Efficient Intrusion Detection Based on Static Analysis and Stack Walks

Jingyu Hua[1], Mingchu Li[1], Kouichi Sakurai[2], and Yizhi Ren[1,2]

[1] School of Software, Dalian University of Technology,
Dalian 116620, China
huajingyu@gmail.com, li_mingchu@yahoo.com
[2] Dept.of Computer Science and Communication Engineering, kyushu University,
Fukuoka 812-0053, Japan
sakurai@csce.kyushu-u.ac.jp, renyizhi@gmail.com

Abstract. Some intrusion detection models such as the VPStatic first construct a behavior model for a program via static analysis, and then perform intrusion detection by monitoring whether its execution is consistent with this behavior model. These models usually share the highly desirable feature that they do not produce false alarms but they face the conflict between precision and efficiency. The high precision of the VPStatic is at the cost of high space complexity. In this paper, we propose a new context-sensitive intrusion detection model based on static analysis and stack walks, which is similar to VPStatic but much more efficient, especially in memory use. We replace the automaton in the VPStatic with a state transition table (STT) and all redundant states and transitions in VPStatic are eliminated. We prove that our STT model is a deterministic pushdown automaton (DPDA) and the precision is the same as the VPStatic. Experiments also demonstrate that our STT model reduces both time and memory costs comparing with the VPStatic, in particular, memory overheads are less than half of the VPStatic's. Thereby, we alleviate the conflict between precision and efficiency.

1 Introduction

When a program is attacked, such as injected malicious codes, it will behave in a manner inconsistent with its binary code, which can be made use of to perform intrusion detection. We can do a static analysis of the binary code to construct a behavior model, and then different kinds of attacks can be detected by monitoring whether the execution of this program deviates from this model. Actually, a lot of IDSs [2, 3, 4, 6, 9, 10, 11, 12] based on this idea has been proposed since 2000. Because of system calls are easy to be monitored at runtime, most of these systems use system calls to model the program behavior. These models usually do not produce any false alarms because they capture all correct execution behaviors via static analysis. This is the biggest reason why they are appreciated.

According to [3], the precision of intrusion detection models generated via static analysis can be divided into at least two levels. Models on the first level are flow-sensitive and they just consider the order of execution of statements in the program, such as the system call sequences. Models on the second level are context-sensitive, which are more precise. They keep track of calling context of functions and are able to match the return of a function with the call site that invoked it. As a result, they are immune to the impossible path problem [4]. However, in most time, accurate is incompatible with efficient. Context-sensitive models are more accurate at the cost of higher program running time and space caused by the overheads of maintaining context information. Our purpose in this paper is right to decrease these overheads to construct an efficient context-sensitive intrusion detection model via static analysis.

1.1 Previous Work

In 2001, Wanger and Dean [4] proposed a precise abstract stack model generated via static analysis of C source code. This model uses stack states maintained in the abstract stack to model the call and return behaviors of function calls. Hence, this model is context-sensitive. Unfortunately, this model is a non-deterministic push down automaton (PDA). As a result, the time and space complexities are so high that it's not practical.

Feng and Giffin [2] pointed out severe non-determinism in the stack state is the major contributing factor to the high time and space complexities of PDA operations. They proposed two different models: Dyck and VPStatic to eliminate this non-determinism to improve the online detection efficiency.

The Dyck model [2, 12] is based on code instrumentation. It uses binary rewriting to insert code before and after each function call site to generate extra symbols needed for stack determinism. However, because the Dyck model is just a stack-deterministic PDA (sDPDA), not a complete deterministic PDA (DPDA), it still requires linear time when waking in the automaton. What's worse, its time efficiency is also affected by the overheads of new inserted codes. As a result, the time complexity for the Dyck model is still too high that slowdowns of 56% and 135% are reported for two linux self-contained programs: *cat* and *htzipd*.

The VPStatic [2] is a statically constructed variant of the dynamic context-sensitive VtPath model [7]. It also uses a statically constructed automaton to model the call and return behavior of function calls between two consecutive system calls, but stack walks are used to observe existing context-determining symbols to eliminate non-determinability. It is a provably DPDA and dose not do any instrumentation. Thereby, the time efficiency is much higher than the Dyck model. However, the VPStatic produces much lager automaton structures than the Dyck model which leads to a higher memory use. Increases of 183% and 194% are reported for *htzipd* and *cat* in memory uses.

1.2 Our Contribution

Our work is focused on constructing a model similar to VPStatic that is a DPDA and efficient in time but with a much lower memory use. Specifically, we make the following contributions:

- We propose a new context-sensitive intrusion detection model called STT based on static analysis and stack walks. We replace the automaton in the VPStatic with a state transition table, which records correct transitions among system call sites and corresponding execution contexts directly. The walk in the automaton is becoming a search in the table, which is more efficient. We use a delta optimization to solve the state explosion problem due to the use of the STT. There're no redundant states and transitions in our STT model. According to our analysis, the number of states in the STT is much less than half of that in the VPStatic for the same program. As a result, the memory overheads are greatly reduced.
- We formally define the STT model and prove it's a deterministic push down automaton (DPDA), which means its time efficiency is at least as high as the VPStatic.
- We prove our STT model has the same precision with the VPStatic. It accepts all VPStatic accepts and refuses all VPStatic refuses. So we improve the efficiency without reducing the precision.
- We implement dynamically-constructed VPStatic models and STT models for programs *gzip* and *cat*. Experiments results show the memory overheads of the STT models are less than half of the VPStatic models'.

```
char* filename; pid_t[2] pid;
int prepare(int index) {
  char buf[20];
  pid[index] = getpid();
  strcpy(buf, filename);
  return open(buf,O_RDWR);
}
void action(){
  uid_t uid = getuid();
  int handle;
  if (uid!= 0)
  {
    handle = prepare(1);
    read(handle, ...);
  }
  else
  {
    handle =prepare(0);
    write(handle, ...);
  }
  close(handle);
}
```

Fig. 1. A sample program fragment. This fragment is composed by two functions: *prepare* and *action*. Functions *getpid*, *open*, *getuid*, *read*, *write* and *close* are system calls.

2 STT Model

Our model is a statically-constructed context-sensitive intrusion detection model. We will first use the sample program in Fig.1 to illustrate its basic idea.

2.1 Basic Idea

See Fig.1, assume we capture two consecutive system calls: *getuid* and *getpid* in an execution of the program and corresponding user stacks are presented in Fig.2. We can perform intrusion detection by checking whether such transitions for both system calls and stack states are possible according to the program's binary code. As we known, the stack state represents the real-time calling context of functions. According to the source code, the system call *getpid* is right following the system call *getuid*. Between them, only a function *prepare* is called (in line 13 or 18), so a new stack frame for *prepare* will be pushed into the call stack, which means the transition of the stack state is correct, too. Therefore, we say this program is still running normally by now. Our model is just based on this idea. We use a state transition table which is constructed via a static analysis of the binary of a program to record all these correct transitions. We perform online intrusion detection by verifying whether both the system call and stack state traces of the execution are consistent with the table. Because we make use of a state transition table, we name our model STT.

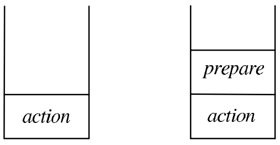

Fig. 2. Abstract stack states when *getuid* is called (left) and *getpid* is called (right)

2.2 Structure of the State Transition Table

In our model, we assume that when a system call is invoked, the monitored program will transfer to a new state. The STT is used to decide what new state the program should transfer to after a system call is invoked according to the binary.

Table 1 presents the STT for the sample program in Fig.1. The left header of the table are states of the program. We assign each an increasing unique id. Every state is corresponding to a system call site and composed by two parts: the address of the system call site and the stack state when the program executes to this site. In order to reduce the memory use of the STT, we use a fixed-length hash value of the return address list in the stack to represent the stack state. In this paper, we assume the hash function we use is so perfect that the probability of conflicts of hash values can be ignored. The first state $< h(a), s\_getuid >$ means the program invokes a system call *getuid* at address *s_getuid* and the

Table 1. State transition table for the sample program in Fig.1. a, $p1$ and $p0$ are return addresses of *action*, $prepare(1)$ and $prepare(0)$,respectively. String with the prefix $s\_$ represents the address of the corresponding system call.

	getuid	*getpid*	*open*	*read*	*close*	*write*
1:$< h(a), s\_getuid >$		2,3				
2:$< h(ap1), s\_getpid >$			4			
3:$< h(ap0), s\_getpid >$			5			
4:$< h(ap1), s\_open >$				6		
5:$< h(ap0), s\_open >$						8
6:$< h(a), s\_read >$					7	
7:$< h(a), s\_close >$						
8:$< h(a), s\_write >$					7	

stack state is $h(a)$ at this site, where h is the hash function. The top header of the table are system calls that trigger the program transferring from the current state to a new one. Assume we are at state 1, and then if the system call *getpid* is captured, according to the STT, the program should transfer to state 2 or state 3.

Because we take stack state into consideration, we define different states for the same system call site when it is executed in different contexts. For example, in the sample program, the system call *open* can be invoked either in $prepare(0)$ or $prepare(1)$. As a result, we define two sates: 4 and 5, for this single system call site. We do this to make sure our model is context sensitive and immune to the impossible path problem [4].

2.3 Online Intrusion Detection

If the STT for a program has been statically constructed, we can use it to monitor the execution of the program. Intrusion detection is performed every time when a new system call is captured. The whole process contains three steps:

1. Use the new captured system call s and the last state to search for the expected state set Q_e in the STT.
2. Walk the current user stack to extract the return address list B and then compute the real state $q =< h(B), s\_s >$ the program is at.
3. Then, if $q \in Q_e$, nothing is wrong, but if $q \notin Q_e$, an attack is considered having occurred.

Let's use an overflow attack targeting to the sample program in Fig.1 to illustrate this process. Assume when the program is executing in the function call $prepare(1)$, an attacker overflows buf using *strcpy* (Line 5) and modifies the return address of this call to the address of $prepare(0)$, then the system call sequence will become $getpid- > open- > getpid- > open- > write- > \cdots$.

The detection process for this attack is presented in Table 2. When the system call *open* is captured at the first time, the real state is $< h(ap0), s\_open >$.

Table 2. Online Detection Process for the overflow attack to the sample program in Fig.1

Captured System Call	Last State	Expected States	Real State	Detect Result
getpid	1	2,3	$< h(ap1), s\_getpid >= 2$	normal
open	2	4	$< h(ap0), s\_open > \neq 4$	attack

However, the current state is expected to be $< h(ap1), s\_open >$ according to the STT. Thereby, we detect the attack.

2.4 Delta Optimization

$$f_n \xrightarrow{a_{n-1}} f_{n-1} \xrightarrow{a_{n-2}} \cdots f_2 \xrightarrow{a_1} f_1 \xrightarrow{1} s$$

Fig. 3. A special situation where a system call s is invoked by the first function f_1 at some site, f_1 itself is invoked at a_1 different sites by the second function f_2, f_2 is invoked at $a_2$different sites by the third function and so on

2.4.1 State Explosion Problem

By now, our STT model suffers the state explosion problem. Because our model is context sensitive, we define different states for the same system call site in different execution contexts. Let's consider the special case presented in Fig.3. In this case, the system call s can be invoked in $a_1 \times a_2 \times \cdots \times a_{n-1}$ different contexts, as a result, the total number of states defined for it is $a_1 \times a_2 \times \cdots \times a_{n-1}$. This is so called state explosion, which means our current model will scale poorly for large programs because of the soaring number of states. Fortunately, we can use a method named delta optimization by us to solve this problem.

2.4.2 Delta Optimization

We find that for two consecutive states, they must share a common prefix between their return address lists in the stack. For example, state 1 and state 2 in Table 1 are consecutive and their return address lists share a as the common prefix. According to this, we redefine the state:

Definition 1. *Let C be the common prefix between the current return address list B and the last return address list A. Then, the state for the current system call site in the delta optimized STT is a triple $S =< l, d, s >$, where:*
l is the length of the postfix [1] *of B excluding C.*
d is the hash value of the postfix of B excluding C.
s is the address of the current system call site.

This definition will reduce the number of states in the STT greatly. Let's consider the system call *open* in the sample program in Fig.1 again. Table 3 presents the

[1] This postfix is the delta part (different part) of B compared with A. So we call this optimization method Delta Optimization.

Table 3. State definitions for the system call *open* of the sample program in Fig.1 in two different execution contexts

Context	Last System Call Site	Last Return Address List	Current Return Address List	State Definition
$prepare(1)$	s_getpid	$ap1$	$ap1$	$< 0, none, s\_open >$
$prepare(0)$	s_getpid	$ap0$	$ap0$	$< 0, none, s\_open >$

Table 4. New state transition table for the sample program in Fig.1 after the delta optimization

	getuid	*getpid*	*open*	*read*	*close*	*write*
1:$< 1, h(a), s\_getuid >$		1,2				
2:$< 1, h(p1), s\_getpid >$			4			
3:$< 1, h(p0), s\_getpid >$			4			
4:$< 0, none, s\_open >$				$5 - (2, p1)$		$7 - (2, p0)$
5:$< 0, none, s\_read >$					6	
6:$< 0, none, s\_close >$						
7:$< 0, none, s\_write >$					6	

state definition for this single site in two different execution contexts. We find that l, d, s remain the same in the two different contexts. As a result, state 2 and state 3 in Table 1 are merged into one state $< 0, none, s\_open >$. In fact, if a function is called in n different sites, after the delta optimization, we only define one state for each system call in the function except the first one [2], for which we still define n states. Thereby, now the total number of states defined for s in the case presented in Fig.3 is less than $a_1 + a_2 + \cdots + a_{n-1}$, which is linear to the number of function calls. So, our new STT model can scale well for large programs.

Although we redefine the state, the online intrusion detection algorithm described in Sec.2.3 remains the same on the whole and the only difference is the way to compute the real state q. Before the delta optimization, q is computed based on the current return address list B got from the stack walks. However, now, we have to compute q based on not only B but also last state's return address list A. As a result, we need to keep track of the last state's return address list during the online monitoring.

Table 4 presents the new STT for the sample program in Fig.1 after the delta optimization. We will describe how to construct it via static analysis in Sec.2.5.

2.4.3 Side Effect of Delta Optimization

Unfortunately, the delta optimization has side effect that it will bring us impossible path problem again. See Table 4, after the optimization, state 4 can either

[2] Note that the first system call of a function refers to the first system call the program will invoke after entering the function, before leaving the function.

transfer to state 5 or state 7 with out any limit. Actually, state 4 can transfer to state 5 only when *open* is called in $prepare(1)$ and to state 7 only when *open* is called in $prepare(0)$. We find states suffering this problem are all last system calls [3] of functions. The reason behind this problem is we neglect the execution context of the function. When a function is called at different sites, we only define one state for its last system call (Assume the function will invoke more than one system call) after the delta optimization. Then, if the program transfers to different states when the same function returns from different sites, we have no ways to distinguish among them just based on the triple of the state. However, we should remember that the return address of the function exits in the return address list. We can turn to it to identify which call site we are returning from and then decide which state we should transfer to. So we add a transfer condition to each transition of these states. A condition specifies the position of the return address of the function in the return address list and what value it should be equal to if the program follows the corresponding transition. In Table 4, we add condition $(2, p1)$ to the transition from state 4 to state 5, which means if this transition takes place, the second address in the return address list of state 4 should be equal to $p1$. By this way, we can solve the impossible problem due to delta optimization completely.

2.5 Model Generation via Static Analysis

Before we can monitor the running of a program, we have to build a STT for it. The STT model for a program is built via a static analysis of its binary executable. We first disassemble the binary, and then we analyze the disassembled instructions recursively following the control flow of the program. We maintain a virtual stack to simulate the real stack: when the analyzer enters into a function, its return address is pushed into the virtual stack and when it leaves the function, the return address is popped out of the stack. In our algorithm, we don't care any other types of instructions except the following three ones:

System Call Instructions: When the analyzer comes to a system call instruction, we use the instruction address, current return address list in the virtual stack and the last state's return address list to create a new state according to Definition 1. If this state has been already in the STT, which means the current analysis path has been covered before, we stop going on analyzing along this path and return. Otherwise, we insert the new state into the STT and update the last state of the analyzer to be this new state. Then, we continue the analysis along this path. In addition, in both cases we need to add a new transition from the last state to the current state in the STT.

Jump Instructions: When the analyzer comes to a jump instruction, we first recursively invoke the analysis algorithm from the target address. Then, after that process returns, we continue at the address following the jump instruction.

[3] Note that the last system call of a function refers to the last system call the program will invoke after entering the function, before leaving the function.

Function Call Instructions: As we said before, except the first one, we only define one state for each system call site in the same function after the delta optimization. As a result, no need to analyze the same function repeatedly. When the analyzer comes to a function call, we first judge whether this function has been analyzed before. If not, we enter it and recursively invoke the analysis algorithm from the beginning of the function. When we create the first state of this function, we store the address s of the system call and the postfix R of its return address list, which starts after the return address of the function. After finishing the analysis, we store the last state of the function and the postfix R' of its return address list, which also starts after the return address of the function. Then, when we revisit this function at other site t, no need to re-analyze this function but just do two things. Firstly, we create a new state $< length(R) + 1, h(tR), s >$, where $length(R)$ means the length of R, and then add a new transition from the last state before we come to the function to this new state in the STT. Secondly, we update the last state of the analyzer to be the last state of the function stored earlier and its return address list should be modified to VtR', where V is the address list in the virtual stack. Then, we go on analyzing at the address after the function call. When we come to a new state and add a new transition to the last state of the function, we have to add a transfer condition $(length(V) + 1, t)$ to this transition to avoid the side effect described in Sec.2.4.3.

3 Formal Proof That the STT Model Is a DPDA

Our STT model can be considered as a push down automaton (PDA). We use the formal language described in [2, 14] to define it formally and prove that it's deterministic.

Definition 2. *The STT model is a push down automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where:*

Q is the set of states. Every state is a triple defined as Definition 1.

Σ is the input alphabet to the automaton. If $a \in \Sigma$, then $a = (s, z')$, where: s is the address of the current system call site and z' is the real return address list got from stack walks.

Γ is the stack alphabet. $z \in \Gamma$ is the last state's return address list.

δ is the transition relation mapping $Q \times \Sigma \times \Gamma$ to $Q \times \Gamma$. Let $z' = b_1 b_2 \cdots b_n$, $z = a_1 a_2 \cdots a_m$ and l be the length of the common prefix of z' and z. Then, the real state the program located at is $q' =< n - l, hash(b_l b_{l+1} \cdots b_n), s >$. On the other hand, we can search the STT and find the expected state set Q_e. Then,

$$\delta(q, a, z) = \begin{cases} none & q' \notin Q_e \\ (q', z') & q' \in Q_e \end{cases} \tag{1}$$

$q_0 \in Q$ is the unique initial state and $z_0 \in \Gamma$ is the initial stack state. $F \subseteq Q$ is the set of accepting states.

Theorem 1. *The STT Model is a deterministic PDA (DPDA).*

Proof. A PDA is called deterministic if the transition relation δ satisfies the following two conditions [2, 14]:

Condition 1: For all $q \in Q$ and $z \in \Gamma$, whenever $\delta(q, \varepsilon, z)$ is nonempty, then $\delta(q, a, z)$ is empty for all $a \in \Sigma$.

Condition 2: For all $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $z \in \Gamma$, $\delta(q, a, z)$ contains at most one element.

First, ε-transition doesn't exist in our model. So Condition 1 is satisfied.

Second, according to (1), $\delta(q, a, z)$ contains none element or one element, so Condition 2 is also satisfied.

Therefore, we obtain the conclusion that the STT Model is a DPDA.

According to [2], the time complexity for processing an input symbol with a non deterministic PDA is $O(nm^2)$, where n and m denote the number of states and transitions, respectively. However, if the PDA is deterministic, the time complexity will be reduced to $O(1)$. So our STT model is efficient in PDA operation.

4 Comparison between STT and VPStatic

Both the STT and the VPStatic perform intrusion detection by monitoring the system call events of the program. The VPStatic uses a virtual path in a statically-constructed automaton to record the call and return behaviors of function calls between two consecutive system calls. In our STT model, we replace this automaton with a state transition table, which records context-sensitive transitions between two consecutive system calls directly. We do a comparison between the two models in precision, time complexity and space complexity.

4.1 Precision

STT performs intrusion detection each time a system call is made. Assume s_B is the new captured system call and b_{n+1} is its address. The return address list got from stack walks is $b_1 b_2 \cdots b_n$. Also assume s_A, a_{m+1} are the last system call and its address, respectively. Its return address list is $a_1 a_2 \cdots b_m$. Suppose l is the length of the common prefix of A and B. We assume everything is ok when the program is at a_{m+1}.

Then, for the STT, if the following two conditions are both satisfied, it will accept the new system call s_B and consider the program is running normally:

Condition 1: $S_B =< n - l, h(b_l b_{l+1} \cdots b_n) >$ is in the STT.

Condition 2: Let S_A be the state corresponding to the last system call. Then, S_A has a transition to S_B in the STT and if this transition contains a condition-the return address list A has to satisfy it.

For the VPStatic, it will generate a sequence of input symbols using A and B, and then feed them to its automaton one by one. If every symbol in the sequence is accepted by this automaton, the new system call s_B will be accepted, otherwise an alarm is raised. There are three kinds of input symbols: e, g and f in the VPStatic. The automaton for the sample program in Fig.1 is presented on the right side of Fig.4.

Theorem 2. *The STT Model has the same precision with the VPStatic Model, which means if s_B is accepted by the VPStatic, it will also be accepted by the STT, and if it is refused by the VPStatic, it will be refused by the STT, too.*

Proof. First, assume the input sequence is accepted by VPStatic, which means s_A and s_B are really consecutive and the correct return address list at b_{n+1} is truly B. So, according to the Definition 1 and the construction algorithm of the STT described in Sec.2.5, the state S_B must be in STT and S_A must have a transition to S_B. In addition, if this transition contains a condition, A must also satisfy it. So we satisfy the two conditions above and s_B is also accepted by the STT.

Secondly, assume the input sequence is not accepted by the VPStatic. Then, there're three cases:

Case 1: One g symbol in the input sequence is not accepted. Let this incorrect symbol be $g(none, a_i, a_i)$, where $i > l$. This means a_i dose not match the top symbol on the virtual stack of the VPStatic and the program returns to a wrong address. Thereby, this execution path does not exist in the real. In this situation, if S_B is still in the STT, there is only one possibility: s_A at a_{m+1} can be invoked in another context [4], in which there is an execution path from a_{m+1} to b_{n+1}, and we just define one state S_A for them because of the delta optimization, which is similar to the state 4 in Table 4. However, we add transfer conditions to all the transitions of this kind of states, based on which we can distinguish between different contexts. So even if Condition 1 can be satisfied in this case, Condition 2 can't be satisfied because the corresponding transition condition can't be satisfied. So s_B is not accepted by the STT, too.

Case 2: One f symbol in the input sequence is not accepted. We can use the similar way in Case 1 to prove the transition either will not be accepted by our STT model. We omit it here.

Case 3: One e symbol is not accepted. There're three sub-cases in this situation. If the incorrect symbol is $e(none, Exit(Func(a_i)))$, which means we can't return from the corresponding function at that time according to the binary. We may enter a new function or make a system call, which means the next symbol $g(none, a_{i-1}, a_{i-1})$ is either wrong. So we come to Case 1, which has been proved above. Else if the incorrect symbol is $e(none, b_i)$, which means we can't enter the function $Func(b_i)$ at present. Thereby, the last transition $f(none, Entry(Func(b_i)), b_{i-1})$ must be either wrong and we come to Case 2. At last if the incorrect symbol is $e(s_B, b_{n+1})$, which means we can't reach to b_{n+1} at present following this execution path. Then, either S_B is not in the STT or the corresponding transition condition is not satisfied, which can be proved similarly to Case 1. So an intrusion alarm will still be raised by the STT in this case.

Therefore, we obtain the result our STT model is as precious as the VPStatic. It accepts all VPStatic accepts and refuses all VPStatic refuses.

[4] To be more precise, the function corresponding to a_i can be called at another site in the program.

4.2 Space Complexity

We can learn from Case 3 in the proof of Theorem 2 that all e transitions except the last one are all redundant in the VPStatic Model. They are just equivalent to the next g transitions or the last f transitions. Therefore, all these transitions and corresponding states can be eliminated to compact the automaton. In STT, we replace the automaton in the VPStatic with a state transition table, which records the context-sensitive transitions among system calls directly. Every state in the STT is corresponding to a system call site and all those intermediate states and transitions between any two system call states, which are called virtual paths in the VPStatic, are all eliminated. As a result, the STT is much smaller than the VPStatic.

Theorem 3. *States in the STT are fewer than that in the VPStatic.*

Proof. The exact numbers of states in the two models for the same program are presented in Table 5, respectively. The VPStatic defines two states ('Entry' and 'Exit') for each function, two states ('f' and 'g') for each call site, and one state for each system call site. So there are totally $2m + 2n + p$ states in the VPStatic, where m, n, p denote the number of function call sites, functions and system call sites in a program, respectively. The STT defines one state for each system call site if it is not the first one of a function. For those first system calls, the STT defines t states for each one if the corresponding function is called in t different sites. So there're totally $q + p - k$ states in the STT, where k and q denote the

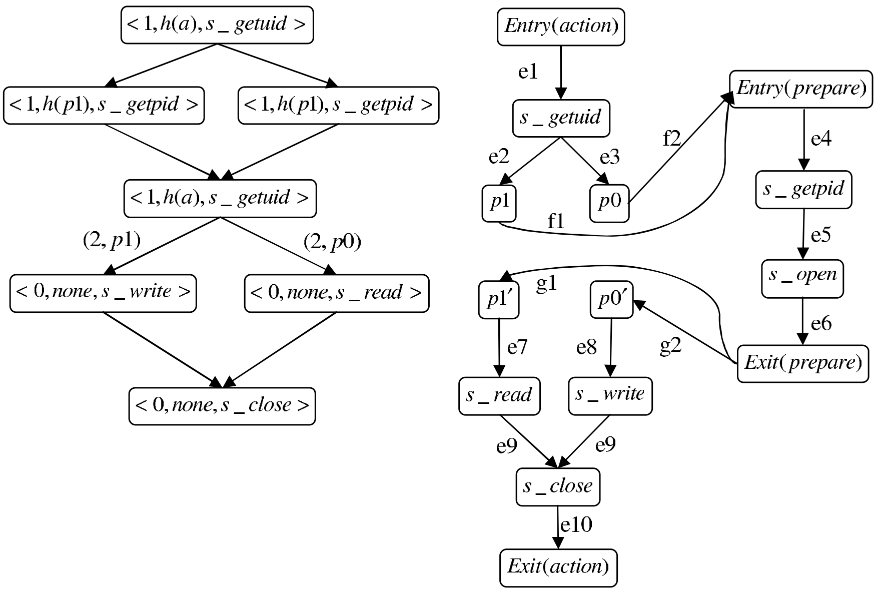

Fig. 4. STT and VPStatic automatons for the sample program in Fig.1. The left one is for the STT and the right one is for the VPStatic.

Table 5. Numbers of states in the STT and the VPStatic. Assume there are n functions, m call sites and p system call sits in the monitored program. Among the n functions, assume only k ones invoke system calls and they are called at q sites.

	VPStatic	STT
Number of States	$q + p - k$	2m+2n+p

Table 6. Time complexities for every step of processing a system call with the STT and the VPStatic. The lengths of the return address lists of the last state and the current state are denoted by m and n, respectively. The length of the common prefix of the two lists is denoted by l.

Model	Intrusion Detection Step	Time Complexity
STT	Search expected states in the STT	$O(1)$
	Compute the real state	$O(n)$
	Compare between the real state and a expected state	$O(1)$
VPStatic	Generate the input sequence	$O(l)$
	Walk the automaton	$O(m + n - 2 * l)$

number of functions which invoke system calls[5] and call sites of these special functions, respectively. Due to the fact $q < m$, the number of states in the STT are much fewer than that in the VPStatic (In most time much fewer than half).

In Fig.4 we present two automatons for the sample program in Fig.1. The right one describes the VPStatic and the left one describes our STT model. We can find the STT has been greatly compacted compared with the VPStatic.

4.3 Time Complexity

Theorem 4. *The time complexity of the STT is lower than that of the VPStatic.*

Proof. For both models, intrusion detection is performed every time when a system call is captured at runtime. So we compare the time costs for the two models to process a single system call. We divide this time cost into two components: the time to perform stack walks and the time to perform verification whether the new system call is accepted. Because the time to perform stack walks to extract the return address list on the stack is the same for the two models, we just consider the later here. The time complexities for every step of verifying a system call with the two models are presented respectively in Table 6. The total time for the STT is $O(n)$, while for the VPStatic it is $O(m + n - l)$. Because

[5] Note that we say a function invokes system calls so long as the program invokes any system calls after entering the function, before leaving the function.

$m > l$, we obtain the result that the time complexity of the STT is lower than that of the VPStatic.

Although we do improve the time efficiency, the improvement is not obvious: the time complexity to process a single system call is linear to the length of the return address list on the stack for both models. This is because the VPStatic is also a DPDA and its time efficiency is already very high. Actually, as we can see from the experiment results described in Sec.5, time overheads for both models are dominated by the time to perform stack walks and the time to perform verification is so small that can be ignored.

5 Experiments

Experiments are conducted to compare the time and memory overheads of the STT and VPStatic models. In this respect, we analyze two test programs: *gzip* and *cat*. Currently, we build the two models for these test programs via dynamic analysis but not static analysis. For every test program, we first execute it to finish a specific workload and capture all system calls and the corresponding stack states, based on which, we construct the STT and the VPStatic for this program. Then, for every model, we execute the program with the same workload for the second time. At this time, we use the model to monitor the execution of the program.

Due to the fact we can't cover all possible execution paths, these models built via dynamic analysis are far from complete and the true memory costs of models built via static analysis are much larger. However, comparisons between these dynamically constructed models still make sense. This is because they are constructed based on the same data. In addition, we can consider test programs we analyze are not *gzip* or *cat*, but just two new programs that formed by execution paths of *gzip* and *cat* we cover in the experiments. From this point, our models are truly complete.

Our experiments are carried on Fedora 7.0. We monitor the execution of a program in user space and process tracing is used to capture system call events. The workloads and corresponding execution statistics for each test program are presented in Table 7. Base time in the table refers to the time a program finishes its workload with process tracing enabled but doing nothing at each system call stop. We regard it as the execution time of a program without IDS.

Table 8 presents the accumulated time overheads for the dynamically constructed VPStatic and STT models to monitor the two test programs finishing their own workloads. We separate the models' runtime into two components: the time to perform stack walks and the time to perform verification. From this table, we find time overheads for both models are dominated by the time to perform stack walks. The STT does improve the time efficiency to perform verification but not obviously. Actually, compared with the overheads caused by stack walks, those caused by verification are so small that can be even ignored.

Numbers of states and memory overheads for the dynamically constructed VPStatic and STT models are presented in Table 9. From that we find our

Table 7. Workloads and corresponding execution statistics for test programs. Based times are measured in seconds.

Program	Workloads	System Call Events	Base Time
gzip	Compress a 24.4 MB tar file	2281	11.72
cat	Concatenate 40 files totaling 500MB to a file	520131	85.19

Table 8. Model execution times in seconds. Percentages compare against base execution.

Program	Model	Stack Walks	%	Verification	%
gzip	STT	9.25	79	0.03	0
	VPStatic	9.21	79	0.03	0
cat	STT	22.73	27	3.2	4
	VPStatic	22.75	27	5.12	6

Table 9. Numbers of states and memory uses in KB for models

Program	Model	Number of States	Memory Use
gzip	STT	41	0.750
	VPStatic	109	1.58
cat	STT	28	0.460
	VPStatic	87	0.94

STT models do reduce the numbers of states greatly. As a result, the memory overheads due to monitoring are also greatly reduced. In our experiments, all the memory uses of the STT models are less than half of the VPStatic models'.

6 Conclusion

We propose a novel efficient context-sensitive intrusion detection model via static analysis. It uses stack walks to eliminate non-determinability and is a provably DPDA, which is similar to the VPStatic. We replace the automaton in the VPStatic with a state transition table and the automaton walk in VPStatic is replaced by a search in the STT, which is more efficient. We perform a delta optimization to solve the state explosion problem of the STT and no redundant states and corresponding transitions, which exist in the automaton of the VPStatic, exist in our STT model. As a result, the memory use is greatly reduced. In our experiments, the memory overheads of the dynamically-constructed STT models for programs *gzip* and *cat* are both samller than half of the corresponding VPStatic models'. We prove that our STT model has the same precision with the VPStatic. Thereby, we improve the efficiency of the VPStatic greatly without reducing its precision, which alleviates the historical conflict between the efficiency and precision, which is suffered by similar intrusion detection models.

References

[1] Forrest, S., Longstaff, T.: A sense of self for unix processes. In: 1996 IEEE Symposium on Security and Privacy, pp. 120–128. IEEE Press, Oakland (1996)

[2] Feng, H.H., Giffin, J.T., Huang, Y., Jha, S., Lee, W., Miller, B.P.: Formalizing sensitivity in static analysis for intrusion detection. In: 2004 IEEE Symposium on Security and Privacy, pp. 194–208. IEEE Press, California (2004)

[3] Gopalakrishna, R., Spafford, E.H., Vitek, J.: Efficient Intrusion Detection using Automaton Inlining. In: 2005 IEEE Symposium on Security and Privacy, pp. 18–21. IEEE Press, Washington (2005)

[4] Wagner, D., Dean, D.: Intrusion detection via static analysis. In: 2001 IEEE Symposium on Security and Privacy, p. 156. IEEE Press, Oakland (2001)

[5] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: 9th ACM Conference on Computer and Communications Security, pp. 255–264. ACM Press, Washington (2002)

[6] Saidi, H.: Guarded Models for Intrusion Detection. In: 2007 Workshop on Programming languages and analysis for security, pp. 85–94. ACM Press, San Diego (2007)

[7] Feng, H., Kolesnikov, P.F., Lee, W.: Anomaly detection using call stack information. In: 2003 IEEE Symposium on Security and Privacy, p. 62. IEEE Press, Los Alamitos (2003)

[8] Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: 11th ACM Conference on Computer and Communications Security, pp. 318–329. ACM Press, Washington (2004)

[9] Giffin, J.T., Dagon, S., Jha, S., Lee, W., Miller, B.P.: Environment-sensitive intrusion detection. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 185–206. Springer, Heidelberg (2006)

[10] Feng, H.: Dynamic monitoring and static analysis: new approaches for intrusion detection. PhD Dissertation, University of Massachusetts Amherst (2005)

[11] Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: 6th Symposium on Operating Systems Design and Implementation, pp. 147–160. USENIX Association, Seattle (2006)

[12] Giffin, J.T., Jha, S., Lee, W., Miller, B.P.: Efficient context-sensitive intrusion detection. In: 11th Annual Network and Distributed Systems Security Symposium. Internet Society, San Diego (2004)

[13] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iver, R.K.: Non-control- data attacks are realistic threats. In: 14th USENIX Security Symposium, pp. 1–12. USENIX Association, Baltimore (2005)

[14] Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, New Jersey (2001)