simplify the process of changing these values, we maintain two extra variables $true\_entry$ and com_entry. Now, when two states such as m and n are combined, instead of going through all states to find those with a cf_entry of n, the value of $true\_entry$ of n points to m. Every time the entry state of any state s is needed, the variable $true\_entry$ is used. This is the value

Algorithm 7 Dynamic learning of HPDA with the assumption that every function has a single *Entry* and *Exit* point

```
$CFS \leftarrow empty$ ▷
$CFS$ stores all function entry states that have not returned
▷define $entry(s) = s.cf\_entry.true\_entry$ that returns the entry state of the function that $s$ belongs to

1: procedure DYNAMIC_LEARNING_2($s_{current}$, $m$)
2:     $X \leftarrow out_T(\hat{s}_{current})$
3:     $M \leftarrow symbols(X)$
4:     if $m \in M$ then
5:         $x \leftarrow x \in X \wedge symbol(x) = m.$
6:         $s_{current} \leftarrow dest(x)$
7:         MANIPULATE_FUNC_STACK($s_{current}, m$)
8:     else
9:         if $\exists x \in T \wedge symbol(x) = m$ then
10:            ADD_EPSILON($s_{current}, m, x$)
11:        else
12:            ADD_NEW_STATE($s_{current}, m$)
13:        end if
14:    end if
15: end procedure

16: procedure ADD_NEW_STATE($s_{current}$, $m$)
17:    $s_1 \leftarrow new(state)$
18:    $T \leftarrow T \cup \{s_{current}, s_1, m\}$
19:    if $type(m) = Exit$ then
20:        if $top(CFS).exit\_state = NULL$ then
21:            $top(CFS).exit\_state = s_{current}$
22:        else if $top(CFS).exit\_state \neq s_{current}$ then
23:            COMBINE_STATES($top(CFS).exit\_state, s_{current}, Exit$)
24:        end if
25:    end if
26:    $s_{current} \leftarrow s_1$
27:    MANIPULATE_FUNC_STACK($s_{current}, m$)
28:    $s_{current}.cf\_entry \leftarrow top(CFS)$
29:    if $type(m) = Entry$ then
30:        $s_{current}.true\_entry \leftarrow top(CFS)$
31:        $s_{current}.com\_entry \leftarrow top(CFS)$
32:    end if
33: end procedure
```

Figure 3.20

Algorithm 7: Dynamic learning of DAHPDA

Algorithm 8 Function MANIPULATE_FUNC_STACK and ADD_EPSILON for Algorithm 7

```
1: **procedure** MANIPULATE_FUNC_STACK($s_{current}$, $m$)
2:     **if** $type(m) = Entry$ **then**
3:         $push(CFS, s_{current})$
4:     **else if** $type(m) = Exit$ **then**
5:         $pop(CFS)$
6:     **end if**
7: **end procedure**

8: **procedure** ADD_EPSILON($s_{current}$, $m$, $x$)
9:     **if** $entry(s_{current}) = s_{current} \wedge entry(source(x)) = source(x)$ **then**
10:        COMBINE_STATES$(s_{current}, source(x), Entry)$
11:    **else**
12:        **if** $\exists y \in T \wedge dest(y) = source(x) \wedge type(symbol(y)) \neq \epsilon$ **then**
13:            $s_{new} \leftarrow new(state), s_{sr} \leftarrow source(x)$
14:            $s_{new}.cf\_entry \leftarrow s_{sr}.cf\_entry$
15:            $s_{new}.true\_entry \leftarrow s_{sr}.true\_entry, s_{new}.com\_entry \leftarrow s_{sr}.com\_entry$
16:            **if** $type(x) = Exit$ **then**
17:                **for all** $z \in T \wedge source(z) = s_{sr} \wedge type(z) = Exit$ **do**
18:                    $source(z) = s_{new}$
19:                **end for**
20:                $T \leftarrow T \cup \{s_{sr}, s_{new}, \epsilon\}$
21:                $T \leftarrow T \cup \{s_{current}, s_{new}, \epsilon\}$
22:                $entry(s_{new}).exit\_state \leftarrow s_{new}$
23:            **else**
24:                $source(x) \leftarrow s_{new}$
25:                $T \leftarrow T \cup \{s_{sr}, s_{new}, \epsilon\}$
26:                $T \leftarrow T \cup \{s_{current}, s_{new}, \epsilon\}$
27:            **end if**
28:        **else**
29:            $T \leftarrow T \cup \{s_{current}, source(x), \epsilon\}$
30:        **end if**
31:        **if** $entry(s_{current}) \neq entry(source(x))$ **then**
32:            COMBINE_STATES$(entry(s_{current}), entry(source(x)), Entry)$
33:        **end if**
34:    **end if**
35:    $s_{current} \leftarrow dest(x)$
36:    MANIPULATE_FUNC_STACK$(s_{current}, m)$
37: **end procedure**
```

Figure 3.21

Algorithm 8: MANIPULATE_FUNC_STACK and ADD_EPSILON for Algorithm 7

Algorithm 9 Function COMBINE_STATES for Algorithm 7

```
1: procedure COMBINE_STATES($main\_s$, $alt\_s$, $type$)
2:     if $type = Exit \wedge ((\exists y \in T, dest(y) = main\_s \wedge type(y) \neq \epsilon) \vee (\exists y \in T, dest(y) = alt\_s \wedge type(symbol(y)) \neq \epsilon))$ then
3:         $s_{new} \leftarrow new(state)$
4:         for all $y \in T \wedge source(y) = main\_s \wedge type(symbol(y)) = Exit$ do
5:             $source(y) = s_{new}$
6:         end for
7:         for all $y \in T \wedge source(y) = alt\_s \wedge type(symbol(y)) = Exit$ do
8:             $source(y) = s_{new}$
9:         end for
10:        $T \leftarrow T \cup \{main\_s, s_{new}, \epsilon\}$, $T \leftarrow T \cup \{alt\_s, s_{new}, \epsilon\}$
11:        $entry(main\_s).exit\_state \leftarrow s_{new}$
12:    else
13:        for all $x \in T \wedge dest(x) = alt\_s$ do
14:            $dest(x) = main\_s$
15:        end for
16:        for all $x \in T \wedge source(x) = alt\_s$ do
17:            $source(x) = main\_s$
18:        end for
19:        if $type = Exit$ then
20:            $S \leftarrow S - alt\_s$
21:        else
22:            $main\_s.com\_entry \leftarrow alt\_s$
23:            UPDATE_ENTRY($main\_s, alt\_s$)
24:            if $main\_s.exit\_state \neq alt\_s.exit\_state$ then
25:                COMBINE_STATES($main\_s.exit\_state, alt\_s.exit\_state, Exit$)
26:            end if
27:        end if
28:    end if
29: end procedure

30: procedure UPDATE_ENTRY($main\_s, alt\_s$)
31:     $alt\_s.true\_entry \leftarrow main\_s$
32:     $c\_s \leftarrow alt\_s.com\_entry, l\_s \leftarrow alt\_s$
33:     while $c\_s \neq l\_s$ do
34:         $c\_s.true\_entry \leftarrow main\_s$
35:         $l\_s \leftarrow c\_s, c\_s \leftarrow c\_s.com\_entry$
36:     end while
37: end procedure
```

Figure 3.22

Algorithm 9: Function COMBINE_STATES for Algorithm 7

returned by $entry(s)$ in Algorithm 7. If for any state s, the value of $entry(s)$ points to itself, this state is an entry state. For each entry state s, a variable named $exit\_state$ stores the exit state of the function represented by s.

The main function defined in Algorithm 7 is similar to the dynamic learning algorithm described in the previous section. Lines 5-7 handle case 1 described in the previous section, line 10 handles case 2, and line 12 handles case 3.

In the function ADD_NEW_STATE, since the transition labeled with the current symbol m does not exist in the HPDA model, a new transition associated with m is created. If $type(m)$ is $Exit$, the execution is returning from a function and the current state is the exit state for this function. A NULL value of $exit\_state$ for the state at the top of *CFS* indicates that the return of the function has not been previously learned. Therefore, the value of $exit\_state$ points to the current state. If the value of $exit\_state$ is not NULL and does not point to current state, it indicates that the function can return through another state. With the assumption of a single entry and exit point, these two states can be combined.

Function ADD_EPSILON handles the situation where a transition x labeled with the current symbol m exists in the HPDA model, but it is not one of the out-transitions of the current state. If both the source state of the existing transition x and the current state are entry states, according to the single function entry and exit point assumption, these two nodes can be combined. The algorithms for handling cases is similar to those defined in the previous section except when the type of the existing transition x is an $Exit$ transition. In this case the source state of x is an exit state. Since the exit state is unique for any

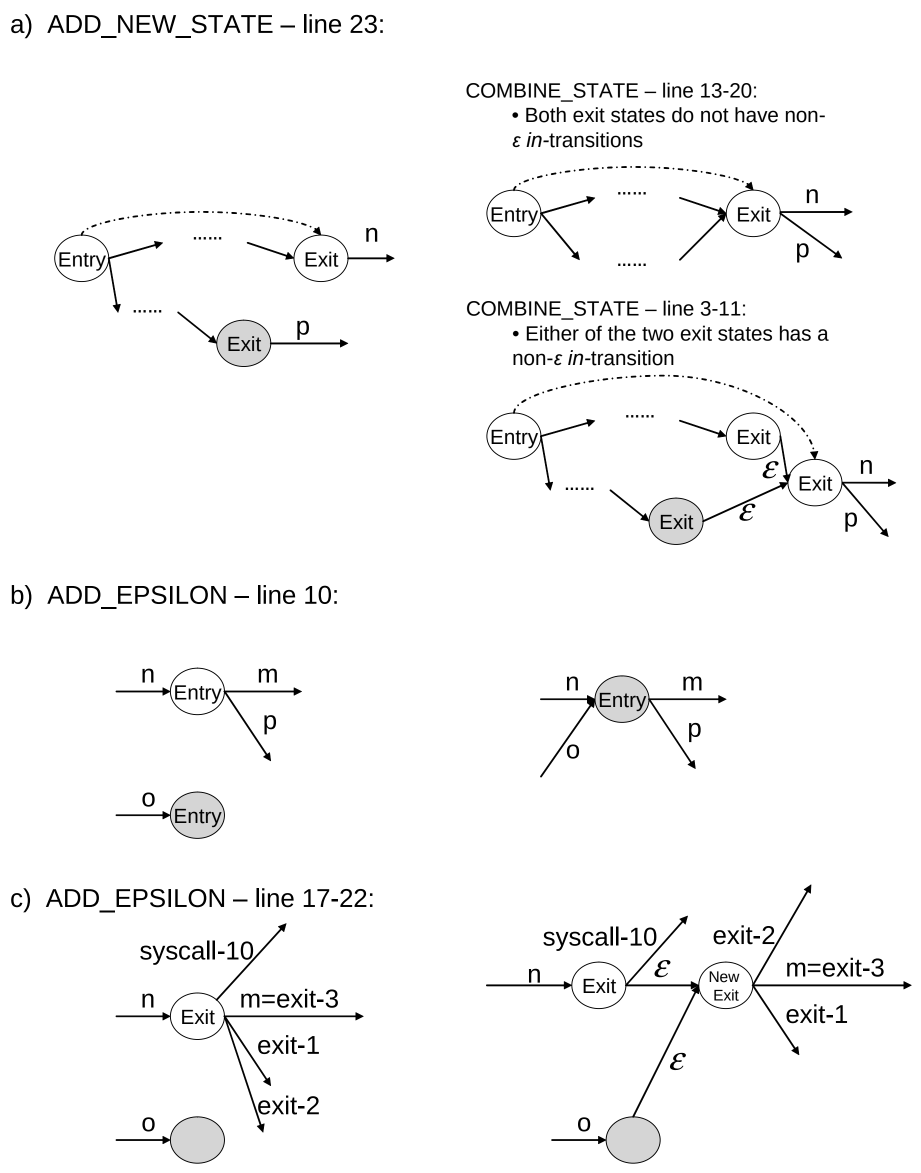

a) ADD_NEW_STATE – line 23:
COMBINE_STATE – line 13-20:
• Both exit states do not have non-ε in-transitions
COMBINE_STATE – line 3-11:
• Either of the two exit states has a non-ε in-transition
b) ADD_EPSILON – line 10:
c) ADD_EPSILON – line 17-22:
Entry
Exit
New Exit
n
p
m
o
ε
......
syscall-10
m=exit-3
exit-1
exit-2

Figure 3.23

Diagram illustrating the cases in the DA_HPDA algorithm function, all $Exit$ transitions should be copied to the new state. An illustration of the actions in line 17–24 are shown in Figure 3.23. Line 33 checks if the current state belongs to the function identified by the state at the top of *CFS*. If not, it indicates the existing transition is currently shared by two functions and the two functions should be combined by combining the two entry states because of the assumption that each transition only belongs to one function.

The function COMBINE_STATES combine two states if possible. If either one of the two exit states has a non-ϵ in-transition, the states should not be combined because doing so may introduce an impossible path. In this case, a new state s_{new} is created as the new exit state and two ϵ transitions are created to connect the two exit states to s_{new}. After combination of the two entry states, the values of $true\_entry$ should be updated as discussed previously. If two entry states have different exit states, the two exit states should also be combined.

3.4.4 Advantages of the HPDA model

Our dynamically learned HPDA model can be treated as an alternative representation of the *VtPath* model proposed in [32]. Both approaches represent the call stack difference between two consecutive invocations of system calls. However, the HPDA representation is more compact than the *VtPath* representation. *VtPaths* are represented in a string form and saved in a hash table. One symbol appearing in several *VtPaths* needs to be saved several times. However, in our HPDA, each symbol only appears in one transition [56].

The HPDA also has an advantage when handling recursive function calls. In the *Vt-Path* model [32], recursion needs to be detected and treated as a special case, increasing the detection overhead. In the PDA model [76], the size of the extra stack for the PDA will increase dramatically when recursion is encountered and thus increase the memory and execution time overhead, especially when non-determinism is encountered. Since the HPDA does not maintain a separate call stack and the structure of the automaton graph can represent recursive functions naturally, no additional overhead is required to deal with a recursive function call.

Since *VtPath* treats the execution path as a string and records the strings in a hash table, it does not generalize the program control flow. For the example shown in Figure 3.24, the model is part of an HPDA that was learned from the two input sequences shown. Using the HPDA model, the two paths (a c d) and (b c e) are accepted. But in the *VtPath* model, the learned input sequences are treated separately and thus the paths (a c d) and (b c e) are not accepted.

The dynamic learning algorithm based on the single *Entry* and *Exit* point assumption can effectively capture the control flow structure and thus exhibits faster convergence during learning. For example, consider the program shown in Figure 3.2. After learning two traces that use *two(1)-12* twice (one called *getuid-2* and the other called *geteuid-3*), the flow structure in function *f()* has been learned. After learning a trace which contains a path *two(0)-9* and *getuid-2* the transitions *Entry-9* and *Exit-9* are learned. From this information, the HPDA model can infer that a path *Entry-12*, *geteuid-3*, *Exit-12* is valid in

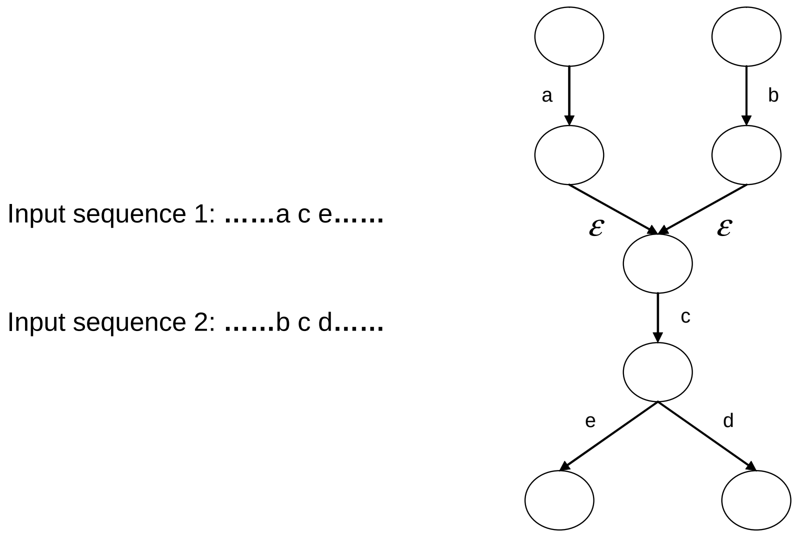


Figure 3.24

Generalization capability of the HPDA model compared to *VtPath*

the program execution. However, *VtPath* will not accept this behavior until it encounters a trace that explicitly contains an execution path of *Entry-12*, *geteuid-5*, *Exit-12*.

3.5 Combination of static analysis and dynamic learning

The HPDA model can be acquired using a combination of static analysis and dynamic learning and experiments (Chapter IV) have shown that the acquired model demonstrates a better tradeoff between detection and false alarm rates than similar models acquired only through static analysis or dynamic learning. A base model is learned by static analysis and is supplemented by dynamic learning. The base model contains most of execution paths that can be obtained by analyzing the program executable statically and dynamic learning supplements the model with the behavior that cannot be learned statically.

Although the dynamic algorithm we have described above can be used alone to learn the HPDA model completely from audit data, we propose combining the dynamic learning method with static analysis to take advantage of the strengths of both. Unlike the work described in [41] and [42], we work with programs that are dynamically linked (that is the reality for real-world applications). In the combination approach, dynamic learning is used to supplement the base model learned by static analysis by acquiring transitions representing legal behavior that cannot be captured by static analysis. Static analysis captures the execution flow in the program executable and dynamic learning captures the flows that cannot be determined without running the program or that are generated by dynamic libraries. Since the type of transitions and structure of the HPDA learned by static analysis and dynamic learning both follow the definition of the HPDA presented in 3.1.1, the two approaches can be seamlessly integrated.

Due to the difficulties mentioned in section 2.1.3.2, the static analysis based approaches described in section 2.1.3.1 cannot capture all normal behavior of a program and approximations are typically used to acquire the behavior that is not represented in the executable. Since our combination approach does not employ any approximations during learning, the resulting model is more exact and is more difficult to evade than a model acquired through static analysis alone. Moreover, because static analysis discovers most execution paths defined in the program executable, the model acquired through the combination approach will result in lower false alarm rates than using a model acquired through dynamic learning alone.

3.6 Component-Based Hybrid Push Down Automata (CBHPDA) model

In this section, an extended HPDA model, the component-based hybrid push down automata (CBHPDA) is described. This model separates the models for dynamic libraries used in a program so that the models can be shared by different programs models. A definition of the model will be given, the method used to generate the models by static analysis and dynamic learning will be presented, and a method for combining the component models at run-time will be described.

3.6.1 Motivation for a component based model

Previous approaches for acquiring models of program behavior from code have mainly focused on statically linked programs. The prototype implementations and experiments do not take dynamic libraries into consideration. However, dynamic libraries are widely used in modern programs such as *Apache*, *Oracle*, *VsFtp*, etc. The *Apache* web server provided in Red Hat 9.0 uses at least 20 dynamically linked libraries as shown in Table 3.1. This list of dependent libraries is generated by the program *ldd* which analyzes program executable headers and discovers a list of libraries used by this program.

The distribution of the number of transitions across different execution components are presented in Table 3.2. The numbers were collected from a model dynamically learned from a dataset that was obtained by running a simple web site with several CGI programs and HTML pages with an *Apache* server. Table 3.2 shows a list of libraries used by *Apache* and the learned HPDA transitions for each component (note: the main executable

Table 3.1

A list of libraries required by *Apache* extracted using *ldd*

```
libssl.so.4 ⇒ /lib/libssl.so.4 (0x4002d000)
libcrypto.so.4 ⇒ /lib/libcrypto.so.4 (0x40062000)
libresolv.so.2 ⇒ /lib/libresolv.so.2 (0x40153000)
libgssapi_krb5.so.2 ⇒ /usr/kerberos/lib/libgssapi_krb5.so.2 (0x40165000)
libkrb5.so.3 ⇒ /usr/kerberos/lib/libkrb5.so.3 (0x40178000)
libk5crypto.so.3 ⇒ /usr/kerberos/lib/libk5crypto.so.3 (0x401d6000)
libcom_err.so.3 ⇒ /usr/kerberos/lib/libcom_err.so.3 (0x401e7000)
libz.so.1 ⇒ /usr/lib/libz.so.1 (0x401e9000)
libaprutil.so.0 ⇒ /usr/lib/libaprutil.so.0 (0x401f7000)
libgdbm.so.2 ⇒ /usr/lib/libgdbm.so.2 (0x4020c000)
libdb-4.0.so ⇒ /lib/libdb-4.0.so (0x40213000)
libpthread.so.0 ⇒ /lib/libpthread.so.0 (0x402bb000)
libexpat.so.0 ⇒ /usr/lib/libexpat.so.0 (0x4030e000)
libapr.so.0 ⇒ /usr/lib/libapr.so.0 (0x4032e000)
libm.so.6 ⇒ /lib/libm.so.6 (0x4034c000)
libcrypt.so.1 ⇒ /lib/libcrypt.so.1 (0x4036e000)
libnsl.so.1 ⇒ /lib/libnsl.so.1 (0x4039b000)
libdl.so.2 ⇒ /lib/libdl.so.2 (0x403b0000)
libc.so.6 ⇒ /lib/libc.so.6 (0x403b4000)
/lib/ld-linux.so.2 ⇒ /lib/ld-linux.so.2 (0x40000000)
```

of *Apache* is *httpd*). As shown in the Table 3.2, a large number of transitions represent instructions defined in dynamic libraries (i.e. a large portion of program behavior is not defined in the main executable).

Table 3.2

A list of libraries and the number of transitions for each for *Apache*

Library name	#transitions
/usr/sbin/httpd	273
/usr/lib/libapr.so.0.0.0	203
/usr/lib/libaprutil.so.0.0.0	37
/lib/libc-2.3.2.so	98
/lib/libnss_files-2.3.2.so	10
/lib/libpthread-0.10.so	28
/usr/lib/httpd/modules/mod_cgi.so	22
/usr/lib/httpd/modules/mod_dir.so	2
/usr/lib/httpd/modules/mod_include.so	26
/usr/lib/httpd/modules/mod_log_config.so	4
/usr/lib/httpd/modules/mod_mime.so	2
/usr/lib/httpd/modules/mod_mime_magic.so	6
/usr/lib/httpd/modules/mod_negotiation.so	38
/usr/lib/httpd/modules/mod_userdir.so	4

In previous sections, the HPDA model was used to capture the behavior of the main executable and dynamic libraries using one integrated model. Therefore, although the dynamic library can be shared by several programs, the model of the library cannot be shared and must be learned again for each program. Moreover, an update of a library used in the program may require that the entire model to be relearned. The data in Table 3.2 demonstrate that a large portion of program behavior is contained in dynamic libraries and hence it is necessary to separate the models of libraries to solve the aforementioned problems. By sharing the models, well trained models for one program can help reduce the training time and false alarms for another program that uses the same dynamic libraries.

Although static analysis can capture most program behavior represented in the executable, the prevalent use of dynamic libraries makes it difficult to obtain much behavior by static analysis. Even if some of the libraries may be statically identified in the executable headers, many are identified and loaded dynamically. Table 3.1 shows a list of libraries used by *Apache* that can be statically identified in the executable header and Table 3.2 shows a list of libraries identified at run-time. A comparison of these tables demonstrates that several libraries used by *Apache* cannot be identified by static analysis of the program executable. Therefore, static analysis needs to be supplemented with dynamic learning to build a complete model. In this section, modified methods for static analysis and dynamic learning will be provided for producing CBHPDA models for programs.

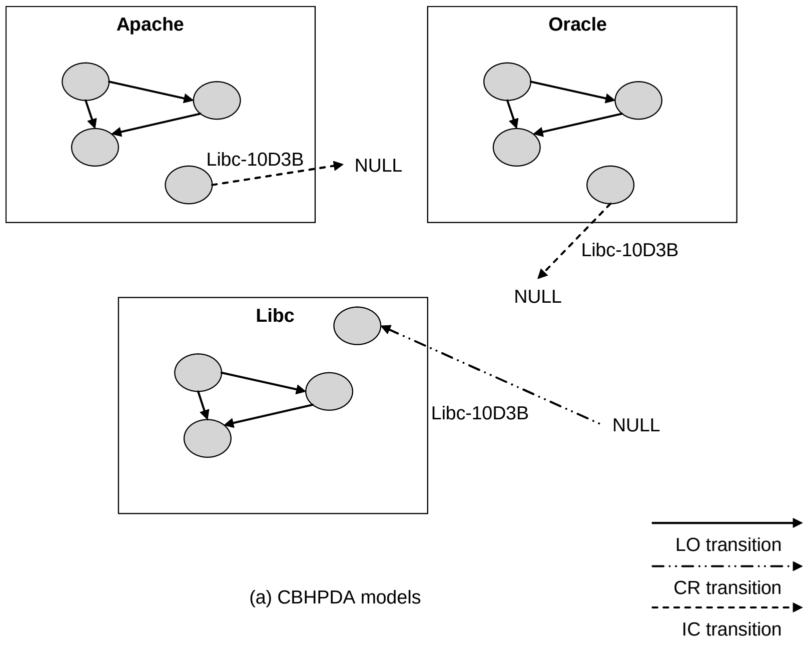
3.6.2 Definition of the CBHPDA

A CBHPDA model is produced for each execution component used by a program. When the CBHPDA model is used for intrusion detection, several CBHPDA models are connected together using the combination algorithm. The value of the executable id (exe_id) stored in the HPDA address provides an identifier of the execution component and thus is used by the learning algorithm to identify the boundaries of models and to divide the model into parts for each execution component.

Definition 6 *(Component-Based Hybrid Push Down Automata): An CBHPDA is a 5-tuple:* (S, Σ, T, s, A) *where* S, Σ, s, A *are the same as in the definition of HPDA.* T *is the set of transition functions* $T = (S \times \Sigma \rightarrow S) \cup (NULL \times \Sigma \rightarrow S) \cup (S \times \Sigma \rightarrow NULL)$ *where the types of transitions are:*

1. *LO transition —- A local transition is a transition that is referenced only within the model for one component. Each LO transition is an* $x \in (S \times \Sigma \rightarrow S)$.
2. *CR transition —- A cross transition defines an execution that crosses a model boundary and that has a dest node defined in the current component model. The source node of a CR transition is unknown. It defines an execution flow from an execution region outside the current CBHPDA model into the current model. Each CR transition is an* $x \in (NULL \times \Sigma \rightarrow S)$.
3. *IC transition —- A IC transition represents an execution flow from the current model to another model. The source node of the IC transition is defined in the current model and the dest node is unknown. Each IC transition is an* $x \in (S \times \Sigma \rightarrow NULL)$.

A graphical view of three CBHPDA models is shown in Figure 3.25.(a). Each CBHPDA model is identified by the component's execution id (*exe_id*). In our implementation, the *exe_id* is the position of the component's name in a hash table. In CBHPDA, an HPDA address uniquely identifies a transition. The CR transitions are definitions in the original model identified by the transitions' exe_id. An IC transition can be treated as an instance of a CR transition with an unknown *dest*. The *dest* of the IC can be found by finding a CR transition with the HPDA address indicated by the *exe_id* of the IC. The dest node of the CR transition defines the destination of the IC transition. Many models may contain the IC transitions that have the same HPDA address and their destination nodes are defined by a single CR transition. The combination algorithm assigns the destination state of each IC



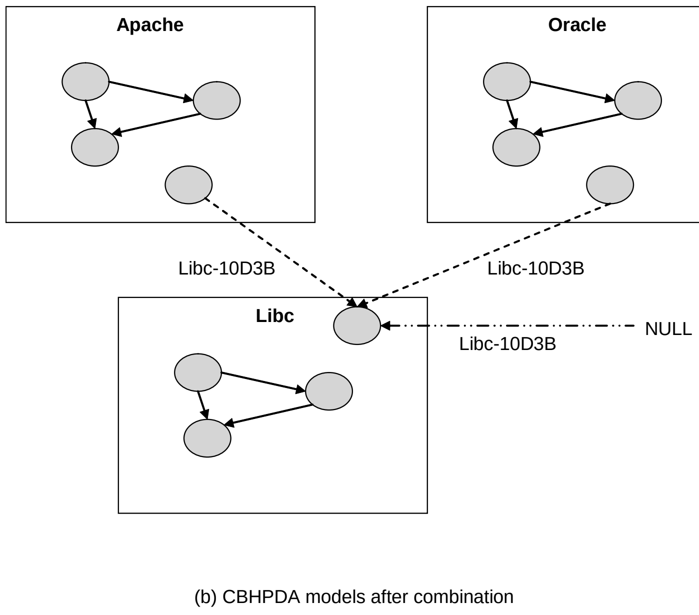


Figure 3.25

Illustrations of three types of transitions before and after model combination transition according to the value defined in the CR transition that has the same HPDA address as the IC transition. Since the CR transition is used for a definition, the source state of a CR transition is undefined (NULL in our implementation). CR transitions are used to patch the destination value of an IC transition. During model combination, the destination of an IC transition will be modified to the correct target as defined by its corresponding CR transition. The details about how CR and IC transition are used will be shown in the dynamic learning algorithm and the run-time combination algorithm in the later sections.

In the remainder of this paper, the value of exe_id in each HPDA address is also used as a model identifier. The expression $eid(m)$ returns the value of exe_id of m, a symbol in the Σ.

Properties of CBHPDA:

1. *All LO and CR transitions of a CBHPDA model have the same exe_id value as the identifier of the model. All IC transitions must have an exe_id value that is different from the model's identifier. The exe_id values of IC transitions in a CBHPDA model M represent the identifiers of models that M depends on.*

2. *All LO and IC transitions of a CBHPDA have a unique HPDA address values except the paired Entry and Exit transitions. All CR transitions have unique HPDA address value. An address value appearing in the CR transitions of a CBHPDA model can also appear in the LO transitions of the same model.*

3. *The symbol ϵ cannot appear in the CR and IC transitions.*

3.6.3 Combination of CBHPDA components

The process of combining CBHPDA models involves patching the target of the IC transitions. For example, since the exe_id uniquely identifies a CBHPDA model, the target of an IC transition that has an HPDA address $14 - AD4$ can be obtained by finding the

CR transition with the same HPDA address in the CBHPDA model with the identifier of 14 and using the target state defined in the CR transition as the target of the IC transition. Algorithm 10 shows the procedure for patching an IC transition. The function $TYPE(x)$ in the algorithm returns a value of LO, CR, or IC and $T(model\ identifier)$ represents the set of transitions contained in the model and indicated by the model identifier. In order to conduct intrusion detection for a program, it may be necessary to load several CBHPDA models that are required by the main executable of the program. The patching algorithm will connect the CBHPDAs into a large virtual model of the program. One example of three CBHPDA models after combination is shown in Figure 3.25.(b).

Algorithm 10 CBHPDA Patching for model combination

```
1:  procedure PATCH_CBHPDA(one_transition)
2:      if $dest(one\_transition) = NULL$ then          ▷ check if it is a IC transition
3:          $c\_model \leftarrow the\ model - its\ identifier = eid(symbol(one\_transition))$
4:          $et \leftarrow x \in T(c\_model) \wedge TYPE(x) = CR \wedge$
                $symbol(x) = symbol(one\_transition)$
5:          if $et = NULL$ then
6:              raise an error          ▷ this should not happen, every IC must have a
    corresponding definition of CR
7:          end if
8:          $one\_transition.dest \leftarrow et.dest$
9:      end if
10: end procedure
```

Figure 3.26

Algorithm 10: CBHPDA Patching for model combination

3.6.4 Static analysis for acquisition of an CBHPDA model

The steps for extracting CBHPDA models by static analysis are similar to those described in section 3.3. The three programs (extraction program, mapping program, and aggregation program) have extended functionality to generate the CBHPDA models. In any executable format, there are two special sections that record *import* and *export* functions. For any executable, *import* functions are the functions used in the current executable but defined in other libraries. *Export* functions are the functions defined in the current executable that may be invoked from other executables. The extraction program collects the names of *import* functions and *export* functions from the analyzed executables. The mapping program generates all LO transitions. The aggregation algorithm will read several intermediate models that are used by a program and create the IC and CR transitions. For example, the execution of program cat depends on three executables, cat, $libc.so.6$, and $ld-linux.so.2$. The intermediate models for the three executables generated by the mapping program must be presented to the aggregation program so that the IC transitions can be created. For every *export* function, the aggregation algorithm creates a CR transition for each non-ϵ transition that can be reached from the state of the function through zero or several ϵ transitions. This process is presented in Figure 3.27. For every call site that invokes an *import* function, the aggregation algorithm searches through the list of executables, finds the first *export* function that has the same name as the invoked *import* function, and creates an IC transition for each CR transition defined in this *export* function.

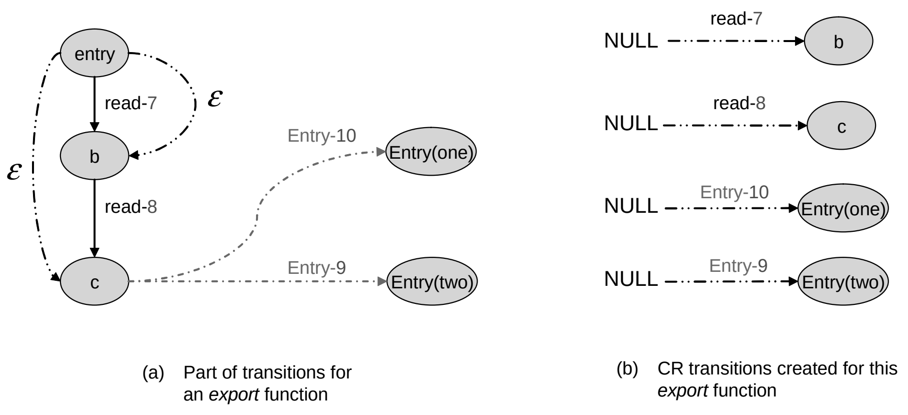


Figure 3.27

An example of transitions for an *export* function and CR transitions for it

3.6.5 Dynamic learning of CBHPDA models

Dynamic learning for CBHPDA models is similar to dynamic learning for HPDA models. Although both dynamically learned CBHPDA models and HPDA models represent execution flows between execution regions, the dynamic learning for the CBHPDA model divides the model into pieces for each execution component according to the exe_id information in the HPDA addresses. Algorithm 11 shows the dynamic learning algorithm for CBHPDA and Algorithms 12 and 13 define some functions used in Algorithm 11.

In the dynamic learning algorithm, the variable $last\_id$ records the exe_id of the last symbol. In the rest of the description, the current model refers to the model whose identifier is same as the exe_id of current symbol and the last model refers to the model whose identifier is same as the $last\_id$.

Algorithm 11 Dynamic learning of CBHPDA

```
    $last\_eid$                                  ▷ The value of $exe\_id$ of last symbol.

 1: procedure DYNAMIC_LEARNING_CBHPDA($s_{current}$, $m$)
 2:     $X \leftarrow out_T(\hat{s}_{current})$
 3:     $M \leftarrow symbols(X)$
 4:     if $m \in M$ then
 5:         $x \leftarrow x \in X \wedge symbol(x) = m$
 6:         PATCH_CBHPDA($x$)
 7:         $s_{current} \leftarrow dest(x)$
 8:         $last\_eid \leftarrow eid(m)$
 9:     else
10:         $c\_model \leftarrow eid(m)$
11:         if $last\_eid \neq eid(m)$ then          ▷ $m$ will associate with a IC transition
12:             LEARN_CROSS()
13:         else                                    ▷ $m$ will associate with a LO transition
14:             if $\exists x \in T(c\_model) \wedge symbol(x) = m \wedge TYPE(x) = LO$ then
15:                 if $\exists y \in T(c\_model) \wedge dest(y) = source(x) \wedge type(symbol(y)) \neq \epsilon$
    then
16:                     LEARN_2_1($c\_model, s_{current}, x$)
17:                 else
18:                     LEARN_2_2($c\_model, s_{current}, x$)
19:                 end if
20:                 $s_{current} \leftarrow dest(x)$
21:             else if $\exists x \in T(c\_model) \wedge symbol(x) = m \wedge TYPE(x) = CR$ then
22:                 LEARN_CROSS_L()
23:             else
        ▷ it is impossible that $\exists x \in T(c\_model) \wedge symbol(x) = m \wedge TYPE(x) = IC$
24:                 LEARN_3($c\_model, s_{current}, m$)
25:             end if
26:         end if
27:     end if
28: end procedure
```

Figure 3.28

Algorithm 11: Dynamic learning of CBHPDA

Algorithm 12 Functions LEARN_CROSS_L and LEARN_CROSS for Algorithm 11

```
1: procedure LEARN_CROSS_L($c\_model$, $s_{current}$, $x$)
2:     $y \leftarrow newtransition$
3:     $y.source \leftarrow s_{current}, y.dest \leftarrow x.dest$
4:     $y.symbol \leftarrow m, y.TYPE \leftarrow LO$
5:     $T(c\_model) \leftarrow T(c\_model) \cup y$
6:     $s_{current} \leftarrow dest(x)$
7: end procedure

8: procedure LEARN_CROSS($c\_model$, $s_{current}$, $m$)
9:     $l_model \leftarrow last\_eid$
10:    if $(\exists x \in T(c\_model) \wedge symbol(x) = m \wedge TYPE(x) = LO) \vee (\exists y \in T(c\_model) \wedge symbol(y) = m \wedge TYPE(y) = CR)$ then
11:        if $\neg\exists y \in T(c\_model) \wedge symbol(y) = m \wedge TYPE(y) = CR$ then
12:            $y \leftarrow newtransition$
13:            $y.source \leftarrow NULL, y.dest \leftarrow x.dest$
14:            $y.symbol \leftarrow m, y.TYPE \leftarrow CR$
15:            $T(c\_model) \leftarrow T(c\_model) \cup y$
16:        end if
17:        $y \leftarrow newtransition$
18:        $y.source \leftarrow s_{current}, y.dest \leftarrow x.dest$
19:        $y.symbol \leftarrow m, y.TYPE \leftarrow IC$
20:        $T(l\_model) \leftarrow T(l\_model) \cup y$
21:        $s_{current} \leftarrow x.dest$
22:        $last\_eid \leftarrow eid(m)$
23:    else
24:        $s\_1 \leftarrow newstate$
25:        $y \leftarrow newtransition$
26:        $y.source \leftarrow NULL, y.dest \leftarrow s\_1$
27:        $y.symbol \leftarrow m, y.TYPE \leftarrow CR$
28:        $T(c\_model) \leftarrow T(c\_model) \cup y$
29:        $S(c\_model) \leftarrow S(c\_model) \cup s\_1$
30:        $y \leftarrow newtransition$
31:        $y.source \leftarrow s_{current}, y.dest \leftarrow s\_1$
32:        $y.symbol \leftarrow m, y.TYPE \leftarrow IC$
33:        $T(l\_model) \leftarrow T(l\_model) \cup y$
34:        $s_{current} \leftarrow s_1$
35:        $last\_eid \leftarrow eid(m)$
36:    end if
37: end procedure
```

Figure 3.29

Algorithm 12: LEARN_CROSS_L and LEARN_CROSS for Algorithm 11

Algorithm 13 Functions LEARN_2_1, LEARN_2_2, and LEARN_3 for Algorithm 11

```
 1: procedure LEARN_2_1($c\_model$, $s_{current}$, $x$)
 2:     $s_1 \leftarrow new(state)$
 3:     $s_2 \leftarrow source(x)$
 4:     $T(c\_model) \leftarrow T(c\_model) \cup \{s_2, s_1, \epsilon\}$
 5:     $source(x) \leftarrow s_1$
 6:     $T(c\_model) \leftarrow T(c\_model) \cup \{s_{current}, s_1, \epsilon\}$
 7: end procedure

 8: procedure LEARN_2_2($c\_model$, $s_{current}$, $x$)
 9:     $T(c\_model) \leftarrow T(c\_model) \cup \{s_{current}, source(x), \epsilon\}$
10: end procedure

11: procedure LEARN_3($c\_model$, $s_{current}$, $m$)
12:     $s_1 \leftarrow new(state)$
13:     $T(c\_model) \leftarrow T(c\_model) \cup \{s_{current}, s_1, m\}$
14:     $s_{current} \leftarrow s_1$
15: end procedure
```

Figure 3.30

Algorithm 13: LEARN_2_1, LEARN_2_2, and LEARN_3 for Algorithm 11

If the exe_id of the current symbol m does not equal $last\_id$, the execution flow is crossing the boundary between two models. In this case, the current state $s_{current}$ is a node in the last model. Since control is flowing outside the last model, a new IC transition associated with symbol m should be created from $s_{current}$ to a state in the current model. In the function LEARN_CROSS defined in Algorithm 12, if there is no LO and CR transition in the current model labeled with the same symbol as the current symbol, a new state should be created in the current model, a new CR transition having the new state as *dest* state should be created in current model, and a new IC transition should be created in the last model connecting $s_{current}$ and the new state. If there exists one LO transition that is labeled with m but no CR transition is satisfied, a CR transition must be created and the *dest* node of the existing LO transition must be used as its *dest* node. An IC transition is also created in the last model that connects $s_{current}$ and the *dest* node of the CR transition. If a satisfied CR transition is found in the current model, it only needs to create an IC transition in the last model that connects $s_{current}$ and the *dest* node of the existing CR transition.

If the exe_id of the current symbol equals $last\_id$, the execution flow remains in the same model and an LO transition should be added into the current model. The functions LEARN_2_1, LEARN_2_2, LEARN_3 defined in Algorithm 13 handle the same cases as those defined in Algorithm 6. Function LEARN_CROSS_L handles the case when a CR transition labeled with the current symbol m exists in the current model. In this situation, the transition associated with symbol m has been referenced from an *execution region* outside the current model and now this transition is also referenced from inside the current model. This happens when a function defined in a dynamic library is called from other execution components and inside the library. In this case, a new LO transition is created and its *dest* state is assigned to the state as the destination of the existing CR transition.

During learning, the destinations of all IC transitions are not NULL. However, when the models are saved to disk, the destinations are set to NULL as indicated in the definition of the CBHPDA because the CBHPDAs are separate and do not have information about other models. The information is patched to the correct value during model combination.

CHAPTER IV

EXPERIMENTS

Prototype systems based on the models described in Chapter III have been developed in Red Hat Linux 9.0. In this chapter, we describe experiments that were conducted to demonstrate the effectiveness of the HPDA approach. The prototype systems that were developed are described, the data sources that were used in the experiments and performance criteria are presented, the effectiveness of the method for detecting attacks is evaluated, the results measuring the performance of the different learning algorithms is analyzed, and the overhead penalty of the prototypes is presented.

4.1 Prototype system implementation

We first describe the overall architecture of our prototype implementation and then give a detailed description of a number of technical implementation issues.

4.1.1 System architecture

The model building and intrusion detection system features two main components:

- *Binary analyzer* —- This analyzer extracts models from the binary executable of programs. Detailed descriptions of the functions used for static analysis can be found in sections 3.3 and 3.6.4.

- *Real-time intrusion detection system* —- The prototype IDS is composed of functions to collect call stack logs, perform dynamic learning at runtime, and conduct real-time intrusion detection.

A plug-in for IDA Pro [1] was developed to extract call graphs from the assembly information collected by IDA Pro. The mapping and aggregation programs are implemented as a standalone application that generates the models from graphs for every component for a program.

The real-time intrusion detection system was developed as a kernel patch and two loadable kernel modules for Linux. The kernel patch contains only a small number of modifications of the Linux kernel that introduce extra data in *task_struct* for maintaining the intrusion detection status, insert codes after exit processes for deleting the extra data, add code to handle signals, and provides the functionality to abort or postpone execution of a process. The kernel patch also provides the capability to insert hook functions before and after system call invocations. This is the basis for all system call interposition based intrusion detection systems. The amount of code and functions implemented in the kernel patch has been kept as small as possible so that it is easy to validate its correctness and security because any vulnerability in the kernel is critical and has the potential to give the attackers *root* privilege. Moreover, limiting the modifications of the kernel make it easier to port the kernel patch to a new version of the kernel or other system.

An IDS service loadable kernel module was implemented to provide basic functions that can be shared by different IDS implementations. For example, *VtPath* [32], *Vt-Static* [31], *execution graph* [37], and our approaches all require extraction of call stack information. Currently, the service module provides interfaces to collect system call and return addresses information.

The HPDA IDS kernel module implements the functions that dynamically learn an HPDA or CBHPDA model at run-time and conduct real-time intrusion detection. The module can also save the call stack audit log into a file for off-line learning. Since the module is only based on the functionalities provided by lower level kernel patch and service module, it is independent of machine architecture and can be easily ported to different architectures with no modifications.

In the HPDA definition, a paired address (HPDA address) is used to accommodate the dynamic characteristics of dynamic libraries. In any operating system, there exists a data structure for each process to manage the process's memory space. We created a new data structure to record the name, begin address, and end address of each execution component for each process in order to save the overhead that would be incurred by extracting such information from the kernel maintained data structure every time an absolute address is converted. Three system calls will lead to a change in memory mapping, *mmap*, *munmap*, and *brk*. An update of values in the new data structure occurs after any of the three system calls is invoked. If the memory map of a stack does not have a file name associated with it, a NULL name is used and an alarm will be generated. Any code execution on the stack may generate a return address belonging to stack area and thus trigger an alarm. Many implementations of exploits place their shell code on the stack and require execution of code on the stack. Unless the attacks manipulate the stack in an extremely clever manner, they can be easily detected by our component for address transformation.

The data structure above is implemented as a hash table of pointers to the names of executables so string comparisons are not required when locating the position of the name. This greatly improves the efficiency of the process of converting an absolute address to an HPDA address. All executable names stored in the hash table are shared by many processes and names can be inserted into and deleted from the table. Since the *exe_id* may change during updates, the *exe_id* cannot be stored directly in the data structure, but must be computed every time it is needed.

In the HPDA IDS kernel module, a program model is shared by all processes that are instances of this program. For each process, the module only maintains the current state, last call stack, and the data structure for memory mapping mentioned above.

A configuration file is used to define a list of programs that are being monitored and protected by the detection system. The file simply lists the name of the programs. The detection module looks up the name of the process in this name list. If the process's name is in the list, detection is conducted on this process. When a process invokes a system call *execve()* and executes another program, the process will not be monitored in our design if the executed program has a different name. For example, *Apache* will execute a new program for each CGI request. In our system, the execution of the CGI programs is ignored since the processes for the CGI programs do not have the same name as the processes for *Apache*. If the users want to monitor the CGI programs, the names for the CGI programs must be put in the configuration file.

4.1.2 Implementation issues

We have developed methods to handle the implementation issues for behavior based detection system.

4.1.2.1 HPDA address

Conversion from an absolute address to an HPDA address is required whenever a system call is invoked and this imposes extra overhead on the kernel. It is possible to eliminate this computation if the HPDA addresses are converted back to absolute addresses when the models are loaded into memory and a recalculation is conducted whenever the memory mapping for a process is changed. In this setting, the model cannot be shared by the processes that are instances of the same program, because processes may be loaded at different addresses. Recently, this randomness property has been treated as a security mechanism by many operating systems to defend against buffer overflow attacks. Our experiments (described in section 4.7.1.2) demonstrate that the overhead incurred by address transformation is acceptable and so we did not pursue the solution that converts HPDA addresses back to absolute addresses.

4.1.2.2 Signal handling

Signals are a mechanism used by operating systems to allow interactions between user space processes and the kernel. Signals notify user space processes of system events from the kernel [13]. In Linux, the kernel checks whether a signal for any processes has arrived when the execution is switching from kernel mode to user mode. This happens at almost every timer interrupt (roughly every 10 ms) [13]. Handling the signal requires process switching to a handler function and restoration of the original execution context after the function returns. This can happen at any point during the process's execution. If signal handling is to be represented in a model, it must be possible for the model to move from any state to the portion of the model that represents the behavior of signal handler. We agree with the assertion of Wagner et al. [76] that it is not feasible to add an extra transition from each state to each possible signal handler. We have observed that very little program behavior is from signal handling. For example, when Apache is used to run a website with CGI programs, Apache never uses signals. Therefore our current implementation does not represent the behavior of signals. However, our implementation does include a mechanism that notifies the detection system before and after signal handling. Because the signal events are captured by our system, it would be easy to extend the system to monitor signal handlers when signals are received.

4.1.2.3 Thread

There are two types of thread implementations: user-space threads and kernel-space thre-ads. User-space implementations avoid the kernel and manage the resources itself. From the kernel point of view, a multithreaded application based on user-space threads is just a normal process. The multiple execution flows of a multithreaded application are created, handled, and scheduled entirely in user mode. As discussed in [76], the context-switching operation introduces another type of implicit control flow.

Linux uses kernel-space threading to generate lightweight processes that offer better support for multithreaded applications. Basically, two lightweight processes may share some resources such as the address space, open files, and so on. Whenever one of them modifies a shared resource, the other immediately sees the change. Each lightweight process has a separate *task_struct* data structure that is used by the kernel to maintain process specific information. Moreover, each lightweight process can be scheduled independently by the kernel.

For the dynamic learning approach, the context-switching operations in user-space threading is just part of the program behavior and thus no special consideration needs to be given. For kernel-space threading, since the detection module maintains the information in the *task_struct* structure for each process and light weight process, our implementation does not have any problem handling kernel-space threads. However, our implementation may miss some behavior of multithreaded applications that make extensive use of signals for communication since our current implementation ignores the behavior of signal handling.

4.2 Experimental methods

The prototype system was implemented in RedHat Linux 9.0 with kernel version 2.4.20-8. All experiments were conducted on a Pentium 4 3.2GHz machine with 1G memory and 80G hard disk. In this section, we describe the performance criteria used for model evaluation and the experimental design with which measurements were performed.

4.2.1 Data sources: programs and data sets

Programs to be monitored in our experiments were chosen to demonstrate the capabilities of our approach with programs exhibiting different kinds of behavior. Since our goal is to demonstrate that our methods can be used to develop practical real-time intrusion detection systems, some large real world applications were among those chosen for evaluating the prototype system. Five programs shipped with Red Hat Linux were used for our experiments: *cat*, *gzip*, *kon*, *Apache*, and *VsFtp*. The program *cat* represents applications with a workload consisting of extensive I/O operations that do little computation. The program *gzip* represents a computation-intensive application. The program *kon* is a Kanji emulator for the console that allows the user to read Japanese on the console. This program was chosen because documented vulnerabilities are available for the program. The programs *cat*, *gzip*, and *kon* only use two dynamic libraries: *libc* and *ld-linux*. Two network servers, *Apache* and *VsFtp*, were chosen to test performance under real world workloads. To test the ability of our system to detect attacks, we also selected another program *xv* that is an interactive image manipulation program for the X Window System and comes preinstalled in many Unix system. The programs *cat*, *gzip*, *Apache*, and *VsFtp* were used to test the ability of our approaches to model normal program behavior. Two attacks were implemented on the programs *kon* and *xv*. Both exploit buffer overflow vulnerabilities residing in the software. The programs represent different levels of complexity ranging from the simple command line *cat* to the complex network server *Apache* and the interactive X windows application *xv*. Table 4.1 presents some statistics about the complexity of these programs including the number of internal functions, number of external functions, and number of dependent libraries used by each. Note that the name of the *Apache* web server executable is *httpd* and these names are used interchangeably in the remainder of this paper.

Table 4.1

Statistic information for tested programs

Program	Main executable size	#Internal funcs	#External funcs	#libraries Used
cat	14.3KB	34	35	2
kon	45.2KB	138	89	2
gzip	53.7KB	106	48	2
vsftpd	72.5KB	388	109	7
httpd	308.9KB	592	586	20
xv	1.2MB	1337	302	5

Three lossless data compression software benchmarks were used to obtain the behavior of *gzip* and to evaluate the performance overhead when *gzip* is monitored. The *Calgary Corpus* (ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/) contains nine different types of text files for evaluating text compression schemes and was first represented in the book “Text compression” [9]. The *Canterbury Corpus* (http://corpus.canterbury.ac.nz/) file set has been developed specifically for testing new compression algorithms and works as a replacement for the *Calgary Corpus*. The files were selected based on their ability to provide representative performance results. The last benchmark from the “Maximum Compression” website (http://www.maximumcompression.com/) includes up-to-date evaluations of most of the latest compression applications. This data set contains a great variety types of files and the descriptions are listed in Table 4.2.

Table 4.2

A list of different types of files in the “Maximum Compression” benchmark

Type	Content	Size
English text	1995 CIA World Fact Book	2.9 Mbyte
Logfile	Fighter-planes.com traffic log file	20 Mbyte
Sorted Word-list	Alphabetically sorted English word-list	4 Mbyte
HTML files	750 concatenated HTML files	17.3 Mbyte
Exe	Acrobat Reader 5.0 executable	3.7 Mbyte
DLL	Microsoft Office 97 DLL	3.6 Mbyte
Bitmap	1356x1020 pixels / 16.7 million colors	4 Mbyte
Jpg/Jpeg	1152x864 pixels / 16.7 million colors	823 Kbyte
HLP	Delphi First Impression OCX Help file	4.0 Mbyte
DOC	Occupational Health and Safety; MS Word file	4.0 Mbyte
PDF	Macromedia Flash MX Manual; Acrobat document	4.4 Mbyte

A benchmark named *WebBench* that tests web server performance developed by VeriTest and PC Magazine was used to evaluate the performance overhead when monitoring *Apache* web server. WebBench lets users measure Web server software performance by running different Web server packages on the same server hardware or by running a given Web server package on different hardware platforms. WebBench's standard test suites produce two overall scores for the server: requests per second and throughput as measured in bytes per second. This benchmark has been used in many reviews from PC Magazine. Though currently VeriTest and PC Magazine no longer support this benchmark, it provides a valuable test set for measuring system overhead of the learning and monitoring process.

Audit data for each program is required to test the performance of our learning algorithm. We have run each program with a wide variety of workloads and different execution paths to simulate real world use of the program. There are 11 switches for the program *cat*. A *perl* script was written to exercise different combinations of these 11 switches and to use *cat* to concatenate or show the files in the three compression benchmark suites mentioned above. A total of 200 executions traces were generated for training data and 200 for test data. Around 10% of the executions in the test set use a command combination that did not appear in the training data. We used similar methods to obtain traces representing the normal behavior of *gzip* that uses 13 switches.

To generate a normal workload for *Apache*, two personal homepages of students from our department were set up on the machine used for testing. The workload for these two homepages contains HTML and CGI program requests. A PHP based groupware system named *phpGroupWare* was also set up on the test machine. The *phpGroupWare* provides online ftp, email, a calendar, an address book, forum, etc. for a total of more than 20 different applications and uses a *MySQL* database to store the user information. The *WebBench* benchmark was also used to generate the workload.

We have written a shell script to generate the workload for the *Vsftpd* server. The script exercised about 70 commands provided by the ftp client that comes with RedHat Linux. In the 70 commands, around one third do not generate any workload in the server side and some generate commands that are not supported by the *Vsftpd* server. The files transferred in the test were those from the compression benchmark and in the groupware.

4.2.2 Performance criteria

Since our static analysis method does not apply any approximations, it cannot be used for intrusion detection directly. A base model generated by static analysis must be supplemented by dynamic learning. Two measures commonly used to evaluate models acquired from audit data are the amount of data required to learn an accurate model (convergence speed) and the number of false positives generated. Models acquired by purely static analysis are typically assumed not to encounter false positives, so the false positive rate for these systems is generally not reported.

Convergence speed is a measure of how fast a model can be learned as a function of the size of the training data. The training process is considered to have converged when the size of the normal profile stops increasing (i.e., no new behavior added). The convergence speed is important because faster convergence means that less time, effort and training data are needed to produce an accurate model. All evaluated methods activate their learning process when a system call is invoked. Each system call invocation may or may not trigger the learning algorithm to acquire new information. The number of system calls that have triggered learning provides a consistent measure of the learning activity across detection systems. Although different algorithms have different internal representation of profiles, all methods activate learning at the time of a system call invocation. A long sequence from the call stack log like that shown in Figure 3.1 provide the input for the learning algorithms. At any point during learning, the convergence speed is computed as

$$Convergence\ speed = \frac{Num\ of\ system\ calls\ so\ far\ that\ have\ triggered\ learning}{Total\ num\ of\ system\ calls}$$

False positives, also called false alarms, are alarms raised when no attacks or anomalies are present. False negatives represent failure to issue an alarm when an anomaly has occurred. Section 4.3 demonstrates that common exploits available on the Internet all modify the execution flow without special manipulation of the stack and thus will be detected by our system. Therefore, false negatives were not considered in our experiments. During the evaluation, test data only contains the audit data from normal execution and no anomaly data is contained in test data. Therefore, any alarms that are generated from the test data are false alarms. The false positive rate is computed as

$$False\ positive\ rate = \frac{Num\ of\ system\ calls\ where\ an\ alarm\ is\ raised}{Total\ num\ of\ system\ calls}$$

Timing data was collected using the Linux command *time* for the performance measurements. This command returns three types of timing data for each execution, *real*, *user*, and *sys*. The *real* time is the time between program start and end. The *user* time measures the total number of CPU-seconds that the process spends in user mode. The *sys* time measures the total number of CPU-seconds that the process spends in kernel mode. Although the *real* time represents the user's perception of the program execution duration, the overhead can be hidden by the overwhelming I/O time and context switching that may vary significantly due to the workload on the current system. The *sys* time is a more accurate measure of the overhead introduced in the kernel. We have used all three time measurements to evaluate the performance of our prototype system.

4.2.3 Baseline methods

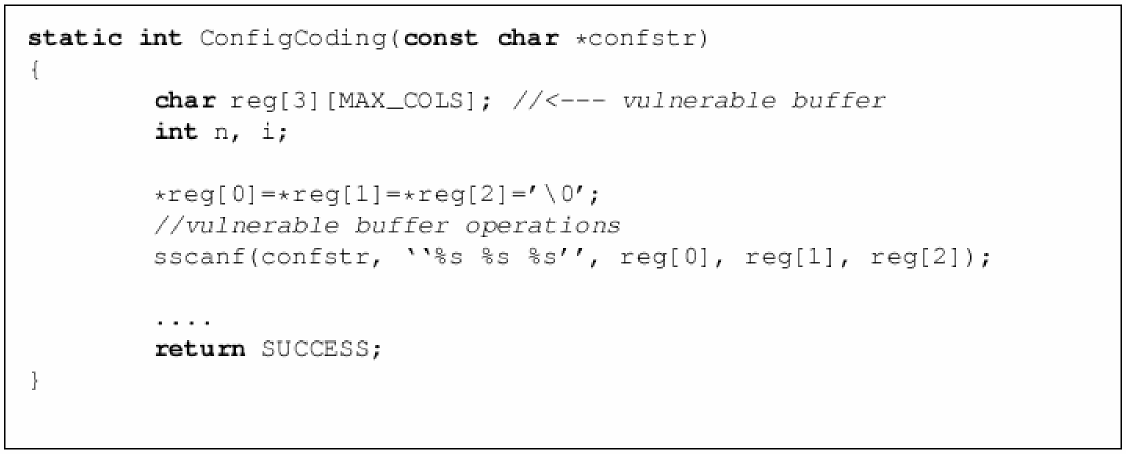
In our experiments, we will compare our approaches with two other methods, *n-gram* and *VtPath*. The *n-gram* method introduced in section 2.1.1 is a baseline method used in most papers regarding intrusion detection based on sequences of system calls. In our experiments, a window size 6 was used for the *n-gram* method. The *VtPath* method introduced in section 2.1.2 is the first method that uses call stack information. Since the *VtPath* is also a dynamic learning based method, we will compare our approach with it. Many recent approaches acquire their models only using static analysis. These approaches use approximations to handle unusual control flow and thus do not have false positives. Therefore, we will only compare the performance overhead of our system with the performance data reported in their papers.

4.3 Detection of attacks

In this section, we examine the capability of our prototype system to detect real world attacks and synthetic attacks.

4.3.1 Common exploits available on the Internet

The ability to detect buffer overflow, Trojan horse, format string, and other attacks using a behavior based model similar to ours has been demonstrated by a number of research groups [31,34,41,42,71,76]. Intuitively our approach has the same ability as theirs to detect basic attacks that modify program execution flows. To demonstrate the effectiveness of the HPDA based intrusion detection system, attacks on two programs in Linux (*xv* and *kon2*) were implemented and conducted. The program *xv* is an interactive image manipulation program for the X Window System and has been preinstalled in many UNIX systems. The program *kon2* is a Kanji emulator for the console that allows users to read Japanese on the console and it is a *setuid* root application program. The implementations of the exploits used common techniques available on Internet web sites and in books and were not tailored to evade a system call based intrusion detection system. Since this kind of exploit is widely used in real world viruses, worms, and attacks, the capability of our system to detect these exploits is demonstrated in this section.



```
static int ConfigCoding(const char *confstr)
{
        char reg[3][MAX_COLS]; //<--- vulnerable buffer
        int n, i;

        *reg[0]=*reg[1]=*reg[2]='\0';
        //vulnerable buffer operations
        sscanf(confstr, ``%s %s %s'', reg[0], reg[1], reg[2]);

        ....
        return SUCCESS;
}
```

Figure 4.1

Buffer overflow vulnerability in *kon2*

In the version 0.3.9.b of *kon2*, a buffer overflow vulnerability exists in function *ConfigCoding()* [14]. When using the -*Coding* command line switch, the user provided argument may overflow the local buffer defined in function *ConfigCoding()*. This vulnerability, if appropriately exploited, can lead to local users being able to gain root privilege. Figure 4.1 presents an abbreviation of the code for the function *ConfigCoding()* and shows the vulnerability. The vulnerable buffer and the possible overflow operations reside in the same function.

The program *xv* versions 3.10a and prior has a buffer overflow vulnerability that allows a malicious attacker to construct an image that will exploit the vulnerability and execute code on the target victim's machine [15]. Figure 4.2 shows the vulnerable codes in program *xv*. This vulnerability is more complicated than the one in *kon2*. The vulnerable buffer and the vulnerable operations reside in different files and functions.

Two kinds of attacks were implemented on these programs, shellcode [4, 29, 49, 64] and return-into-lib(c) [62, 82] attacks. In the shellcode attack, the faked return address redirects the execution to a piece of shell code and the shell code invokes the system call *execve* and spawns a new shell. Since the attack invokes the system call directly through an instruction "*int 80*", it can escape detection systems based on analysis of library call usage such as *IAM* [43]. The return-into-lib(c) attacks are generally considered to be difficult to prevent and detect. For example, non-executable stack [24] and program shepherding [46] techniques are incapable of preventing this kind of attack. In our experiments, the attacks overwrite the original return address with the address of the library call *system*. A successful execution of the attack also spawns a new shell.

All attacks were detected successfully by our system. It is interesting to observe that the attacks are detected very early in the detection process during the address transformation process. In program memory mapping, the memory space of a stack is not mapped to any disk file. In our implementation, the address transformation function generates an alarm if a return addresses appears in a section that is not associated with any disk file. Note that the implementations of the attacks that we are using do not take special care to manipulate the stack pointer. This means that the shell code attack generates a return address on the stack and the return-into-lib(c) attack generates a random address for a nonexistent section of code or a section that is not associated with any disk file. Therefore,

```
static int openPic(filenum) //in file xv.c
        int filenum;
{
        ....
        PICINFO pinfo; //vulnerable buffer
        ....
}

int LoadBMP(fname, pinfo) //in file xvbmp.c
        char    *fname;
        PICINFO *pinfo;
{
        ....
        if (biBitCount!=24) {
                int i, cmaplen;
                cmaplen = (biClrUsed) ? biClrUsed : 1 << biBitCount;
                //vulnerable buffer operations
                for (i=0; i<cmaplen; i++) {
                        pinfo->b[i] = getc(fp);
                        pinfo->g[i] = getc(fp);
                        pinfo->r[i] = getc(fp);
                        if (biSize != WIN_OS2_OLD) {
                                getc(fp);
                                bPad -= 4;
                        }
                }
        }
        ....
}
```

Figure 4.2

Buffer overflow vulnerability in *xv*

the address transformation module generated alarms for each attack used in our experiments. A highly sophisticated attack could manipulate the stack [38] to appear normal to the detection system. However, such attacks will have large shell codes and are more difficult to implement [38].

In our system, both the last call stack and the *EBP* values of the last call stack are recorded. When an anomaly is detected, a list of return addresses is extracted using the recorded *EBP* values. By comparing this list of addresses and the last call stack, we can determine the functions that have not returned when the attack happens. In many instances, analysis of this information can be used to help determine the vulnerability that was used by the stack based buffer overflow attack. There are three cases that can occur when attacks take advantage of stack based buffer overflow vulnerabilities.

1. The vulnerable buffer and the vulnerable operations are in the same function. There exists a call site in the function that will lead to a system call and this call site is prior to the vulnerable operations. In this case, the list of addresses and the last call stack are compared and the first address in last call stack that is different indicates the return address of the vulnerable function that contains the vulnerable buffer and the vulnerable operations.

2. The vulnerable buffer and the vulnerable operations are in different functions. There exists a call site in the function that will lead to a system call and this call site is prior to the vulnerable operations. The vulnerable function is identified as in the first case. This function contains the vulnerable buffer. All codes that can be reached from this function must be studied to reveal the vulnerable operations.

3. There is no call site in the function prior to the vulnerable operations that will lead to a system call. In this case, there is insufficient information to determine the location of the vulnerable function.

Of the vulnerabilities used in our experiments, the *xv* vulnerability falls in case 2 and the *kon2* vulnerability falls in case 3. If a multilevel detection system is used in which library calls are monitored in addition to system calls, additional execution traces become available for post-study of the attacks. For example, if library functions are monitored, the *sscanf* will generate an event before the attack happens in *kon2* and thus the vulnerability for *kon2* becomes an instance of case 1.

It should be noted that in the previous discussion we have assumed that the return addresses of functions that have not returned are not destroyed by the attack. However, the attacker may overflow the entire stack and destroy all the return address information.

4.3.2 Function pointer attacks

Another kind of buffer overflow attack causes overwriting of a function pointer or a *longjmp* buffer. This kind of attack can evade the detection of static analysis based approaches, such as IAM [43], Dyck [41, 42], VPStatic [31], NDFA, and PDA [76].

Wilander and Kamkar present a detailed description of different kinds of buffer overflow attacks and compare different tools for dynamic buffer overflow prevention [81]. One interesting observation is that attacks overflowing function pointers and *longjmp* buffers can circumvent almost all methods, especially when the pointer and buffer are on the heap/BSS/data section. In this section, a buffer overflow attack overwriting function pointer is described and we demonstrate that this kind of attack can evade detection by methods that depend entirely on static analysis, but can be detected by our method. Since we do not have access to the implementations of other methods that use static analysis, all the tests were performed manually using the algorithms described in the papers.

Approximations are used in static analysis based approaches to handle the indirect calls where the target of the function call is not revealed until runtime. The approximation simply assumes that "every pointer could refer to any function whose address has been taken" [76]. This simple assumption introduces many impossible execution paths that may be exploited by attackers to evade detection by these approaches.

The following shows a segment of code containing a buffer overflow vulnerability. The function call *gets* in function *f* and *g* can be used to overflow *buf* in the BSS section and overwrite the function pointer *do_operation*. An attack may exploit the impossible execution paths introduced by the approximation to evade detection.

```
char buf[10];
int (*do_operation)(int op);
int privileged_operation_one(int op)
{
        seteuid(0);
        privileged_operation......
        seteuid(getuid());
}
int privileged_operation_two(int op)
{
        seteuid(0);
        privileged_operation......
        seteuid(getuid());
}
int non_privileged_operation_one(int op)
{
        seteuid(100);
        non_privileged_operation......
        seteuid(getuid());
}
int f(int fd)
{
        int op;
        ...privileged_initiliztion...
        gets(buf); //buffer overflow vulnerability
        op = atoi(buf);
        return do_operation(op);
}
int g(int fd)
```

```
{
        int op;
        ...non_privileged_initializtion...
        gets(buf); //buffer overflow vulnerability
        op = atoi(buf);
        return do_operation(op);
}
int main()
{       ...
        if(getuid()==50) {
                do_operation=privileged_operation_one;
                f();
        }
        else if(getuid()==60) {
                do_operation=privileged_operation_two;
                f();
        }
        else {
                do_operation=non_privileged_operation_one;
                g();
        }
        ...
}
```

The correct operation of the code is as follows. The program assigns different operations with different privileges based on the user's *uid* by using a function pointer *do_operation*. The operation is then invoked in function *f* and *g*. In function *f*, an operation with high privilege is always called, but in function *g*, it is impossible to access the privileged operation.

However, a buffer overflow vulnerability exists in the above implementation. Attackers can overflow the variable *buf* using *gets* and overwrite the function pointer *do_operation*. In this case, an attacker with low privilege can overwrite the function pointer, making it point to the functions for a privileged operation, and thus enable execution of the privileged operation.

Since in static analysis, it is difficult to capture the execution path for a function pointer, a tradeoff method that has been used by other researchers is to mark the call-sites of indirect calls as targeting any function whose address is known. Therefore, the call site for the function pointer *do_operation* points to all functions defined in this program. This tradeoff will allow the attack described above to occur because the *do_operation* call site in function *g* can refer to the privileged operation.

Our approach can detect this attack. In function *g*, *gets* uses the system call *read* and we record the call stack at this point. If an attacker uses the buffer overflow attack and overwrites the function pointer to point to the function *privileged_opera-tion_one*, the next invocation of a system call will be *seteuid* in the function *privileg-ed_operation_one*. Since the addresses for this *seteuid* and *read* are unique in our model, the sequence of these two operations cannot appear in the normal training data and our method is able to detect this anomaly. Similar attacks can be easily implemented with the *longjmp* buffer.

4.4 Evaluation of the dynamic learning method with generalization

In section 3.4.3, a dynamic learning method is discussed that uses the assumption that every function has a single entry and exit point. In the remainder of this paper, the term DA_HPDA is used as the name of this dynamic learning method and D_HPDA is used to refer dynamic learning methods that do not use this assumption. The assumption of a single entry and exit point allows the DA_HPDA algorithm to learn more general models of program control flow as discussed in section 3.4.3. However, the assumption of single function entry and exit point does not always hold in real world applications. In this section, we report measurements of the convergence speed and false positive rates of the DA_HPDA model to demonstrate the advantages of the generalization it affords. The binary codes of several executables were also studied to determine the extent of violation the assumption of DA_HPDA approach. At the end of this section, the computational overhead of DA_HPDA approach is also presented.

4.4.1 Acquisition of normal behavior with DA_HPDA

In this section, the convergence speeds and false positive rates for D_HPDA and DA_HPDA are measured with several different programs and the results are compared with the results for *n-gram* and *VtPath*. Figure 4.3 and 4.4 present the convergence speed and false positive rate with the program *cat*. Since the behavior of *cat* is very simple, the generalization of the DA_HPDA approach does present any advantage in this case. *VtPath*, D_HPDA, and DA_HPDA have essentially the same convergence speed. In the figures the three curves overlap to form one line and cannot be distinguished. The cumulative number of system calls that trigger learning with *n-gram* is much higher than the other methods because this method learns short sections of exact paths and does not have a general representation of control flow. All four methods exhibit similar false positive rates for this simple program.

Figure 4.5 and 4.6 presents the results from program *gzip*. In this case, the convergence speed of our method is substantially better than that of both *VtPath* and *gzip* due to the bet-

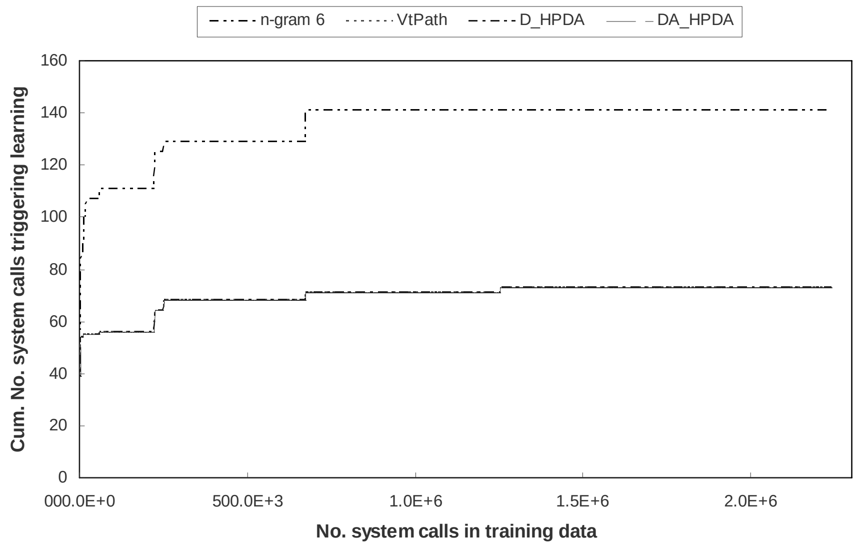


Figure 4.3

Comparison of convergence speeds of *D_HPDA* and *DA_HPDA* for *cat*

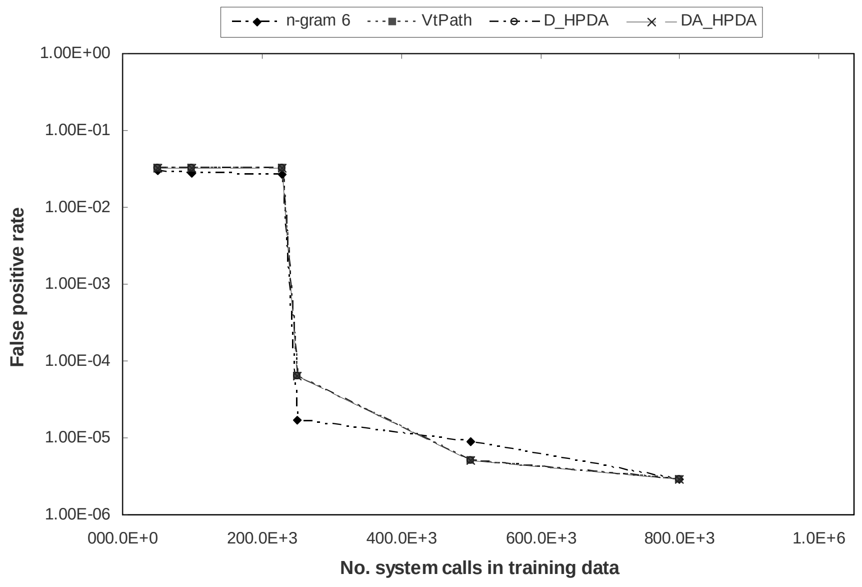


Figure 4.4

Comparison of false positive rates of *D_HPDA* and *DA_HPDA* for *cat*

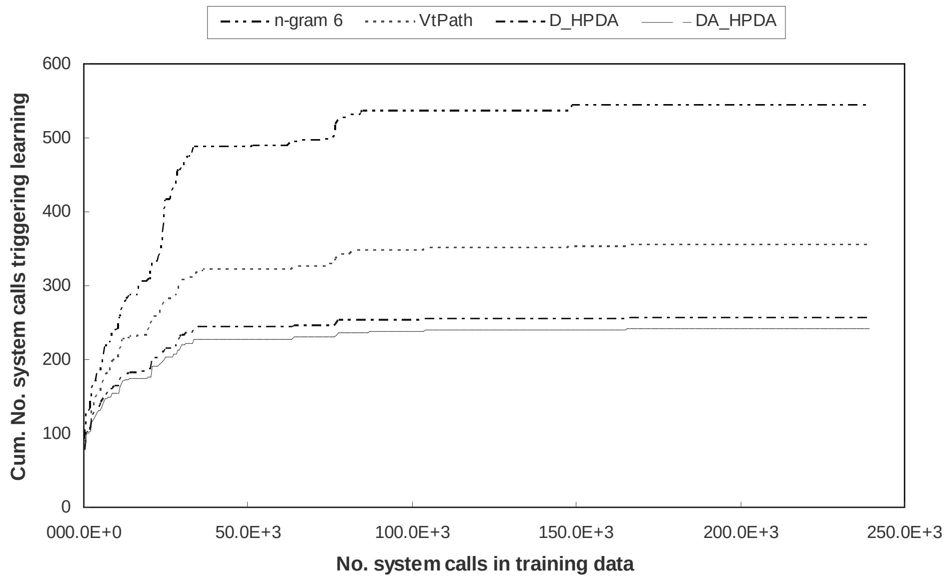


Figure 4.5

Comparison of convergence speeds of *D_HPDA* and *DA_HPDA* for *gzip*

ter generalization capabilities of our methods. The false positive rates of our methods are also better than the other approaches, particularly *VtPath*. This confirms the advantages of HPDA model described in section 3.4.4. Moreover, the DA_HPDA has improved performance over D_HPDA. The single entry/exit point assumption helps the method learn a more general model of the program control flow. It is interesting to compare the false positive rates of *VtPath* and *n-gram*. During the initial learning phase, *VtPath* has a lower false positive rate. However, as learning proceeds, *n-gram* has lower false positive rate. This may be due to two factors. First, *VtPath* uses extra return address information for the detection. Two system calls with the same ID appearing as the same by *n-gram* may be treated as different system calls by *VtPath* since the system calls may have different re-

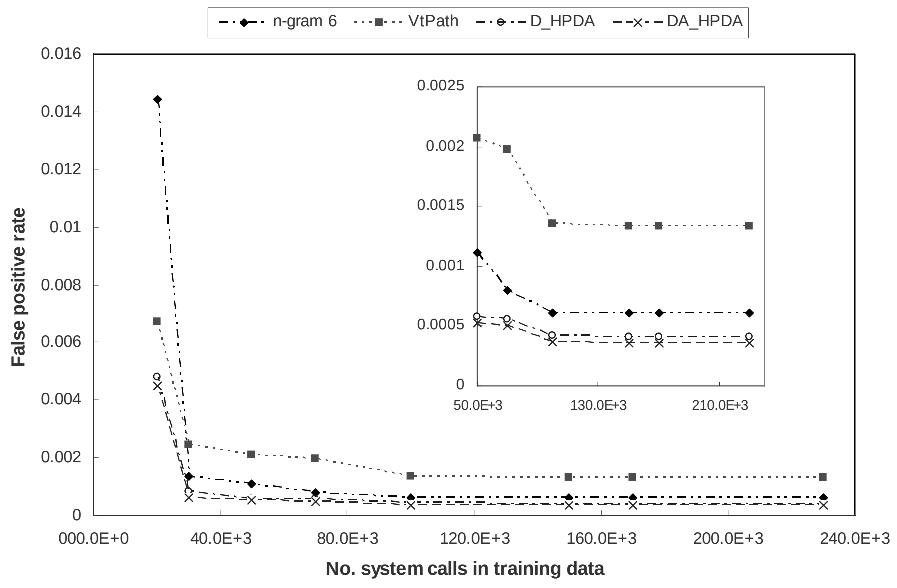


Figure 4.6

Comparison of false positive rates of *D_HPDA* and *DA_HPDA* for *gzip*

turn addresses. The ineffective handling of recursive functions by *VtPath* may be another problem. As described in the original paper, *VtPath* handles recursion by finding a pair of return addresses that are the same and removing all return addresses between the pair from the virtual stack list, including one end of the pair. Ignoring the behavior between two identical addresses prevents *VtPath* from learning some paths that are encountered in the training data. For example, for the stack shown below:

$Original\ stack:\ F\ A\ C\ D\ B\ A\ B\ G\ ...$

$Stack\ after\ removing\ recursion:\ E\ A\ B\ G$

One possible path $F\ A\ C\ D\ B\ G$ is missed.

The results with *httpd* shown in Figure 4.7 and 4.8 give additional evidence of the advantage of our approaches. Both D_HPDA and DA_HPDA have faster convergence speeds and lower false positive rates than the *n-gram* and *VtPath* approaches. Again, DA_HPDA has better performance than D_HPDA.

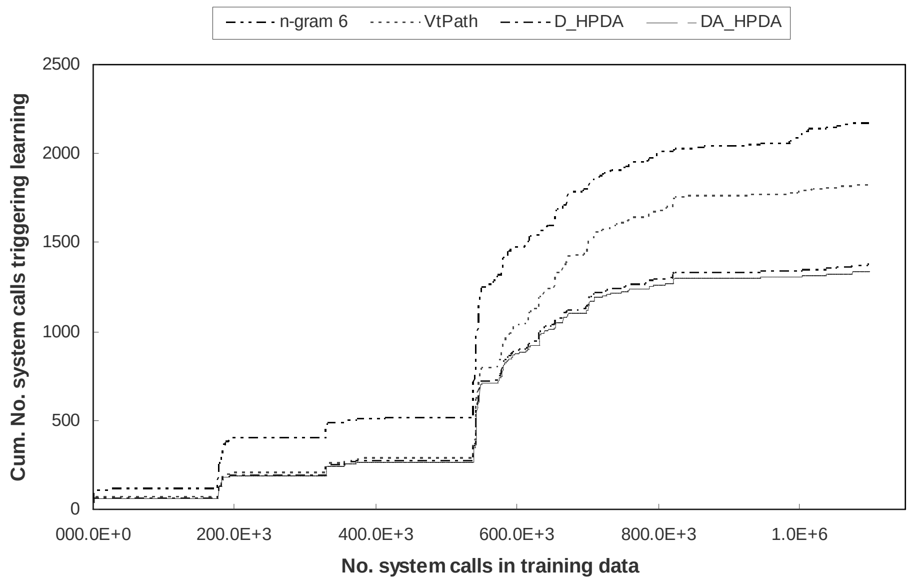


Figure 4.7

Comparison of convergence speeds of *D_HPDA* and *DA_HPDA* for *httpd*

Our approaches also have faster convergence rates and lower false positive rates than *n-gram* with *vsftpd*. For this program, *VtPath* and the HPDA based approaches have similar performance.

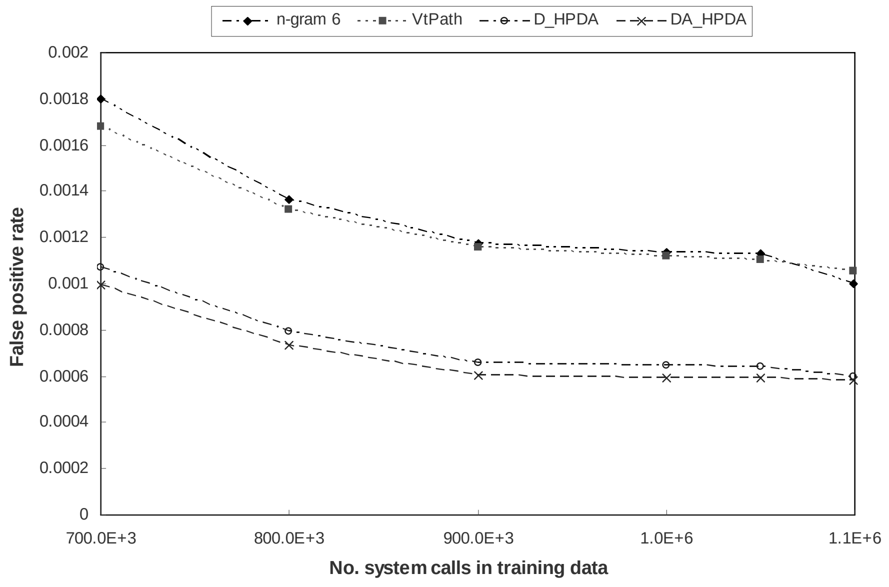


Figure 4.8

Comparison of false positive rates of *D_HPDA* and *DA_HPDA* for *httpd*

4.4.2 Applicability to real-world applications

The experimental results in the previous section demonstrate that the DA_HPDA approach has better generalization of program control flow than all other approaches because it takes advantage of the assumption that every function has a single *entry* and *exit* point. In the learning algorithm shown in section 3.4.3, this assumption also implies that any code belongs to only one function and should not be shared by two or more functions. We assess the applicability of this assumption with real world applications by investigating the number of shared chunks in the executables used in our previous experiments. A chunk is a portion of instructions and shared chunks are pieces of code referred to by more than one function. Shared chunks violate the assumption of the DA_HPDA model. Table 4.3

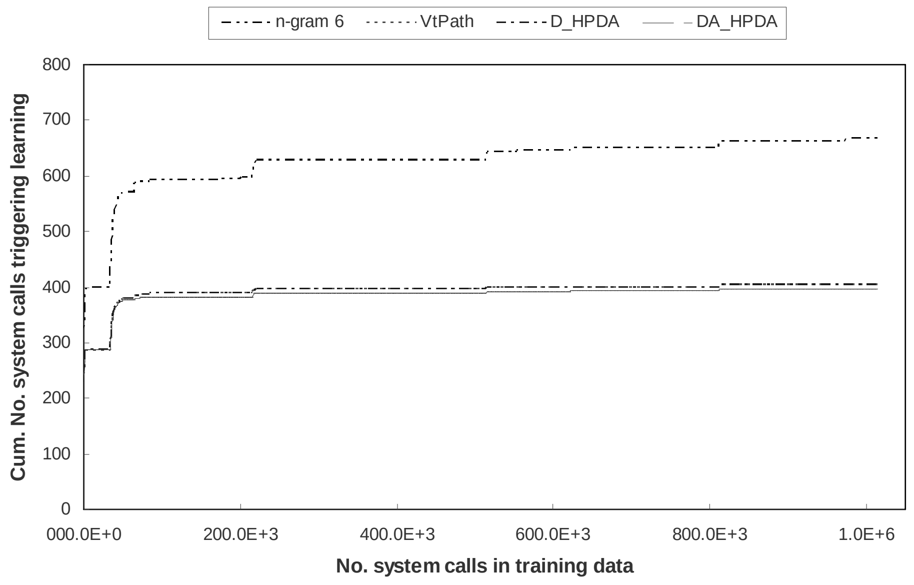


Figure 4.9

Comparison of convergence speeds of *D_HPDA* and *DA_HPDA* for *vsftpd*

shows the total number of functions and number of shared chunks found in the executables used in previous sections. From Table 4.3, we can see that shared chunks seldom appear in these applications and most of the executables and dynamic libraries do not contain shared chunks. Even in the executable that contains shared chunks, the proportion of the shared instructions is very small (less than 1%). Although the DA_HPDA algorithm introduces impossible paths when shared chunks are encountered, the learning process is not affected by the chunks and thus the DA_HPDA algorithm can work with the executable containing shared chunks without any problem. Therefore, we believe that the increased generalization capablity afforded by the DA_HPDA algorithm outweighs the potential for

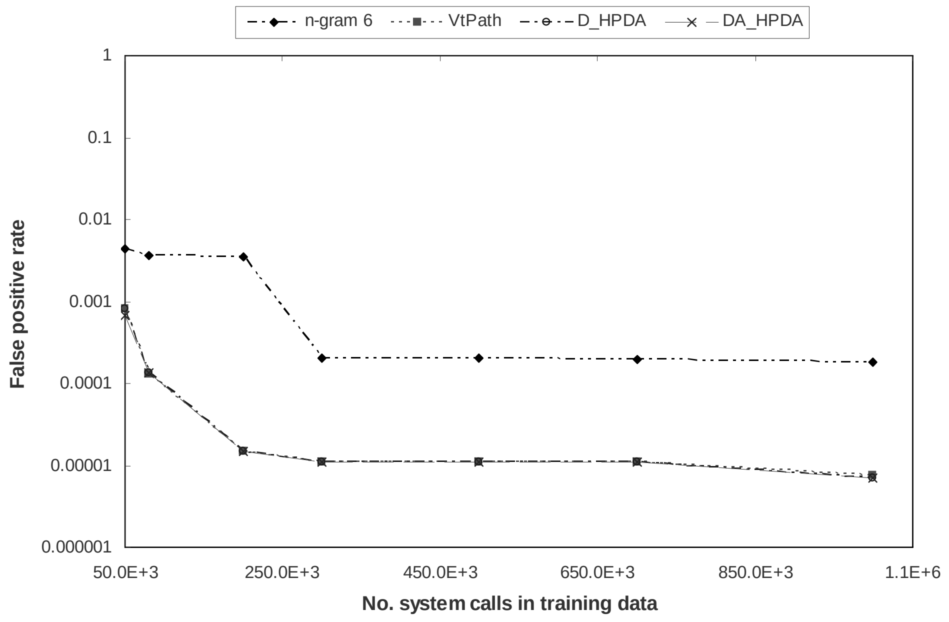


Figure 4.10

Comparison of false positive rates of *D_HPDA* and *DA_HPDA* for *vsftpd* introducing impossible paths that appear to be rarely encountered in real world applications.

Table 4.3

Number of code chunks shared by several functions in several executables

program	#func	#shard	program	#func	#shard
cat	104	0	xv	1856	0
gzip	201	0	ld-2.3.2.so	157	0
httpd	1763	1	libc-2.3.2.so	2694	9
kon	316	2	libpthread-0.10.so	439	0
vsftpd	585	3	libX11.so.6.2	2174	2

4.4.3 Computational overhead for DAHPDA learning

The DA_HPDA algorithm must maintain extra information for each node in the model and every time it modifies the model it must determine if it is necessary to combine the function's *entry* and *exit* points. This introduces substantial additional computational overhead. Table 4.4 presents the learning time of the DA_HPDA learning compared to the learning time of the D_HPDA algorithm for the *httpd* and *gzip* programs. The additional overhead incurred by the DA_HPDA algorithm is the difference in these learning times. The additional overhead for DA_HPDA is about 2.5% for both training data sets. When using DA_HPDA models for detection, the model is transformed into an HPDA and thus does not have any additional overhead for intrusion detection. The additional overhead for learning is relatively low and does not prevent the algorithm from being used with a large real world application.

Table 4.4

Comparison of learning times of DA_HPDA and D_HPDA

(Average of 50 runs, unit: seconds)

	#calls	D_HPDA	DA_HPDA	%Overhead
httpd	1,099,953	43.598	44.719	2.572
gzip	239,874	7.124	7.289	2.325

4.5 Evaluation of the dynamic learning method for CBHPDA

In this section, the CBHPDA approach is evaluated with real world applications.

4.5.1 Motivation for CBHPDA revisited

We have previously shown the proportion of model transitions that come from code in the dynamic libraries used by the *Apache* server in section 3.6. In this section, similar statistics are shown for several additional programs to further motivate the CBHPDA approach. Table 4.5 presents the distribution of transitions that came from the program executable and those that came from behavior in the dynamic libraries for *cat*, *gzip*, and *httpd*. The results are from the models acquired from the training data described in section 4.2. For even a simple program like *cat*, much of the behavior that is learned comes from dynamic libraries rather than the program executable. The training data used to acquire the model for *httpd* that is compiled in Table 4.5 contains more complex behaviors than that used to acquire the *httpd* model in 3.6. The results in Table 4.5, demonstrate that *httpd* uses many dynamic libraries such as the ones that handle PHP and MYSQL database operations. Since the application *PhpGroupware* was used to generate the workload for *httpd*, the library handling PHP accounts for 53.32% of the behavior. For *gzip*, most functions are implemented in the main executable and thus 59.19% of the transitions reside in the main executable.

4.5.2 Acquisition of normal behavior

The dynamic learning algorithm for CBHPDA extends the dynamic learning algorithm for HPDA and separates the transitions for each execution component. The algorithm CBHPDA learns exactly the same behavior as the HPDA algorithm, but it changes the organization of the transitions. Therefore, the performance of the CBHPDA algorithm should be same as HPDA. We have conducted experiments that compare the convergence speeds and false positive rates for the CBHPDA algorithm and compared the performance to that of the D_HPDA algorithm. The experimental results confirm that the performance of the two algorithms with regard to convergence speed and false positive rates are the same.

Since CBHPDA separates the transitions for each execution component, parts of a model learned for one program can be used when building a model of another program that uses the same dynamic libraries. For example, the model for *cat* contains three com-

Table 4.5

Distribution of model transitions

cat		
program	#transition	%
/bin/cat	66	19.24
/lib/ld-2.3.2.so	67	19.53
/lib/libc-2.3.2.so	210	61.23

gzip		
program	#transition	%
/bin/gzip	454	59.19
/lib/ld-2.3.2.so	67	8.74
/lib/libc-2.3.2.so	246	32.07

httpd		
program	#transition	%
/usr/sbin/httpd	419	10.03
/lib/libc-2.3.2.so	356	8.53
/usr/lib/libapr.so.0.0.0	215	5.15
/lib/libpthread-0.10.so	36	0.86
/usr/lib/libaprutil.so.0.0.0	35	0.64
/usr/lib/httpd/modules/mod_log_config.so	8	0.19
/usr/lib/httpd/modules/mod_negotiation.so	45	1.08
/usr/lib/httpd/modules/mod_include.so	34	0.81
/usr/lib/httpd/modules/mod_mime_magic.so	12	0.29
/usr/lib/httpd/modules/mod_python.so	421	10.08
/lib/libdl-2.3.2.so	6	0.14
/lib/ld-2.3.2.so	17	0.4
/lib/libnss_files-2.3.2.so	52	1.24
/usr/lib/httpd/modules/mod_perl.so	16	0.38
/usr/lib/httpd/modules/mod_unique_id.so	2	0.05
/usr/lib/httpd/modules/mod_rewrite.so	4	0.1
/usr/lib/httpd/modules/mod_userdir.so	4	0.1
/usr/lib/httpd/modules/mod_dir.so	2	0.05
/usr/lib/httpd/modules/mod_cgi.so	33	0.79
/usr/lib/httpd/modules/libphp4.so	2227	53.32
/usr/lib/php4/mysql.so	57	1.36
/usr/lib/mysql/libmysqlclient.so.10.0.0	142	3.4
/lib/libnss_dns-2.3.2.so	4	0.1
/lib/libresolv-2.3.2.so	26	0.62
/usr/lib/php4/imap.so	4	0.1

ponents, one for its main executable, one for *libc*, and one for *ld*. Since *httpd* also uses *libc* and *ld*, the models acquired by learning the program *cat* can be reused for building the models for *httpd*. The effect of this type of sharing is shown in figures Figure 4.11 and Figure 4.12. In each case, convergence rates for learning models from "scratch" are compared to convergence rates when models for model components learned for other programs were used as a starting point.

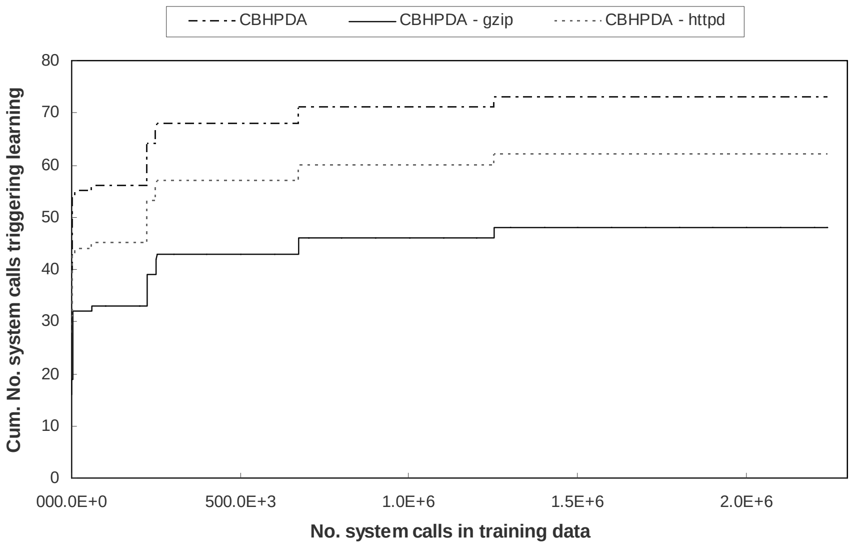


Figure 4.11

Comparison of the convergence speed for CBHPDA for *cat*

In the case of *cat*, use of previous models can substantially reduce learning time. However, with *http*, the reuse did little to reduce learning time because the behavior learned from previous models accounted for a very small portion of the total learned behavior.

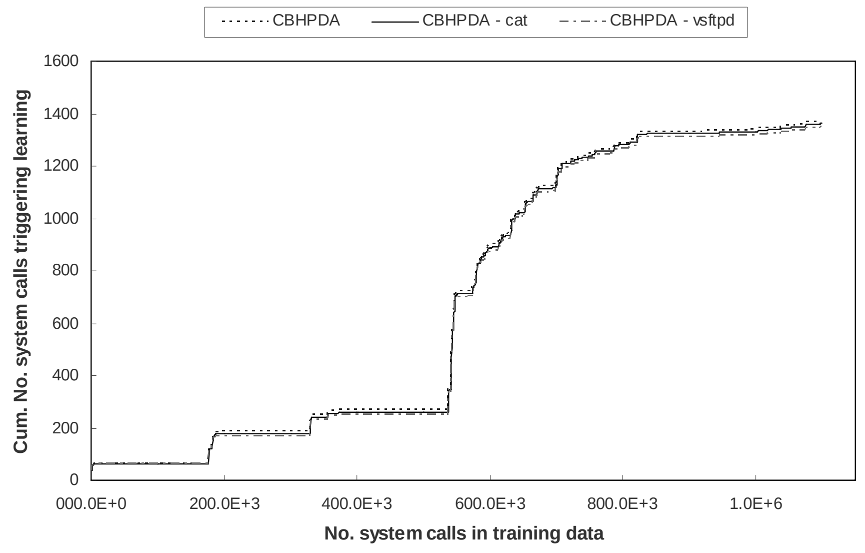


Figure 4.12

Comparison of the convergence speed for CBHPDA for *httpd*

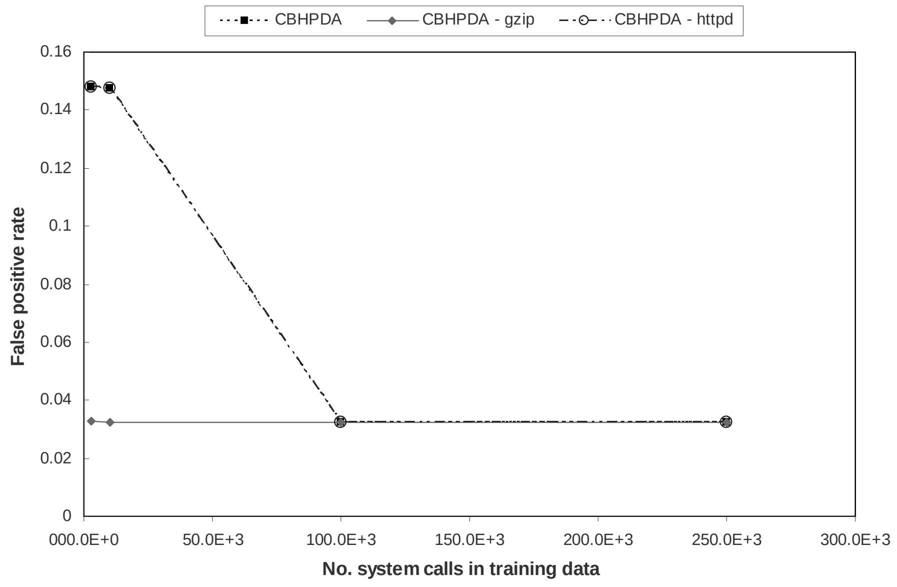


Figure 4.13

Comparison of the false positive rate for CBHPDA for *cat*

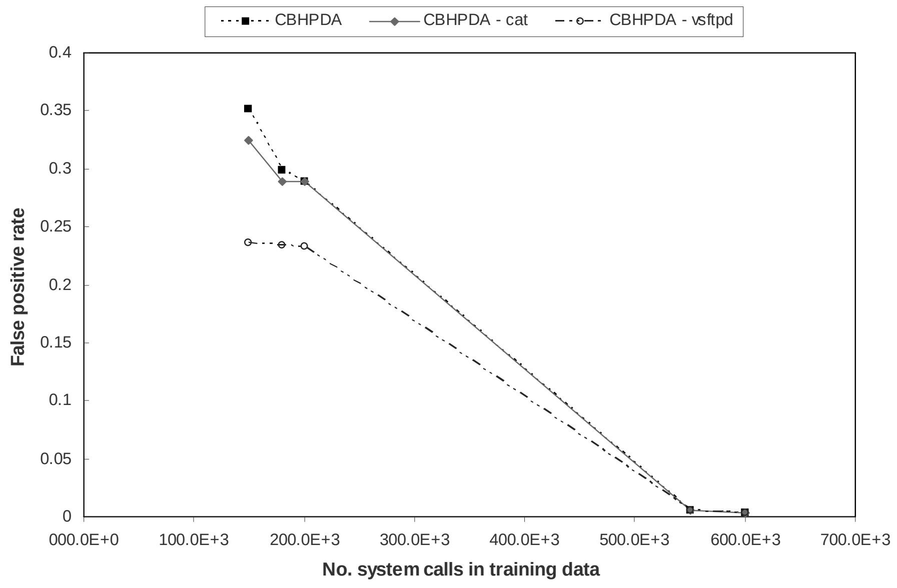


Figure 4.14

Comparison of the false positive rate for CBHPDA for *httpd*

Reuse of previous models reduced false positive rates for both programs. It is clearly more helpful if the models that are being reused are from similar programs. For example, the results presented in Figure 4.11 show that CBHPDA learning of a model for *cat* converges much faster when starting with a model for *gzip* than when starting with a model for *http*. Likewise, results in Figure 4.12 show that CBHPDA learning of a model for *http* converges faster when starting with a model for *vsftpd* than when starting with a model for *cat*. Results for false positive rates demonstrate similar trends. For example, the results in Figure 4.13 demonstrate that the use of base models from *httpd* does not help the learning of program *cat* in terms of false positive rate. However, the models from *gzip* reduces the false positive rate. Results in Figure 4.14 demonstrate that models from both *cat* and *vs*-

ftpd are helpful for developing models for *httpd* and use of both reduces the false positive rates. However, since *vsftpd* shares more components with *httpd* than *cat*, starting with a model for *vsftpd* is more helpful. The program *cat* derives 61% of its behavior from *libc*. Therefore, improvements in learning rates for *cat* models starting from other base models is more noticable than for *httpd* that derives only 8.53% of its behavior from *libc*.

Since the shared components between models contain only a small part of behavior for the programs, the contribution of learning CBHPDA from a base model is only visible when the training data is relatively small. The number of system calls in training data used to generate the results in Figure 4.13 and Figure 4.14 is substantially less than that used for analysis of the performance of DA_HPDA reported in previous section. When the size of the training data becomes larger, the advantages of reuse facilitated by CBHPDA tend to become much less significant.

4.5.3 Computational overhead

In this section, we measure the computational overhead of the dynamic learning algorithm for CBHPDA models. The experimental design is the same as described in section 4.4.3. Table 4.6 compares the learning time for CBHPDA compared with that of D_HPDA. The times are essentially the same and the overhead introduced to handle different cases of transitions between components with CBHPDA is negligible.

Table 4.6

Comparison of learning times of D_HPDA and CBHPDA

(Average of 50 runs, unit: seconds)

	#calls	D_HPDA	CBHPDA	%Overhead
httpd	1,099,953	43.598	43.307	-0.67
gzip	239,874	7.124	7.118	-0.08

4.6 Combination of static analysis and dynamic learning

Previous approaches for building models of program behavior based on system calls have used either static analysis or a learning approach. Our HPDA model is unique because it can be acquired by learning from audit data or by a combination of static analysis and learning. Static analysis of the program executable can be used to obtain a large proportion of program behavior. Application of dynamic learning algorithm to the base models acquired by static analysis can be used to learn transitions that come from behavior that cannot be known until runtime. The combination approach results in faster convergence and lower false positive than when learning from audit data from "scratch" because the base models already contain much of the behavior. In this section, the performance improvement using this combination method is measured and compared with the results from learning the models *de novo* using dynamic learning.

Figure 4.15, 4.17, and 4.19 compare the convergence speeds of the combination approach with that of the dynamic learning approach for *gzip*, *httpd*, and *vsftpd*. Figure 4.16, 4.18, and 4.20 present a comparison of the false positive rates for these three programs. In every case, the combination method results in a faster convergence speed and a lower false positive rate than dynamic learning alone.

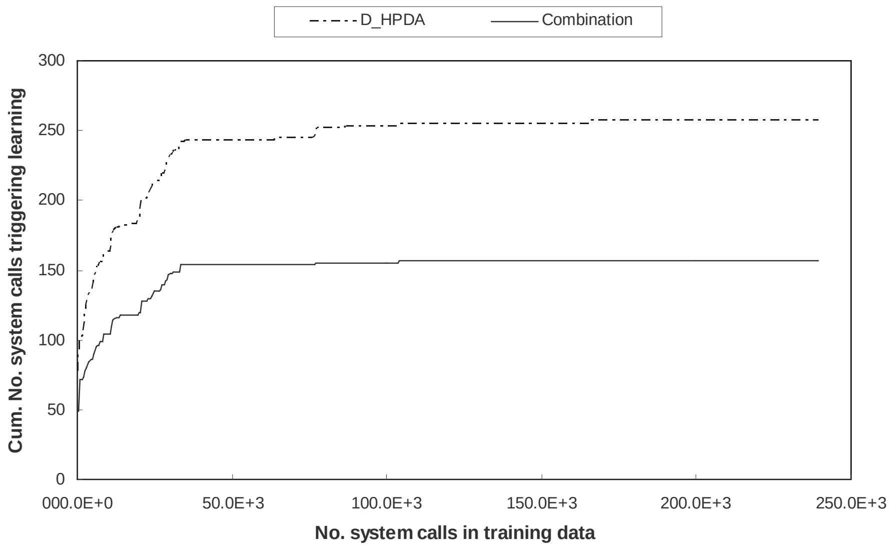


Figure 4.15

Convergence speeds of dynamic learning and combination method (*gzip*)

Although, in our experiments, the combination method always exhibited faster convergence speeds and lower false positive rates than dynamic learning alone, we were surprised by the large number of new transitions that were still acquired by dynamic learning. Further analysis indicates that may be due in part to the immature implementation of our static analysis method. Our current implementation of static analysis does not acquire transitions for indirect calls. Table 4.7 gives a summary of the number of direct and indirect calls used by several executables. Normal calls are those where the target of the call site is known.

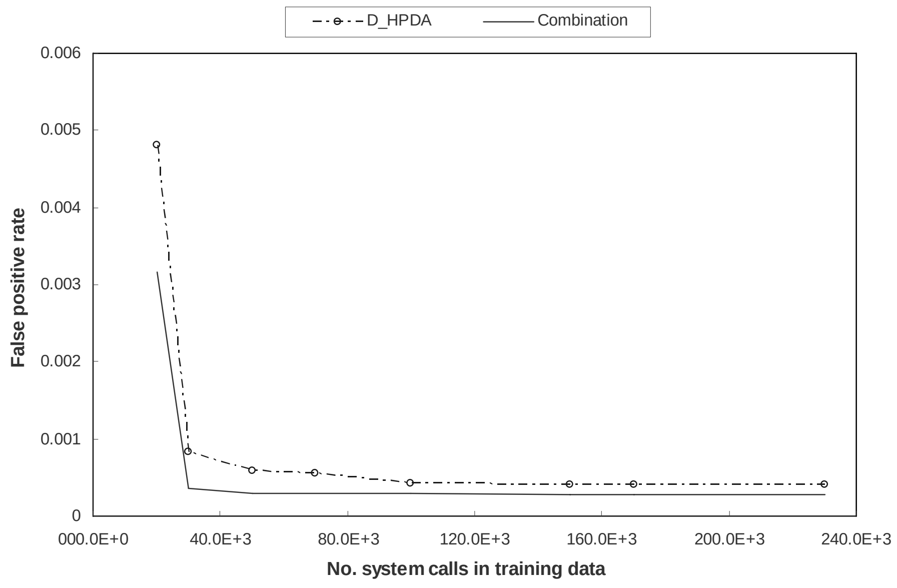


Figure 4.16

False positive rates of dynamic learning and combination method (*gzip*)

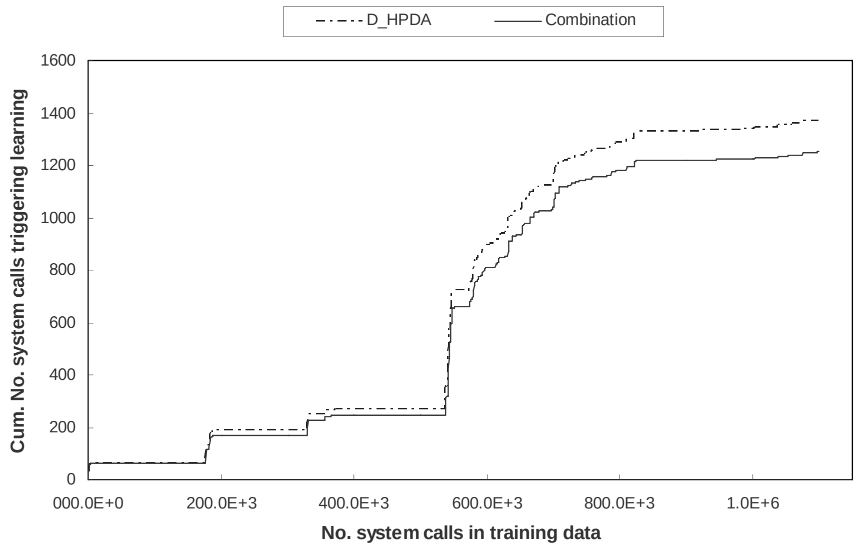


Figure 4.17

Convergence speeds of dynamic learning and combination method (*httpd*)

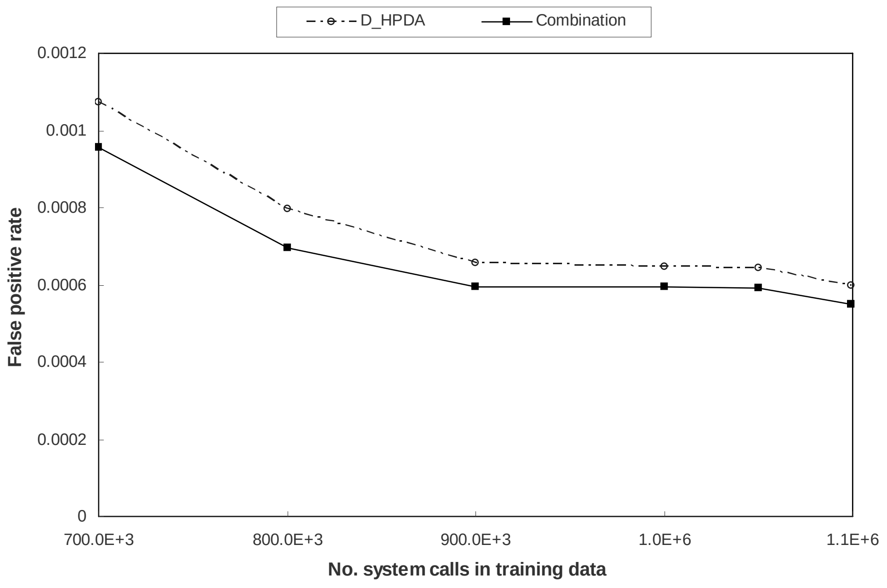


Figure 4.18

False positive rates of dynamic learning and combination method (*httpd*)

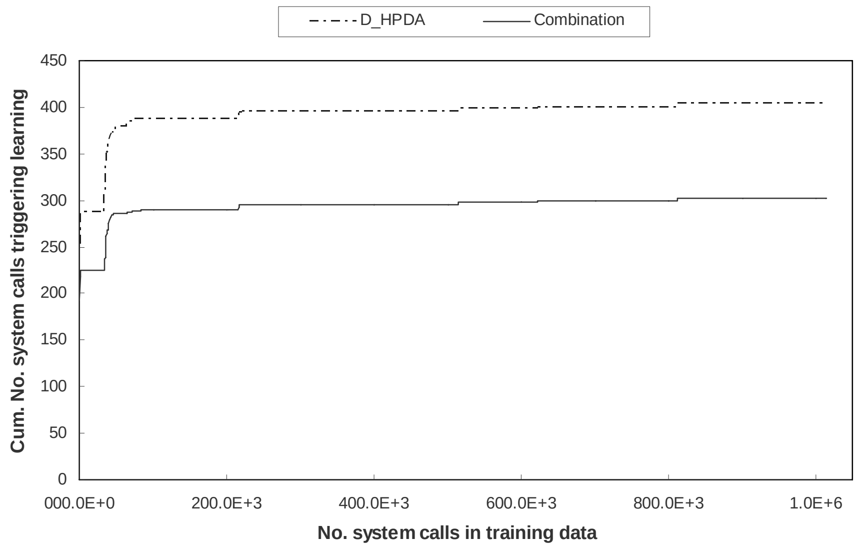


Figure 4.19

Convergence speeds of dynamic learning and combination method (*vsftpd*)

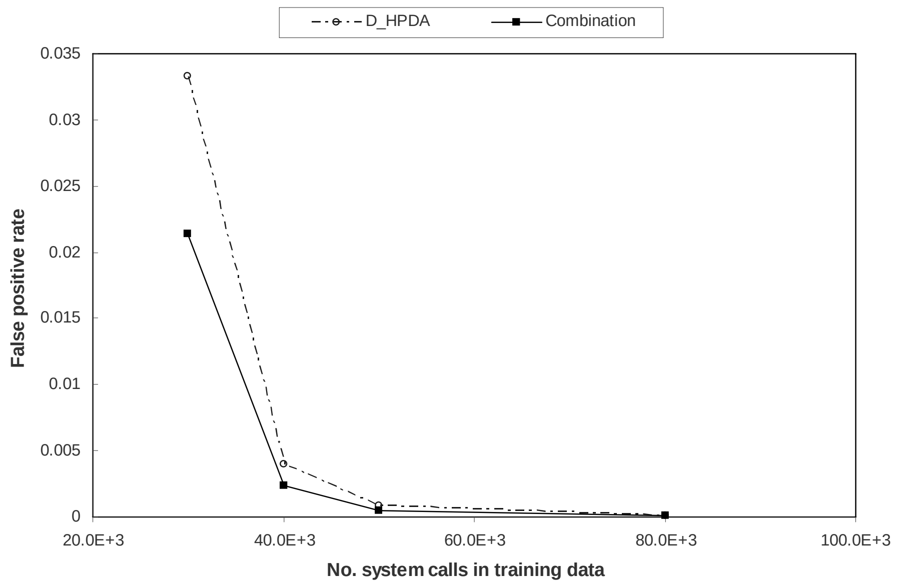


Figure 4.20

False positive rates of dynamic learning and combination method (*vsftpd*)

Indirect calls are call sites where the function is called through a function pointer instead of a direct address. Unhandled calls are call sites that our implementation of static analysis cannot handle for some reason. These can be caused by calls to a relative address or incomplete address information provided by the disassembler. The data in Table 4.7 shows that for the programs analyzed, indirect calls and unhandled calls account for a substantial proportion of the calls. In the future, a pointer analysis algorithm can be used to recover indirect calls. Other researchers have shown that a good pointer analysis algorithm can recover most indirect calls (70% or 80%) [42].

Table 4.7

Number of indirect calls in several executables

executable	normal call	indirect call	unhandled call
httpd	2231	97	3
gzip	484	9	3
vsftpd	1500	8	3
libc-2.3.2.so	10788	2070	193
ld-2.3.2.so	684	17	7

4.7 Performance penalty of the HPDA-based detection system

If an intrusion detection system is to be used for real-time detection, it must be efficient. In this section, we measure the performance penalty introduced by our approaches. We perform a detailed analysis of the overhead for monitoring *cat*, *gzip* and *Apache*. This includes measurements of the the overhead associated with the steps in the call stack walk process and measurements of the overhead of different aspects of the detection process. This analysis is substantially more detailed than that presented by other researchers who use related methods [31, 32, 37, 71]. The runtime for *cat* was measured by concatenating several files into a new file. We used the set of files from the compression benchmark and one large iso file (disk image file) in this measurement. Runtime measurements for *gzip* were collected by compressing the files described above. The measurement for *Apache* were generated by *WebBench* as described in section 4.2. All the measurements were calculated as the average of 10 runs.

4.7.1 Collection of call stack information

Researchers [31, 32, 37, 71] have discovered that return addresses on the program call stack provide a useful resource for intrusion detection and that detection approaches based on analysis of this information are more difficult for attackers to evade. A common process used by all of these approaches is extraction of return addresses from the program call stack. Although this step is the key part for all approaches that use call stack information, other researchers have not reported a detailed analysis of the overhead associated with this step. In this section we evaluate the performance overhead of the process of collecting call stack information. There are two steps involved in obtaining the return addresses from the call stack. One is the extraction of the absolute return address from the program call stack and the other is the transformation from absolute addresses to paired addresses. Measurements of the overhead introduced by the call stack walk are presented in section 4.7.1.1 and measurements of the overhead introduced by address transformation are presented in section 4.7.1.2.

4.7.1.1 Overhead associated with the call stack walk

Return addresses are extracted by walking the program call stack. As shown in section 2.1.2, all activation records are connected by the *EBP* pointer. Extraction of call stack information occurs when any system call is invoked. The process of the extraction includes backtracking the program stack using the *EBP* pointer and fetching the IP values (return addresses). During the process of backtracking the program stack, all the values that are inspected are in user space memory. A reference to user space memory from kernel mode requires checking to determine if the page with the referenced value is loaded in memory or has been swapped to virtual memory. The Linux kernel provides several mechanisms for referencing user space memory from kernel mode and all include complicated checking procedures.

In this section, measurements of the overhead introduced by walking the program call stack are presented. The base runtime shown in following tables is the runtime collected without any part of the detection system loaded. This runtime was collected using the patched kernel. Since the patch adds fewer than 30 lines of assembly code for system call invocation and has only three more checks than an un-patched kernel, the overhead introduced by the patch is ignored. The *csw (Call Stack Walk)* items in the following tables represent the runtime of the program with our module loaded, walking the program call stack, and collecting the absolute return addresses.

Table 4.8 shows the call stack walk overhead for *cat*, *gzip*, and *gunzip*. The *real* runtime values are not equal to the sum of the *user* and *sys* times because *real* also includes time spent on I/O operations. The sum of *user* and *sys* time is far less than the *real* time for *cat*. This is reasonable because most of execution time for *cat* is spent in I/O operations. Although the impact of our system is significant for the simplest program *cat* (greater than 30%), the overhead for the programs involving more computation is less than 5%.

The call stack overhead for the *Apache* server is presented in Table 4.9. *Requests per second* and *throughput* are two critical measurements for web servers. *Requests per sec-*

Table 4.8

Overhead introduced by call stack walk

(unit: seconds)

		real	user	sys
cat	base	15.228	0.041	1.38
	csw	15.155	0.055	1.894
	overhead	-0.4%	34%	37%
gzip	base	38.3245	36.162	1.3605
	csw	38	36.1965	1.4425
	overhead	-0.8%	0.09%	6%
gunzip	base	6.183	4.659	1.064
	csw	6.28	4.85	1.062
	overhead	1.6%	4%	-0.1%

Table 4.9

Overhead introduced by call stack walk for *WebBench*

	Requests Per Second	Throughput (Bytes/Sec)
base	257.404	1,570,829.250
csw	256.354	1,571,205.625
overhead	0.4%	-0.02%

ond represents the ability of the server to handle concurrent requests and the *throughput* demonstrates the ability of the server to handle large requests. The call stack walk introduces little overhead for the *Apache* server. In Table 4.9, the throughput for the system with call stack walking enabled was greater than that of the base system resulting in negative overheads in terms of the *real* time. The overhead associated with the call stack walk appears to be negligible compared to the impact of other factors such as the workload in the machine and cache misses with various http requests.

4.7.1.2 Overhead associated with address transformation

Address transformation is another key step in the process of collecting information from the call stack log. As discussed in section 3.1.1, the paired address is required to make the systems based on analysis of return addresses applicable in many execution environments. Our system transforms the absolute address to the paired address (HPDA address) as described in section 4.1. The maintenance of the data structure for transformation is activated whenever a change of the program memory mapping occurs. This happens when a *mmap* or *munmap* system call is invoked.

Although overheads shown in Table 4.10 for *real* and *user* time are still very small, the overhead for *sys* time is much more noticable than the overhead with only call stack walking enabled. The computation in the kernel module increases the *sys* time spent by all programs. Table 4.11 shows the results from the *WebBench*. The overall overheads for

Table 4.10

Overhead of address transformation

(unit: seconds)

		real	user	sys
cat	base	15.228	0.041	1.38
	at	15.47	0.046	1.944
	overhead	1.5%	12%	40%
gzip	base	38.3245	36.162	1.3605
	at	38.1185	36.217	1.568
	overhead	-0.5%	0.15%	15%
gunzip	base	6.183	4.659	1.064
	at	6.246	4.7535	1.149
	overhead	1%	2%	7.9%

Table 4.11

Overhead of address transformation for *WebBench*

	Requests Per Second	Throughput (Bytes/Sec)
base	257.404	1,570,829.250
at	251.058	1,514,974.875
overhead	2.46%	3.56%

both *request per second* and *throughput* are less than 4 percent. Overheads of less than 5% are typically considered acceptable.

Our analysis shows that, overall, the process of collecting call stack information introduces acceptable overhead. Although stack walking and address transformation are common steps in all approaches that use the return addresses from the call stack [31,32,37,71], other researchers have not reported the overhead associated with this process. In this section, we have demonstrated that the collection of call stack information introduces little overhead and can be applied to real world programs. This analysis applies to all approaches [31, 32, 37] based on analysis of return addresses from the call stack.

4.7.2 Detection overhead

In this section, we examine the performance overheads associated with our detection system. These include the overhead associated with capturing log data, the overhead of learning (both *de novo* learning and combination learning), and the overhead of detection using the learned model. As described in section 4.1, the HPDA IDS kernel module is used in several different phases of operation of the system. It implements the functions that acquire program models using dynamic learning, those that write the call stack log to a file, and those that perform detection. The learning component collects call stack information and extracts models from a log collected at runtime. The models can be learned *de novo* from the audit logs or learning can begin with a base model acquired by static analysis. The logging component writes the call stack information to a log file. The detection component collects the call stack information in memory and inspects the call stacks using the HPDA models. It includes the call stack walking and address transformations operations described in the previous section. We also investigate the advantage of a kernel implementation over a user space implementation by comparing the overhead of a kernel approach with that of a user space approach.

Table 4.12 presents the overhead measurements for learning and detection of CBHPDA models for *cat*, *gzip*, and *gunzip* and Table 4.13 gives the overhead of the CBHPDA approach for *WebBench*. Surprisingly, in most cases the overhead of the learning component is less than the overhead of the detection component. This result is unexpected because the learning component uses the same operations as the detection component and, in addition, it adds new transitions to the model if a matched one cannot be found. Since the values for the *user* and *sys* times are very small, any variation in the execution environment can lead to a significant differences in the results.

The overheads presented in the *sys* time are larger than others. This is evidence that the kernel based system influences the *sys* time. All of the overheads for *real* time of the learning and detection components are less than 4 percent. The exception is the overhead for *gunzip* for detection at 8.27 percent. The results from Table 4.13 show that the overhead of the learning and detection components are less than 9 percent. In this case, the overhead of the learning component is higher than that of the detection component. The learning component has an overhead that is on average 2% more than the overhead for detection.

Table 4.12

Overhead of learning and detection using the CBHPDA approach

(unit: seconds)

Performance for program *cat*					
	base	learning	%	detection	%
real	15.228	15.346	0.77	15.178	-0.32
user	0.041	0.048	17.07	0.054	31.71
sys	1.38	2.276	64.93	2.246	62.75

Performance for program *gzip*							
	base	learning	%	detection	%	us-detection	%
real	33.673	33.613	-0.18	33.898	0.67	74.428	121.03
user	31.75	31.788	0.12	31.778	0.09	32.485	2.31
sys	1.283	1.53	19.29	1.763	37.43	3.12	143.27

Performance for program *gunzip*							
	base	learning	%	detection	%	us-detection	%
real	5.47	5.685	3.93	5.923	8.27	12.695	132.08
user	4.055	4.078	0.55	4.035	0.49	4.113	1.41
sys	0.95	1.063	11.84	1.163	22.37	1.275	34.21

Table 4.13

Overhead of learning and detection of CBHPDA approach with *WebBench*

	Requests Per Second	Throughput (Bytes/Sec)
Base	257.404	1,570,829.250
learning	239.367	1,432,476.750
%	7	8.81
detection	243.971	1,465,842.500
%	5.22	6.68
logging	219.188	1,309,306.125
%	14.85	16.65

The user space detection system was only tested with *gzip* and *gunzip*, because the frequency of system call invocation of the program *cat* and *httpd* is so frequent that it overflowed the buffer for transmitting information from kernel space to user space. Since our focus is the kernel space implementation, we did not try to improve the performance of our user space implementation so that it could handle *cat* and *httpd*. From Table 4.12, we can see that the user space detection system introduced considerable overhead. When detection is conducted in user space, context switching between the monitored program and the detection program consumed large amounts of time and thus the *real* time overheads are more than 100%. Since the detection system does not change the execution of the monitored program, the *user* time overheads are similiar to others. Table 4.13 shows the overhead of our system when logging the data into files. Since logging involved extensive I/O operations, the overhead is greater than for other operations.

Table 4.14

Comparison of the performance of CBHPDA model and similar models

	% Runtime Overhead		
Program	Dyck	VPStatic	IAM
cat	56	32	0
htzipd	135	97	8.3

(a) Overhead of similar models [43]

	% Runtime Overhead	
Program	learning	detection
cat	0.77	0
httpd	9.65	7.16

(b) Overhead of CBHPDA models

Since the *real* time represents the users' perception of the speed of an application, the experiments conducted in papers [31, 43] use *real* time to compute the overhead. Table 4.14 compares the results reported in the paper by Gopalakrishna et al. [43] with our approach. The overhead in the table is the *real* time overhead. Gopalakrishna et al. worked with a simple web server (*htzipd*) rather than an industry standard web server like the *Apache* server (*httpd*) that we used in our experiments. We believe that the results with *htzipd* should be comparable to our results with *Apache*. In our experiments with *httpd*, we used throughput as a measure of performance but Gopalakrishna reported results for *htzipd* using runtime as a measure of performance. We compute the overhead for our system as shown below.

$$R_b = \frac{S}{T_b} \quad and \quad R_m = \frac{S}{T_m}$$

$$\%Overhead = \frac{R_m - R_b}{R_b} = \frac{S/T_m - S/T_b}{S/T_b} = \frac{1/T_m - 1/T_b}{1/T_b} = T_b/T_m - 1$$

$where\ R_b\ is\ the\ runtime\ of\ the\ base\ system, R_m\ is\ the\ runtime\ with\ the\ detection\ system\ running, T_b\ is\ the\ throughput\ of\ the\ base\ system, T_m\ is\ the\ throughput\ with\ the\ detection\ system\ running, S\ is\ the\ size\ of\ data$

The results in Table 4.14 demonstrate that our system has much better performance than *Dyck* and *VPStatic*. These two methods are very similar to ours and intrusion detection is activated at the system call invocation point in all of these approaches. The *IAM* approach is based on monitoring the library calls. Our system also has a better performance than *IAM*.

4.8 Effect of epsilon reduction

The epsilon reduction algorithm introduced in section 3.3.2 can be used to reduce the number of epsilon transitions in the models learned by static analysis or dynamic learning method. As mentioned in section 3.3.2, although it is possible to remove all ϵ transitions, some ϵ transitions are kept in our method to avoid violation of properties of the HPDA. Since some ϵ transitions are useful for the dynamic learning algorithm, a new epsilon reduction algorithm was designed to reduce the number of useless ϵ transitions. All models used in previous results were for models without ϵ reduction. In this section, we investigate the number of ϵ transitions in several program models and demonstrate that epsilon reduction does not affect the computational performance of the learning and detection algorithms for the HPDA.

Figure 4.21 presents the percentage of learned transitions that are ϵ transitions in several CBHPDA models acquired by dynamic learning. In the figure, the labels with a program name followed by an ϵ sign indicate a model from which the useless ϵ transitions have been removed using the epsilon reduction algorithm. Models of simpler programs contain a smaller proportion of ϵ transitions than models of more complex programs. For example, in Figure 4.21 the proportion of ϵ transitions in the model of *cat* is less than 20%, while in the model of *httpd*, ϵ transitions account for almost 35% of all transitions. After reduction, all models used in our experiments have less than 5% ϵ transitions.

Figure 4.22 also shows the proportion of ϵ transitions for models acquired by the combination approach. For these experiments, component models were obtained for all of

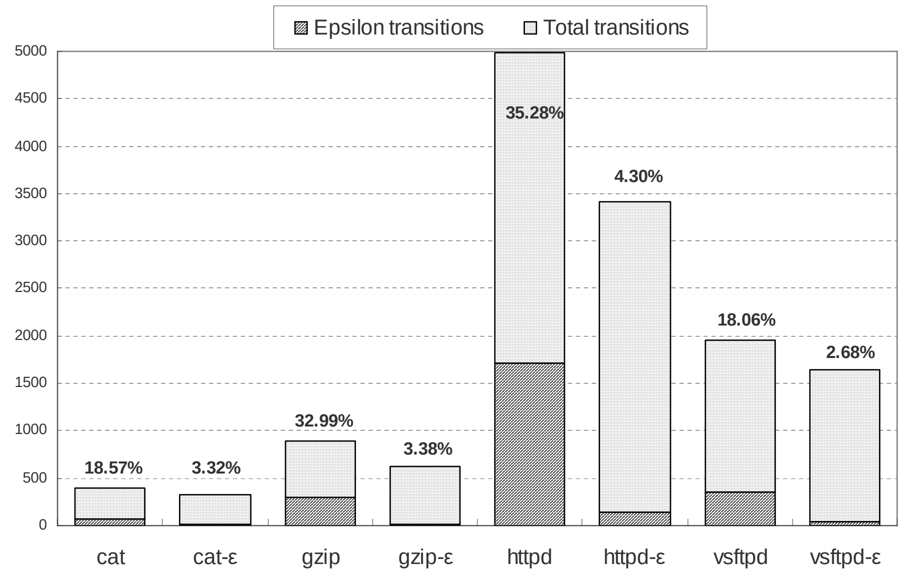


Figure 4.21

Proportion of ϵ transitions — CBHPDA

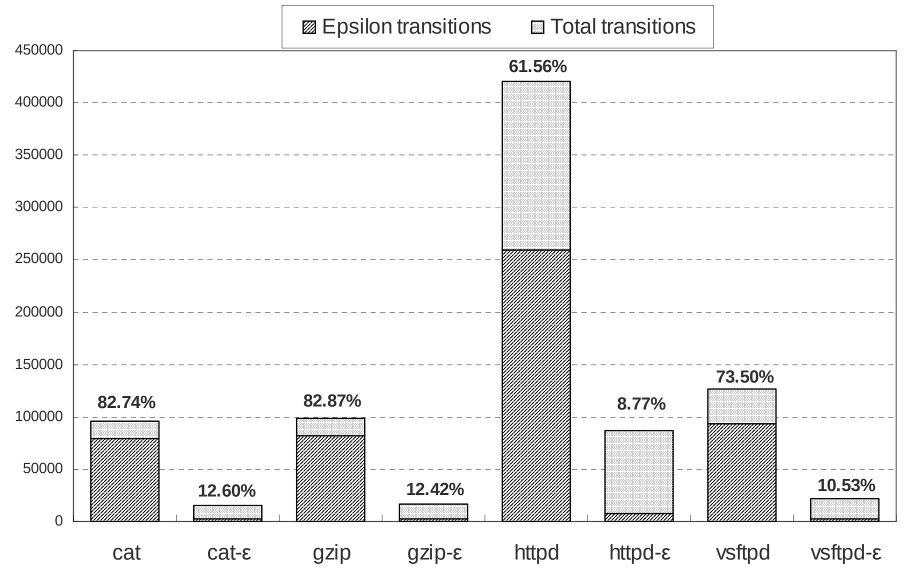


Figure 4.22

Proportion of ϵ transitions — combination approach the programs and then all component models were shared to construct the model for the program. For example, the model for *cat* was constructed from three models, *cat*, *libc.2.3.2.so*, and *ld.2.3.2.so*. The data in the figures is the sum of all component models used for each program. In Figure 4.22, the proportion of ϵ transitions for *cat* is larger than 10%. This is because much of behavior in components models *libc.2.3.2.so* and *ld.2.3.2.so* is not used in *cat* and the models for these libraries contain many ϵ transitions. The model for the *cat* executable contains only 6 ϵ transitions of the 295 total transitions. This also illustrates that simpler program has fewer ϵ transitions. Figure 4.22 shows that the combination approach generates models that have a larger number of ϵ transitions due to features of the static analysis algorithm for components. The number of ϵ transitions removed by the epsilon reduction algorithm can also be seen from this figure.

Because of property 3 of the HPDA, a node cannot have two out transitions with identical symbols. Therefore, the only non-determinism of the HPDA model is introduced by ϵ transitions. In the learning and detection algorithms, ϵ transitions contribute to the complexity of a breadth first search when the algorithm matches an input symbol. Figure 4.23 and Figure 4.24 show the distribution of nodes with different numbers of ϵ transitions. From Figure 4.23, we can see that most nodes in all models do not have ϵ transition. In all models, few if any nodes have more than 2 ϵ transitions. Therefore, we believe that the remaining ϵ transitions have little effect on the breadth first search and have a negligible affect on performance of the learning and detection algorithms.

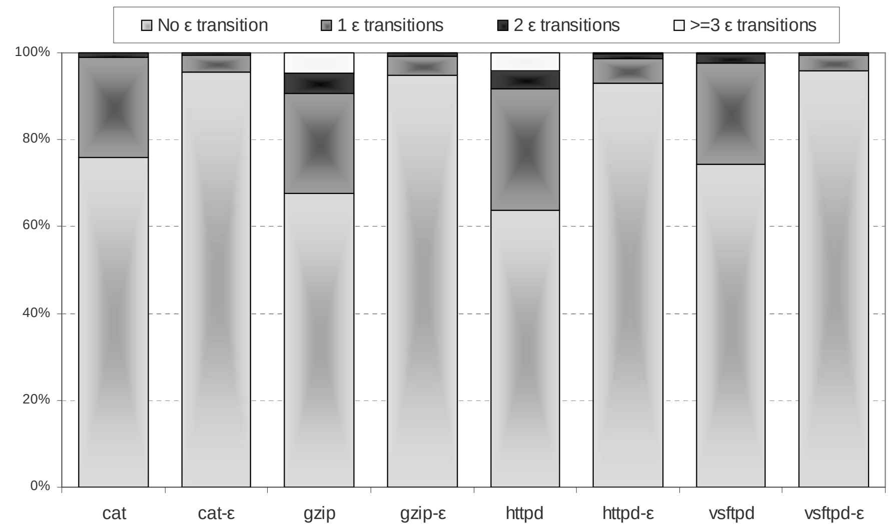


Figure 4.23

Distribution of the number of ϵ out transitions per node — CBHPDA

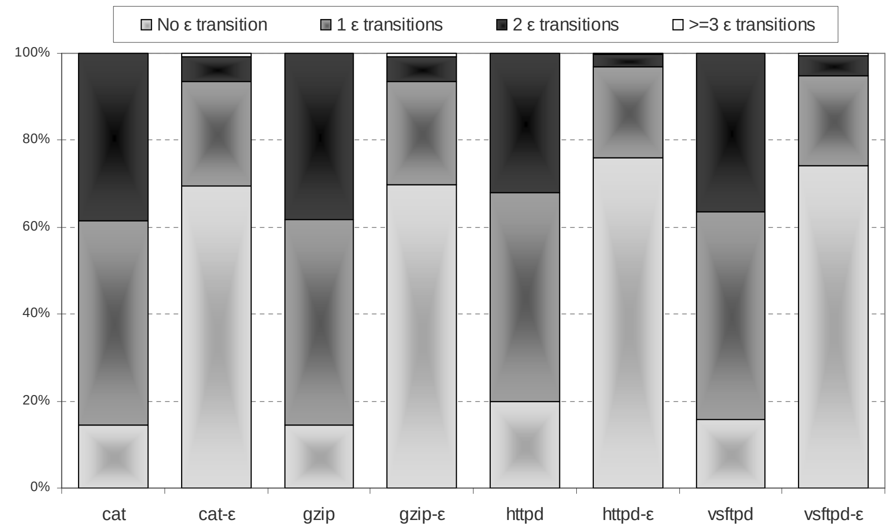


Figure 4.24

Distribution of the number of ϵ out transitions per node — combination

Figure 4.24 shows the distribution of the number of ϵ out transitions per node for models learned by the combination approach. Fewer than 20% of the nodes in models before ϵ reduction was applied contain no ϵ out transitions. Many nodes have 2 or more ϵ out transitions. This large number of useless ϵ transitions resulting from static analysis will slow down subsequent application of dynamic learning and will also slow detection. After epsilon reduction, around 70% of the nodes do not contain ϵ out transitions and less than 6% contain more than two ϵ transitions. The performance evaluation shown in section 4.7.2 used models acquired by dynamic learning and without epsilon reduction. From Figure 4.23 and Figure 4.24, we can see the models acquired by dynamic learning and before epsilon reduction have a larger percentage of ϵ transitions than the models acquired by combination approach and after epsilon reduction. The performance measurements presented in the previous section were done with CBHDPA models without epsilon reduction. We predict that models learned by the combination approach and with epsilon reduction will have a similar or better performance than the data shown in previous section.

One more factor that affects the complexity of breadth first search is the average number of out transitions for each node. Figure 4.25 shows the average number of transitions per node for several models. The data *cb_before* is collected from models acquired with the combination approach with no epsilon reduction. The data *cb_after* was collected from the models acquired by the combination approach with epsilon reduction. The average number of transitions for the models learned by the combination approach and after epsilon reduction is around 3.2 transition per node. The computation of search within this number of transitions should contribute little to the monitoring overhead.

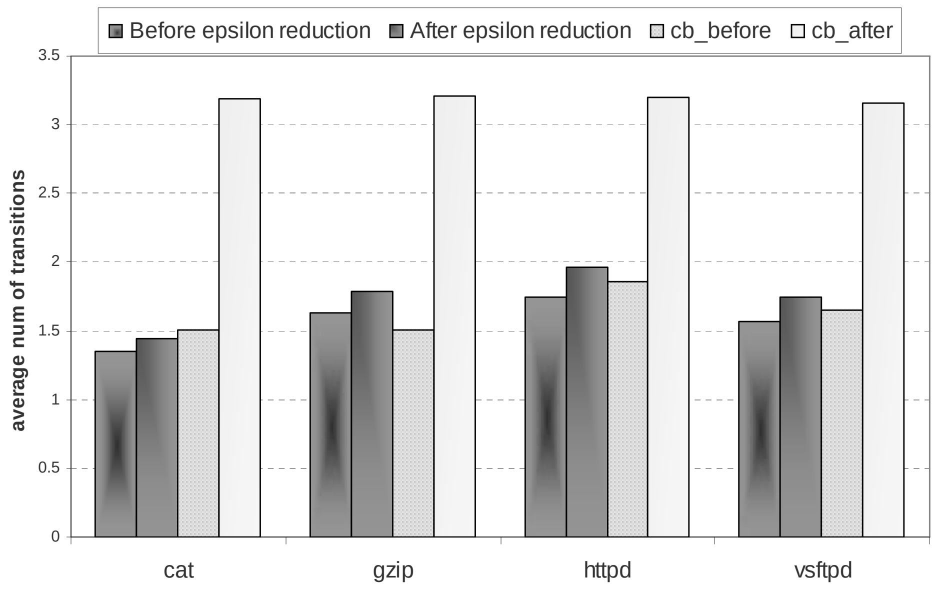


Figure 4.25

Average number of transitions per node

CHAPTER V

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the contributions of the HPDA family of models and associated algorithms that we have developed. We also summarize the experimental results and compare these results to those reported by other researchers for similar models. The limitations of our current implementation are described and direction for future extensions of the research are discussed.

5.1 Summary and contributions

This dissertation presents a new family of models (HPDA and CBHPDA) that capture the program execution flow and can be acquired by dynamic learning or by a combination of static analysis and dynamic learning. The models effectively capture the program execution flow and thus result in a lower false positive rate than other models that do not take special consideration of control flow. The combination approach further reduces the false positive rate by construction of a detailed base model using static analysis that is supplemented by dynamic learning from audit data. Moreover, the static analysis algorithm in our combination approach uses no approximations and thus avoids the introduction of impossible execution paths introduced by the approximations used by other approaches based solely on static analysis. This makes it much more difficult for attackers to evade detection based on models acquired by our combination approach.

The HPDA model is an automata based model for profiling program behaviors utilizing the return addresses and system call information collected from the program call stack. Several properties of the HPDA model are based on the one-to-one correspondence between addresses in the HPDA model and instructions in the corresponding program. These properties are the basis of the dynamic learning algorithm and the anomaly recovery algorithm for the HPDA model. This dissertation has defined the concepts of *execution block* and *execution region* that are subsequently used to construct a mapping between HPDA models and program control flow. These concepts are also used as the basis of the epsilon reduction and dynamic learning algorithms.

The automata representation of program behavior effectively captures loops and recursion in program control flows and thus the dynamically learned HPDA model has a lower false positive rate than the models such as *n-gram* and *VtPath* that cannot naturally capture this type of control flow. Two dynamic learning algorithms, the D_HPDA and DA_HPDA, have been developed to acquire HPDA models from audit logs. The D_HPDA algorithm constructs a model that exactly matches the control flow contained in the audit data and the DA_HPDA algorithm performs further generalization by assuming that every function has a single *entry* and *exit* point and that any code in a program is part of one and only one function. Analysis and experimental results have shown that the D_HPDA approach has a faster convergence speed and a lower false positive rate than *n-gram* and *VtPath*. The

DA_HPDA approach further improves the convergence speed and false positive rates of D_HPDA. An investigation of real-world applications such as *Apache* and *vsftpd* reveals that very few violations of the assumption used in DA_HPDA approach occur in these applications.

A new anomaly recovery mechanism has been designed based on the properties of the HPDA model. All dynamically learned program profiles capturing execution context may result in consecutive alarms due to one anomaly that appears in the test sequence. For example, in the *n-gram* method an anomalous system call invocation may cause several alarms in the sequences containing this system call and the number of sequences affected by this system call depends on the window size used in the method. Our anomaly recovery algorithm is used to rematch the simulated automata and the execution when the automata is in an anomalous state so that the detection method will not generate consecutive false alarms.

This dissertation introduces the first model that can be acquired by a combination of static analysis and dynamic learning. In the combination approach a base model is acquired by static analysis of the program executable and the dynamic learning method is applied on the base model to obtain the behavior that is difficult or impossible to be learned by static analysis. Experimental results demonstrate that the combination approach has faster convergence and a lower false positive rate than methods that depend solely on dynamic learning. We demonstrate attacks that can be designed to exploit the approximations used in other static analysis methods to evade the detection but that cannot circumvent detection by models acquired using the combination approach that uses no approximations.

This dissertation presents the first algorithm that can acquire separate models for each execution component including dynamic libraries and an algorithm for combining these models for detection. In most current applications, the main executable of a program only contain a small portion of program behavior and the execution of the program depends on several dynamic libraries that often contains most of the program behavior. The CBH-PDA model is the first model that defines a model for each execution components (main executable and dynamic libraries) and thus the models for dynamic libraries can be shared by the detection system to build detection models for several programs. This framework reduces the memory usage of the detection system since the models are shared not only by processes of the same program by also by processes of different programs that use the same libraries. This framework also facilitates the update of models. When a system updates dynamic libraries, the program models do not need to be completely relearned. Experimental results also demonstrate that a library model learned for one program can help reduce the false positive rate of other programs that use the same dynamic library.

This dissertation presents a detailed evaluation of the overhead associated with different aspects of the learning and detection modules. We present an empirical evaluation of operations for methods based on call stack information. Experimental results shows that both the call stack walk and address transformation operations impose acceptable overheads on the program *cat*, *gzip*, and *gunzip*. The overhead for call stack walking for a web server *Apache* is also negligible. The address transformation introduces an overhead of less than 3% for the *Apache* server. These results provide evidence that detection methods based on call stack information can be used in the real-world systems.

The efficiency of our approaches were also measured in this dissertation. A measurement of our prototype system reveals a 7% overhead on the *Apache* server and negligible overhead for *cat*. Overhead reported for similar approaches are much higher. For example, *Dyck* has a 56% overhead on *cat* and 135% overhead on *htzipd* a simpler web server than *Apache*. *VPStatic* has a 32% overhead on *cat* and 97% overhead on *htzipd*.

5.2 Limitation and future work

Our current implementation of static analysis does not apply a pointer analysis algorithm to reveal the targets of indirect calls for static analysis. Giffin et al. [42] report that a good pointer analysis algorithm can recover most indirect calls (70% or 80%). Use of a pointer analysis program would enable our system to acquire a more detailed base model by static analysis and thus further reduce the false positive rate.

Our current implementation ignores the behavior in signal handlers. A further improvement can include separate models for each signal handler and switch the model to the corresponding model when a signal is received.

Our approach can potentially be used in areas other than intrusion detection and anomaly detection. Four possible areas are 1) detection of hidden code in code developed by untrusted entities, 2) defense against worms and Spyware, and 3) creation of an automata model for model checking, and 4) collection of new attacks using a honeypot.

It is critical to make sure that a product developed by untrusted sources does not contain any hidden code that provides back doors for future attacks. Manual review of all code is not practical. However, it is possible to build the models of behavior using dynamic learning during the black-box evaluation of the product and then detection of behavior from hidden code becomes an anomaly detection problem. The ability to detect the hidden behavior is based on an assumption that the malicious operations are not activated during the black-box testing and thus the acquired model does not contain behaviors for the malicious operations. The feasibility of applying our anomaly detection approach for discovery of hidden codes should be further studied with real-world products.

Worms and spyware have become a serious problem in recent years. Our approach can be used to build a secure system that prevents any execution of worms or spyware. All processes running on the secure system will be monitored and protected and new processes that do not have a known profile will be prevented from executing. This approach can prevent the popular worms such as Code Red and MS Blaster that exploit the software vulnerability in Windows to propagate. A further study of efficiency issued related to monitoring of the entire system should be conducted and the structure of the protection system should be designed.

Automata models acquired by static analysis of program source code have been used as the base model for model checking and several unknown vulnerabilities have been discovered using this method [17]. The possibility of using the model acquired by our static analysis method with model checking approach requires further study.

As described in section 4.3, a post analysis using the information collected during a detection of stack overflow attack may reveal the location of a vulnerability in the software that was attacked. It is possible to integrate our system as a component in a honeypot system. Our system could monitor all of the service programs provided by the honeypot system, detect any anomalies, and collect the information for post analysis when an anomaly is detected. This approach could potentially make it possible to discover software vulnerabilities that are unknown by security experts but that are already be employed by attackers.

REFERENCES

[1] "IDA Pro homepage," http://www.datarescue.com/idabase/ (current 5 Sept. 2003).

[2] "SELINUX: Security enhanced Linux," (http://www.nsa.gov/selinux/) (current May 2005).

[3] Anonymous, "Once upon a free(...)," *Phrack*, vol. 11, no. 57, 2001.

[4] M. Balaban, "Designing shellcode demystified," 2002, (http://www.linuxsecurity.com/content/view/117709/49/) (current September 2004).

[5] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," *Proceedings: SPIN 2001, Workshop on Model Checking of Software*, 2001.

[6] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," *Proceedings: ACM Symposium on Principles of Programming Languages 2002*, 2002.

[7] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," *Proceedings: USENIX Annual Technical Conference*, San Diego, California, 2000.

[8] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovi, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," *Proceedings: 10th ACM Conference on Computer and Communications Security*, Washington, DC, 2003, pp. 281–289.

[9] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*, Prentice Hall, 1990.

[10] S. M. Bellovin, "Computer security - an end state?," *Communications of the ACM*, vol. 44, no. 3, 2001, pp. 131–132.

[11] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "REMUS: a security-enhanced operating system," *ACM Transactions on Information and System Security*, vol. 5, no. 1, 2002, pp. 36–61.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proceedings: USENIX Security Symposium*, Washington, DC, 2003.

[13] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*, 2th edition, O'Reilly & Associates, Inc., Sebastopol, California, 2002.

[14] "Buffer overflow vulnerability in program kon2 0.3.9b and earlier," 2003, (https://rhn.redhat.com/errata/RHSA-2003-047.html) (current July 2005).

[15] "Buffer overflow vulnerability in program XV versions 3.10a and prior," 2004, (http://secunia.com/advisories/12352/ and http://www.derkeiler.com/Mailing-Lists-/Securiteam/2004-08/0078.html) (current July 2005).

[16] S. N. Chari and P.-C. Cheng, "BlueBox: a policy-driven, host-based intrusion detection system," *ACM Transactions on Information and System Security*, vol. 6, no. 2, May 2003, pp. 173–200.

[17] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code," *Proceedings: 11th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2004.

[18] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," *Proceedings: 9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, 2002, pp. 235–244.

[19] M. Conover, "w00w00 on heap overflows," 1999, (http://www.securityfocus.com-/archive/1/12170) (current June 2004).

[20] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: automatic protection from printf format string vulnerabilities," *Proceedings: USENIX Security Symposium*, Washington, DC, 2001.

[21] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard^{TM}: Protecting Pointers From Buffer Overflow Vulnerabilities," *Proceedings: USENIX Security Symposium*, Washington, DC, 2003.

[22] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, PerryWagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," *Proceedings: USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 63–78.

[23] D. E. Denning, "An intrusion detection model," *Proceedings: IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, 1986, pp. 118–131.

[24] S. Designer, "Non-executable user stack," Openwall Project, (http://www.openwall.com/linux/) (current Jan 2005).

[25] S. Designer, "JPEG COM Marker Processing Vulnerability in Netscape Browsers," July 2000, (http://www.securityfocus.com/archive/1/71598) (current June 2004).

[26] T. G. Dietterich, "Machine learning for sequential data: a review," *T. Caelli (Ed.) Structural, Syntactic, and Statistical Pattern Recognition; Lecture Notes in Computer Science*, 2002.

[27] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," *Proceedings: ACM Conference on Programming Language Design and Implementation*, 2003.

[28] D. C. DuVarney, V. Venkatakrishnan, and S. Bhatkar, "SELF: a Transparent Security Extension for ELF Binaries," *Proceedings: New Security Paradigms Workshop*, Ascona, Switzerland, 2003.

[29] J. Erickson, *Hacking: The Art of Exploitation*, No Starch Press, San Francisco, California, 2003.

[30] E. Eskin, W. Lee, and S. Stolfo, "Modeling system call for intrusion detection using dynamic window sizes," *Proceedings: 2001 DARPA Information Survivability Conference & Exposition*, Anaheim, CA, June 2001.

[31] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2004, IEEE Computer Society.

[32] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2003, IEEE Computer Society.

[33] G. Florez, *Incremental learning of discrete hidden Markov models*, doctoral dissertation, Department of Computer Science and Engineering, Mississippi State University, August 2005.

[34] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," *Proceedings: IEEE Symposium on Security and Privacy*, Los Alamitos, California, 1996, pp. 120–128.

[35] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," *Proceedings: Sixth Workshop on Hot Topics in Operating Systems*, Los Alamitos, California, 1997, pp. 67–72.

[36] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer overrun detection using linear programming and static analysis," *Proceedings: the 10th ACM Conference on Computer and Communication Security*, 2003.

[37] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," *Proceedings: the 12th ACM Conference on Computer and Communication Security*, 2004.

[38] D. Gao, M. K. Reiter, and D. Song, "On gray-box program tracking for anomaly detection," *Proceedings: USENIX Security Symposium*, 2004.

[39] A. K. Ghosh and A. Schwartzbard, "Learning program behavior profiles for intrusion detection," *Proceedings: 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, 1999, pp. 51–62.

[40] A. K. Ghosh and A. Schwartzbard, "A study in using neural networks for anomaly and misuse detection," *Proceedings: USENIX Security Symposium*, Washington, D.C., 1999, pp. 2–16.

[41] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," *Proceedings: 11th USENIX Security Symposium*, 2002.

[42] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection," *Network and Distributed System Security Symposium*, 2004.

[43] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," *Proceedings: IEEE Symposium on Security and Privacy*, Oakland, California, 2005, IEEE Computer Society.

[44] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2003, IEEE Computer Society.

[45] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," *Proceedings: 10th ACM Conference on Computer and Communications Security*, Washington, DC, 2003, pp. 272–280.

[46] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," *Proceedings: USENIX Security Symposium*, 2002.

[47] C. Ko, "Logic induction of valid behavior specifications for intrusion detection," *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2000, pp. 142–153.

[48] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," *Proceedings: Annual Computer Security Applications Conference*, Orlando, Florida, 1994, pp. 134–144.

[49] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, John Wiley & Sons, 2004.

[50] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," *Proceedings: 8th European Symposium on Research in Computer Security (ESORICS)*, Springer Verlag, Norway, 2003, Lecture Notes in Computer Science.

[51] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," *Proceedings: ACM Conference on Computer and Communications Security*, 2003.

[52] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," *Proceedings: 10th USENIX Security Symposium*, 2001.

[53] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," *Proceedings: USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 79–94.

[54] Z. Liang, V. Venkatakrishnan, and R. Sekar, "Isolated program execution: an application transparent approach for executing untrusted programs," *Proceedings: Annual Computer Security Applications Conference*, 2003.

[55] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings: ACM Conference on Computer and Communication Security*, Washington, DC, 2003.

[56] Z. Liu and S. M. Bridges, "Dynamic Learning of Automata from the Call Stack Log for Anomaly Detection," *Proceedings: International Conference on Information Technology (ITCC)*, Las Vegas, Nevada, 2005.

[57] Z. Liu, S. M. Bridges, and R. B. Vaughn, "Combining Static Analysis and Dynamic Learning to Build Accurate Intrusion Detection Models," *Proceedings: Third IEEE International Information Assurance Workshop*, College Park, Maryland, 2005.

[58] Z. Liu, G. Florez, and S. M. Bridges, "A comparison of input representations in neural networks: a case study in intrusion detection," *Proceedings: International Joint Conference on Neural Networks*, Honolulu, Hawaii, 2002.

[59] X. Lu, "A linux executable editing library," 1999.

[60] C. Michael and A. Ghosh, "Simple, state-based approaches to program-based anomaly detection," *ACM Transactions on Information and System Security*, vol. 5, no. 3, 2002, pp. 203–237.

[61] T. Mitchem, R. Lu, and R. O'Brien, "Using kernel hypervisors to secure applications," *Proceedings: the 13th Annual Computer Security Applications Conference*, Washington, DC, USA, 1997, IEEE Computer Society.

[62] Nergal, "Advanced return-into-lib(c) exploits (PaX case study)," 2001, (http-://www.phrack.org/show.php?p=58&a=4) (current September 2004).

[63] "Intrusion detection datasets from University of New Mexico," University of New Mexico, http://www.cs.unm.edu/ immsec/(current June 2004).

[64] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, 1996.

[65] "Pax Team," (http://pax.grsecurity.net/) (current September 2004).

[66] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," *Proceedings: USENIX Annual Technical Conference, General Track*, 2003.

[67] N. Provos, "Improving host security with system call policies," *Proceedings: 12th USENIX Security Symposium*, Washington, DC, 2003, IEEE Computer Society.

[68] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of Win32/Intel executables using etch," *Proceedings: USENIX Windows NT workshop*, 1997.

[69] Y. Sakakibara, M. Brown, R. Hughey, I. S. Mian, K. Sjolander, R. C. Underwood, and D. Haussler, "Stochastic context-free grammars for tRNA modeling," *Nucleic Acid Res.*, 1994.

[70] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," *Proceedings: Working Conference on Reverse Engineering*, 2002, pp. 45–54.

[71] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," *Proceedings: IEEE Symposium on Security and Privacy*, Oakland, California, 2001, IEEE Computer Society, pp. 144–155.

[72] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format-string vulnerabilities with type qualifiers," *Proceeding: USENIX Security Symposium*, Washington, D.C., 2001.

[73] K. M. Tan and R. A. Maxion, ""Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector," *Proceedings: IEEE Symposium on Security and Privacy*, Berkeley, California, 2002, IEEE Computer Society, pp. 173–186.

[74] J. Viega, T. Kohno, and G. McGraw, "Token-based scanning of source code for security problems," *ACM Transactions on Information and System Security*, vol. 5, no. 3, 2002, pp. 238–261.

[75] D. Wagner, *Static analysis and computer security: new techniques for software assurance*, doctoral dissertation, Department of Computer Science, University of California at Berkeley, Fall 2000.

[76] D. Wagner and D. Dean, "Intrusion detection via static analysis," *Proceedings: IEEE Symposium on Security and Privacy*, Oakland, California, 2001, IEEE Computer Society, pp. 156–169.

[77] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," *Proceedings: Network and Distributed System Security Symposium*, San Diego, California, 2000, pp. 3–17.

[78] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," *Proceedings: ACM Conference on Computer and Communications Security*, 2002.

[79] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," *Proceedings: IEEE Symposium on Security and Privacy*, Los Alamitos, California, 1999, pp. 133–145.

[80] A. Wespi, M. Dacier, and H. Debar, "Intrusion detection using variable length audit trail patterns," *Proceedings: the 2000 Recent Advances in Intrusion Detection*, 2000, pp. 110–129.

[81] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," *Proceedings: The 10th Annual Network and Distributed System Security Symposium*, San Diego, California, 2003.

[82] R. Wojtczuk, "Defeating Solar Designer's Non-executable Stack Patch," (http://www.insecure.org/sploits/non-executable.stack.problems.html) (current September 2004).

[83] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: general security support for the linux kernel," *Proceedings: USENIX Security Symposium*, 2002.

[84] H. Xie and P. Biondi, "The linux intrusion detection project," (http://www.lids.org/) (current May 2005).

[85] H. Xu, W. Du, and S. J. Chapin, "Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths," *Proceedings: Seventh International Symposium on Recent Advances in Intrusion Detection (RAID)*, French Riviera, France, 2004.

[86] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," *Proceedings: 22nd Symposium on Reliable and Distributed Systems*, Florence, Italy, 2003.