Formalizing Sensitivity in Static Analysis for Intrusion Detection

Henry Hanping Feng*, Jonathon T. Giffin†, Yong Huang*,
Somesh Jha†, Wenke Lee‡, and Barton P. Miller†

*Department of Electrical and Computer Engineering
University of Massachusetts–Amherst
{hfeng,yhuang}@ecs.umass.edu

†Computer Sciences Department
University of Wisconsin–Madison
{giffin,jha,bart}@cs.wisc.edu

‡College of Computing
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract

A key function of a host-based intrusion detection system is to monitor program execution. Models constructed using static analysis have the highly desirable feature that they do not produce false alarms; however, they may still miss attacks. Prior work has shown a trade-off between efficiency and precision. In particular, the more accurate models based upon pushdown automata (PDA) are very inefficient to operate due to non-determinism in stack activity. In this paper, we present techniques for determinizing PDA models. We first provide a formal analysis framework of PDA models and introduce the concepts of determinism and stack-determinism. We then present the VP-Static model, which achieves determinism by extracting information about stack activity of the program, and the Dyck model, which achieves stack-determinism by transforming the program and inserting code to expose program state. Our results show that in run-time monitoring, our models slow execution of our test programs by 1% *to* 135%*. This shows that reasonable efficiency needs not be sacrificed for model precision. We also compare the two models and discover that deterministic PDA are more efficient, although stack-deterministic PDA require less memory.*

1. Introduction

A typical *host-based intrusion detection system (HIDS)* monitors execution of a process to identify potentially malicious behavior. An anomaly detection HIDS identifies variations from a preconstructed model of normal program behavior. Such a system interposes a monitor between a process and the operating system. All events (usually system calls) that flow from the program to the operating system are validated against the model. Events that do not conform to the model are rejected by the monitor. Figure 1 shows a typical HIDS architecture.

There are several techniques to construct the program model used in an anomaly detection HIDS. Learning-based techniques [4, 5, 8, 12, 14, 15, 22, 26] construct the program model by training on a set of execution traces. Sometimes a specification of the program provided by a domain expert is also used as a program model [11]. This paper focuses on program models constructed using static analysis [6,23,24]. In the context of static analysis, there is a trade-off between efficiency and precision. Non-deterministic finite automaton (NFA) models are efficient to operate, but introduce impossible paths because they do not model the call-return semantics of the program. Pushdown automaton (PDA) models eliminate impossible paths by incorporating the program's stack, but they are inefficient to operate. The inefficiency in the PDA models occurs because the stack activity of the program is hidden from the model and results in non-determinism. Therefore, the state space of the PDA model can become prohibitively large during operation. We call this the *curse of non-determinism*. This paper formally presents several techniques to handle this problem. Specifically, we make the following contributions:

- **Formal framework.** Formal models in intrusion detection research have received scant attention, and we address this shortcoming. Investigating formalisms drives the discovery of why certain program models do or do not exhibit reasonable performance. Our primary purpose is to formally analyze recently proposed models rather than to in-

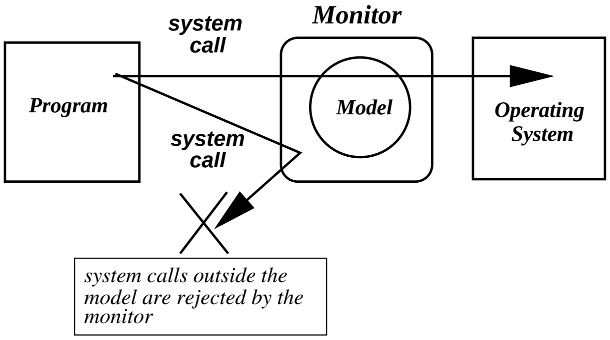

Figure 1. Architecture of a host-based intrusion detection system.

troduce all-new models. A commonality of these models is the exposure of process execution state beyond a simple system call stream. For example, Sekar *et al*. [19] proposed using program counter information. Feng *et al*. [2] and Giffin *et al*. [7] exposed the stack activity of a program. We show that a context-free language (CFL) is homomorphic to a deterministic CFL. The proof of this result is similar to that of Chomsky [1] and provides intuition about techniques that expose program state. Non-determinism in stack activity is the major factor contributing to the time and space complexity of operating PDA models. Motivated by this observation, we define a stack-deterministic PDA model in which the stack activity is deterministic. Section 3 discusses these formalisms.

- **Model determinizing techniques.** Techniques for determinizing the PDA models essentially incorporate additional program state (such as the program counter and stack activity) into the model. We describe two techniques incorporating additional state of the program. In the *observational technique*, the monitor extracts the relevant information from the program. For example, the monitor can extract information about the stack activity of the program by walking the call stack. Our VPStatic model, a statically-constructed variant of the VtPath model [2], implements this technique. The rewriting or *instrumentation technique* transforms the program to introduce additional code that exposes program state. For example, additional system calls introduced before a call to function `f` indicate to the model that a call to `f` is about to happen. Our recent Dyck model implements this approach [7]. We also compare the observational and rewriting approaches to determinizing the PDA model. Sections 4 and 5 present the two models.
- **Evaluation.** Our results show that the formalisms of deterministic and stack-deterministic push-down automata enable construction of context-sensitive program models suitable for online security monitoring. The VPStatic automaton operation slows execution of our test programs by 0% to 17%, although the unoptimized stack walking algorithm adds up to 80% additional overhead. The Dyck model is slightly less efficient due to state non-determinism, with overheads of 8% to 135%. However, the Dyck model has a compact representation, requiring only 38% to 49% more memory for program instrumentation and the state machine. These results vindicate context sensitive models, showing that reasonable efficiency needs not be sacrificed for model precision. Complete results are given in Section 6.

2. Related Work

Significant intrusion detection systems research has focused upon static and dynamic analysis techniques to automatically generate program models. Wagner *et al*. produced models via static analysis of C source code [23, 24]. They described the precise *abstract stack* model, which is a non-deterministic pushdown automaton (PDA). Due to the stack state maintained in a PDA, this model captured the precise call and return behavior of function calls. Unfortunately, runtime automaton operation in the monitor was prohibitively high for some programs, reaching several tens of minutes per transaction. We observed similar results with PDA models extracted using static binary analysis of SPARC executables [6]. Both papers concluded that imprecise, context-insensitive models must be used for reasonable performance.

However, only context-sensitive models, such as the PDA, can detect the impossible path exploits described by Wagner *et al*. [23,24]. Such attacks force control flow to enter a function from one call site but to return to a different call site, presumably in a portion of the program code suitable for the attack. A context-sensitive model detects this illicit control flow by modeling the state of a program's call stack.

Our experience indicates that severe non-determinism in this stack state is the major contributing factor to the time and space complexity of PDA operation. The Dyck model proved this: by eliminating non-determinism on stack transitions, a context-sensitive model could be efficiently operated [7]. This paper formalizes the Dyck model and proves that it is a stack deterministic PDA. We further improve the model by eliminating the additional system calls required by the previous model construction. Our VPStatic model goes further. It is a fully deterministic PDA and requires no modifications to the original, analyzed binary.

Dynamic analysis extends the original work of Forrest *et al.* [3] and constructs a program model based upon observed system call traces from numerous training runs [4,5,8,12,14,15,22,26,27]. Sekar *et al.* showed that learning a deterministic automaton is possible by associating each system call with its corresponding program counter [19]. This model does not monitor context information and may miss attacks due to this imprecision. It may also miss attacks on dynamically linked libraries due to its oversimplified handling of dynamic objects.

Our previous VtPath model improved the precision of dynamically constructed models [2]. This model additionally monitors return addresses on the call stack. Our VPStatic model is a natural extension of VtPath constructed using static analysis techniques. Again, we add formalism to the previous work. The VtPath model calculates an adhoc virtual path from the call stacks of two adjacent system calls and verifies the validity of that path. VPStatic is a provably deterministic PDA. The use of an automaton localizes transitions to states, making VPStatic more precise. Moreover, the VPStatic model does not suffer the false alarms of VtPaths due to its conservative static analysis.

Others have pioneered work outside of static and dynamic analysis. These approaches monitor execution based upon specifications of system calls [10] or of expected program behavior [11]. When provided by a domain expert, these specifications can likely enhance the quality of automatically generated models.

The VPStatic model has an *anomaly recovery* property not considered by previous approaches. After an anomaly occurs, we can still uniquely determine the expected state and stack context for the next valid system call by monitoring its program counter and call stack. Thus, we can continue to operate the automaton and potentially detect more attacks such as a root-level exploit following a probe. This also allows for a greater variety of security policies by enabling the system to fail an anomalous system call and continue execution rather than terminate the program. For example, a monitor that terminates a network daemon after an anomalous system call could be used for a denial of service attack. We can instead prevent just the anomalous call and allow process execution and monitoring to continue.

3. Formal Models

We begin by formally describing pushdown automata, deterministic pushdown automata, and stack-deterministic pushdown automata. These finite state machines are the underlying constructs of our program models used for intrusion detection.

Definition 1 *[PDA and DPDA]*

A pushdown automaton *(*PDA*) P is 7-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where Q is the set of states, Σ is the input alphabet, Γ is the stack alphabet, δ is the transition relation mapping $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times (\Gamma \cup \{\epsilon\})^\star$, $q_0 \in Q$ is the unique initial state, $z_0 \in \Gamma$ is the initial stack* start symbol*, and $F \subseteq Q$ is the set of accepting states. There are three types of transitions in δ:*

1. *(*Input consumption or ϵ transition*): $(p, z) \in \delta\,(q, a, z)$ where $a \in \Sigma \cup \{\epsilon\}$.*
   The top of the stack is z and stack contents do not change. If $a = \epsilon$, then this represents a transition from q to p that consumes no input. If $a \in \Sigma$ and P is in state q, then consume input a and move to state p.
2. *(*Push transition*; pushes z' onto the stack): $(p, zz') \in \delta\,(q, a, z)$ where $a \in \Sigma \cup \{\epsilon\}$.*
   The explanation is the same as (1), but now z' is pushed onto the stack.
3. *(*Pop transition*; pops z from stack): $(p, \epsilon) \in \delta\,(q, a, z)$ where $a \in \Sigma \cup \{\epsilon\}$.*
   The explanation is the same as (1), but now z is popped from the stack.

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is called deterministic *if the transition relation δ satisfies the following conditions [9]:*

- **(Condition 1):** *For all $q \in Q$ and $z \in \Gamma$, whenever $\delta(q, \epsilon, z)$ is nonempty, then $\delta(q, a, z)$ is empty for all $a \in \Sigma$.*
- **(Condition 2):** *For all q in Q, $a \in \Sigma \cup \{\epsilon\}$ and $z \in \Gamma$, $\delta(q, a, z)$ contains at most one element.*

A deterministic PDA is abbreviated as DPDA.

Our definition allows only one stack symbol to be pushed onto or popped from the stack. The most general definition of a PDA (as found in [9]) allows more than one stack symbol to be pushed on the stack. However, it is easy to see that a PDA P (according to the general definition) can always be converted into a PDA which conforms to our definition (the construction essentially transforms pushing many symbols on the stack into a sequence of pushes of one symbol.)

Given a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, a *configuration* c is a tuple (q, γ), where $q \in Q$ is the current state and

γ is a string of stack symbols representing the stack contents. Given two configurations c and c' and $a \in \Sigma$, we say that $c \Rightarrow_P^a c'$ if c is transformed into c' by a sequence of transitions of the PDA P while consuming input a. The relation $\Rightarrow_P^a$ can be extended to words $w \in \Sigma^\star$, i.e., given two configurations c and c' and $w \in \Sigma^\star$, $c \Rightarrow_P^w c'$ if c is transformed into c' by a sequence of transitions of the PDA P while consuming input from string w. When P is clear from the context, we simply write $\Rightarrow^w$ instead of $\Rightarrow_P^w$. The language $L(P)$ accepted by $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is defined as

$$\{w|(q_0, z_0) \Rightarrow_P^w (p, \gamma) \text{ for some } p \in F \text{ and } \gamma \in \Gamma^\star\}$$

PDAs accept context free languages (CFL). If a language L is accepted by DPDA, it is called a *deterministic context free language* or *DCFL*. Theorem 1 proves that every CFL L is homomorphic [9] to a DCFL L'. Moreover, the proof of the theorem gives a procedure for determinizing a PDA by expanding the input alphabet. This proof is similar to that of Chomsky [1].

Theorem 1 *Let L be a CFL. There exists a DCFL L_D and a homomorphism h such that $h(L_D) = L$.*

Proof: *Let $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be a PDA accepting L. We will construct a new input alphabet Σ_D. There are three types of symbols in Σ_D.*

- **Input:** *For each $a \in \Sigma \cup \{\epsilon\}$ and $p \in Q$, there is an input symbol $e_{a,p}$. The input symbol $e_{a,p}$ represents consuming input a and transitioning to state p.*
- **Push:** *For each $a \in \Sigma \cup \{\epsilon\}$, $p \in Q$ and $z \in \Gamma$, there is an input symbol $f_{a,p,z}$. The input symbol $f_{a,p,z}$ represents consuming input a, pushing z on to the stack, and transitioning to state p.*
- **Pop:** *For each $a \in \Sigma \cup \{\epsilon\}$, $p \in Q$ and $z \in \Gamma$, there is an input symbol $g_{a,p,z}$. The input symbol $g_{a,p,z}$ represents consuming input a, popping z from the stack, and transitioning to state p.*

Next, we will construct a DPDA $P_D = (Q, \Sigma_D, \Gamma, \delta_D, q_0, z_0, F)$. Notice that the only components different between P and P_D are the input alphabet and the transition relation. The transition relation for the DPDA P_D is defined as follows:

- *For each transition $(p, z) \in \delta(q, a, z)$ in P, we have the transition $\delta_D(q, e_{a,p}, z) = \{(p, z)\}$.*
- *For each transition $(p, zz') \in \delta(q, a, z)$ in P, we have the transition $\delta_D(q, f_{a,p,z'}, z) = \{(p, zz')\}$.*
- *For each transition $(p, \epsilon) \in \delta(q, a, z)$ in P, we have the transition $\delta_D(q, g_{a,p,z}, z) = \{(p, \epsilon)\}$.*

It is easy to see that P_D is deterministic. Consider the following homomorphism h:

$$\begin{aligned} h(e_{a,p}) &= a \\ h(f_{a,p,z}) &= a \\ h(g_{a,p,z}) &= a \end{aligned}$$

Let $L(P_D)$ be the language accepted by the DPDA P_D. Then $h(L(P_D)) = L(P) = L$. □

The construction used in the proof of Theorem 1 expands the input alphabet by exposing the stack operations and the target state of the transition. For example, the input symbol $f_{a,p,z}$ indicates to the DPDA P_D that it should consume input a, push z on the stack, and transition to state p. Suppose that a PDA P models a program Pr. In this case, the DPDA P_D models the program Pr, where internal state of the program Pr (such as stack activity) is exposed. In other words, exposing program state corresponds to the input alphabet expansion used in Theorem 1.

3.1. Intrusion Detection using PDAs and DPDAs

In *model-based intrusion detection*, one constructs a model $M(Pr)$ of a program Pr (see Figure 1). Pr generates a sequence of symbols (usually a sequence of system calls). After receiving a symbol a from the program, the model $M(Pr)$ determines whether there exist transitions on symbol a. If there *does not* exist a transition on the input symbol a, the monitor reports an intrusion. Otherwise, the model processes symbol a and updates its state.

PDA models. Suppose that the model $M(Pr)$ is a PDA $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. The state of the model is the set of configurations. The model's initial state is $\{(q_0, z_0)\}$. Let C be the set of possible configurations for $M(Pr)$ after processing a sequence of symbols w from Pr. Suppose the next symbol that Pr generates is a. The new state of the program is $succ(C, a)$, which represents all configurations that result from configurations in C after processing input a. Formally, $succ(C, a)$ is defined as $\{c' | \exists c \in C.c \Rightarrow^a c'\}$. If $succ(C, a)$ is empty, the monitor reports an intrusion. Otherwise, the new state of the model $M(Pr)$ is $succ(C, a)$ and the processing continues.

In general, the state of the model can be infinite. For example, suppose the model is in the state $C = \{(p, z)\}$ and receives a symbol a. Assume that the model has the following transitions:

$$\delta(p, \epsilon, z) = \{(p, zz)\} \quad (1)$$
$$\delta(p, a, z) = \{(q, \epsilon)\} \quad (2)$$

It is easy to see that $succ(C, a)$ is the infinite set $\{(q, z^i)\ i \geq 0\}$. Notice that the infiniteness arises from rule 1, which corresponds to left recursion in a program. However, it turns out that the state of the model (which is a set of configurations) is regular and can be represented as a finite-state automaton [18, 23]. The time and space complexity of updating the state of the model after receiving a symbol is unfortunately cubic in the size of the model. Wagner and Dean concluded that operating a PDA model for intrusion detection was prohibitively expensive [23, 24].

DPDA models. Suppose the model $M(Pr)$ is a DPDA. Given an input symbol $a \in \Sigma$, a configuration c, and a DPDA $M(Pr)$, there exists *at most one configuration* c' such that $c \Rightarrow^a c'$. Therefore, it is easy to see that during monitoring the set of configurations C has at most one configuration. The time and space complexity of updating the state of the model after receiving a symbol is $O(1)$.

Stack-deterministic PDA models. In our experience, non-determinism in stack activity is the major contributing factor to the time and space complexity of operating PDA models. This motivates our definition of a stack-deterministic PDA model, which allows non-determinism but requires the state of the stack be left unchanged at points of non-determinism. Formally, a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is called a *stack-deterministic PDA* or *sDPDA* if it satisfies the following two conditions:

- **(Condition 1):** *No stack activity on ϵ-transitions.* There is no push or pop transition $\delta(q, a, z)$ such that $a = \epsilon$.
- **(Condition 2):** *Stack activity only depends upon the input symbol and the top of the stack.* For all $a \in \Sigma$ and $z \in \Gamma$, there *does not exist* two states q_1 and q_2 (not necessarily different), such that

$$\begin{array}{rcl} (p_1, w_1) & \in & \delta(q_1, a, z) \ , \\ (p_2, w_2) & \in & \delta(q_2, a, z) \ , \end{array}$$

  where $w_1 \neq w_2$.

Assume that we use an sDPDA model $M(Pr)$ of a program Pr for intrusion detection. Let C be the set of configurations obtained after processing a sequence of symbols w. From the two conditions given above, all configurations in C must have the same stack. Formally, $C \in 2^Q \times \Gamma^\star$, where Q and Γ are the set of states and stack alphabets for the model $M(Pr)$. Since the size of C can be at most $n = |Q|$, the time and space complexity for processing a new symbol a is $O(n)$. If a language L is accepted by sDPDA, it is called a *stack-deterministic context free language* or *sDCFL*. Theorem 3 in Appendix A proves that the language accepted by a sDPDA is a DCFL. Therefore, an sDPDA is not fundamentally more powerful than a DPDA.

Theorem 2 *Let L be a CFL. There exists a sDCFL L_D and a homomorphism h such that $h(L_D) = L$.*

Proof: *Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA accepting L. We will construct a new set of input symbols Σ_{sD}. There are three types of symbols in Σ_{sD}.*

- **Input:** *This is simply the input alphabet Σ of the PDA P.*
- **Push:** *For each $a \in \Sigma \cup \{\epsilon\}$ and $z \in \Gamma$, there is an input symbol $f_{a,z}$. The input symbol $f_{a,z}$ represents consuming input a and pushing z onto the stack.*
- **Pop:** *For each $a \in \Sigma \cup \{\epsilon\}$ and $z \in \Gamma$, there is an input symbol $g_{a,z}$. The input symbol $g_{a,z}$ represents consuming input a and popping z from the stack.*

Next, we will construct a sDPDA $P_{sD} = (Q, \Sigma_{sD}, \Gamma, \delta_{sD}, q_0, z_0, F)$. Notice that the only components different between P and P_{sD} are the input alphabet and the transition relation. The transition relation for the sDPDA P_{sD} is defined as follows:

- *For each transition $(p, z) \in \delta\,(q, a, z)$ in P, we have the transition $(p, z) \in \delta_{sD}\,(q, a, z)$.*
- *For each transition $(p, zz') \in \delta\,(q, a, z)$ in P, we have the transition $(p, zz') \in \delta_{sD}\,(q, f_{a,z'}, z)$.*
- *For each transition $(p, \epsilon) \in \delta\,(q, a, z)$ in P, we have the transition $(p, \epsilon) \in \delta_{sD}\,(q, g_{a,z}, z)$.*

It is easy to see that P_{sD} is stack-deterministic. Consider the following homomorphism h:

$$\begin{array}{rcl} h(a) & = & a \\ h(f_{a,z}) & = & a \\ h(g_{a,z}) & = & a \end{array}$$

Let $L(P_{sD})$ be the language accepted by P_{sD}. Then $h(L(P_{sD})) = L(P) = L$. □

The construction used in the proof of Theorem 2 expands the input alphabet by exposing the stack operations. For example, the input alphabet $f_{a,z}$ indicates to the sDPDA P_D that it should consume input a and push z onto the stack. Recall that in the proof of Theorem 1 we expanded the input alphabet to expose the stack activity and the target of the transition, e.g., $f_{a,p,z}$ indicated that it should consume input a, push z on the stack, and transition to state p. In constructing an sDPDA, we exposed the stack activity of the PDA but not the target of the transition.

Table 1 summarizes the time and space complexity of processing a new symbol for the three models. The size of input alphabet for the three models is also shown. From Theorems 1 and 2 it is clear that the size of the input alphabets for DPDA and sDPDA models is larger than for the corresponding PDA.

Model	*Time complexity*	*Space complexity*	*Input alphabet size*
PDA	$O(nm^2)$	$O(nm^2)$	k
DPDA	$O(1)$	$O(1)$	$\Theta(knr)$
sDPDA	$O(n)$	$O(n)$	$\Theta(kr)$

Table 1. Time and space complexities for processing an input symbol with various models. The number of states and transitions in the model are denoted by n and m respectively. The size of the input and stack alphabets in the PDA are denoted by k and r respectively.

3.2. Connection to Existing Techniques

Several authors have proposed exposing program state to improve the precision of the models. For example, Sekar *et al*. [19] propose using program counter information. This is equivalent to expanding the input alphabet to expose the target of the transition. Giffin *et al*. [7] and Feng *et al*. [2] expose the stack activity of a program. In our context, this is equivalent to expanding the input alphabet by exposing the stack activity (this is very similar to the homomorphism demonstrated in the proof of Theorem 2). Therefore, the formal framework of PDA, DPDA, sDPDA, and homomorphisms provides a systematic way of understanding and evaluating techniques that expose additional program state.

4. The VPStatic Model: Determinizing via Stack Exposure

The VPStatic model is a statically-constructed variant of the context-sensitive VtPath model [2]. Like its dynamic counterpart, it uses stackwalks during execution to determine the call stack state of the monitored process. Combined with program counter monitoring, this produces the extra symbols necessary to fully determinize the model.

4.1. Model Generation by Static Analysis

The VPStatic model is generated by statically analyzing the binary executable of a program. We first introduce notation. There is a function entry state $Entry(f)$ and exit state $Exit(f)$ for each function f in the executable, system call state S for each system call instruction, and call site entry state C (state right before the call) and exit state C' (state right after the return) for each function call site. $Addr(S)$, $Addr(C)$, and $Addr(C')$ denote the address of the corresponding system call or function call instruction ($Addr(C) = Addr(C')$). $Func(a)$ is the function containing the instruction at address a.

```
                                        States
char* filename;
pid_t[2] pid;
int prepare(int index) {                Entry(prepare)
  char buf[20];
  pid[index] = getpid();                S_getpid
  strcpy(buf, filename);
  return open(buf, O_RDWR);             S_open
}                                       Exit(prepare)
void action() {                         Entry(action)
  uid_t uid = getuid();                 S_getuid
  int handle;
  if (uid != 0) {
    handle = prepare(1);                C1, C1'
    read(handle, ...);                  S_read
  } else {
    handle = prepare(0);                C0, C0'
    write(handle, ...);                 S_write
  }
  close(handle);                        S_close
}                                       Exit(action)
```

Figure 2. A simple code fragment example.

We use a simple program fragment, shown in Figure 2, as a running example. The automaton and the left side list of transitions in Figure 3 describe a non-deterministic PDA for the example program that is quite similar to the callgraph model [24]. As for the callgraph model, system call numbers are the only observed inputs to simulate the automaton. We use "none" (or more commonly ε) as a place holder when transitions are not associated with any system call. This PDA is non-deterministic since we have not exposed stack activities and targets of transitions. To make the PDA deterministic, we extract address information from the binary to expose internal state. The automaton and the right side list of transitions in Figure 3 describe the final DPDA.

The input symbols of the DPDA have the forms in the proof of Theorem 1, with slight modifications. Namely, for system call and call site states, we use $Addr(p)$ instead of p to expose the state, since the address information is what we can extract from program counter and call stack when dynamically monitoring program executions. For example, $g(a, Addr(p), z)$ or $g(a, p, z)$ means the automaton consumes the input symbol, pops z from the stack, and transitions to state p. This is equivalent to the transition $g_{a,p,z}$ used in the proof of Theorem 1. Other symbols can be similarly explained. All three forms of input symbols $e(\ldots)$, $f(\ldots)$ and $g(\ldots)$ appear in Figure 3. The formal models section proved this pushdown automaton is deterministic. A detailed description of the model is in Appendix B.

4.2. Online Detection by Dynamic Monitoring

After the profile is generated, we can simulate the automaton during online program monitoring. When each sys-

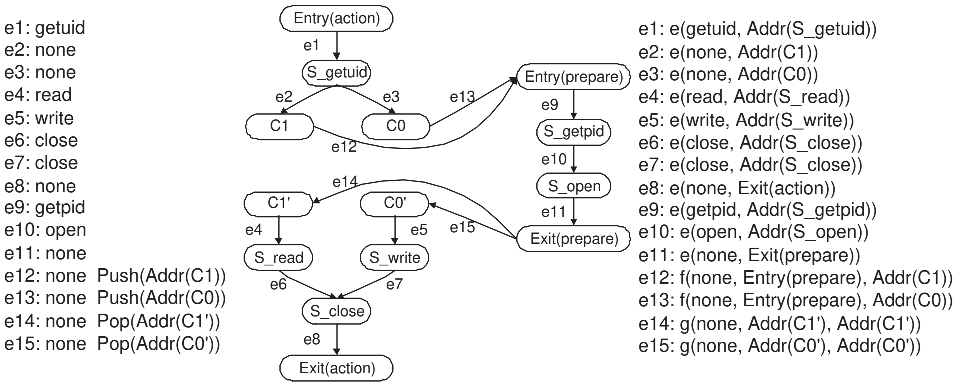

Figure 3. The PDA and VPStatic DPDA generated for the code example.

$$
\begin{array}{rr}
e(none, Exit(Func(a_{m+1}))) & \\
g(none, a_m, a_m) & \\
e(none, Exit(Func(a_m))) & \\
g(none, a_{m-1}, a_{m-1}) & \\
\vdots & \\
e(none, Exit(Func(a_{l+1}))) & \\
g(none, a_l, a_l) & (3) \\
e(none, b_l) & (4) \\
f(none, Entry(Func(b_{l+1})), b_l) & \\
e(none, b_{l+1}) & \\
f(none, Entry(Func(b_{l+2})), b_{l+1}) & \\
\vdots & \\
e(none, b_n) & \\
f(none, Entry(Func(b_{n+1})), b_n) & (5) \\
e(s_B, b_{n+1}) & (6)
\end{array}
$$

Figure 4. VPStatic input symbol sequence generated for system call S_B.

tem call is made, we extract all the call site addresses for the functions that have not returned yet into a *virtual stack list (VSL)*, ordered from the outermost function to the innermost function. The definition of VSL is similar to that in [2].

Assume $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_n$ are the virtual stack lists of the last and the current system calls, respectively. Also, assume s_B is the current system call and b_{n+1} is its address (the current program counter), and s_A and a_{m+1} are the last system call and its address, respectively. Suppose l is the first index for A and B so that the corresponding items are not equal, namely, $a_i = b_i$ for $i = 1, 2, \ldots, l-1$, and $a_l \neq b_l$. For the current system call, we generate a sequence of input symbols and feed them to the automaton one by one. The input symbol sequence generated is shown in Figure 4.

For example, assume an ordinary user (not root) executes the example program and runs to the `getpid` line in Figure 2. The virtual stack list A here should look like "*prefix*, $Addr(C1)$", where *prefix* is a sequence of addresses corresponding to the functions that lead to `action`. The system call S_A is `getpid`. If the program executes to `open`, the virtual stack list B here is the same as A since the call stack does not change, and system call S_B is `open`. So from Figure 4, the symbol sequence generated is $e(open, Addr(S_{open}))$, which successfully leads the automaton to the next state S_{open}. However, if an attacker overflows a buffer using `strcpy`, she could change the return address of `prepare` to $Addr(C0)$, to gain unauthorized write access to the file after `prepare` returns. In that case, the virtual stack list B changes to "*prefix*, $Addr(C0)$". Since $Addr(C0) \neq Addr(C1)$, from Figure 4, the symbol sequence generated will be:

$$
\begin{array}{r}
e(none, Exit(prepare)) \\
g(none, Addr(C1), Addr(C1)) \\
e(none, Addr(C0)) \\
f(none, Entry(prepare), Addr(C0)) \\
e(open, Addr(S_{open}))
\end{array} \quad (7)
$$

However, since state S_{getpid} does not have a transition associated with $e(none, Exit(prepare))$, an alarm will be triggered and the intrusion is detected.

After all input symbols generated for a system call are processed, the current state should be the state corresponding to the system call, and the current automaton stack context is just the VSL of this system call. Namely, the current state and stack context can be uniquely decided for a valid system call. If there is no corresponding transition to follow for an input symbol, then anomalous execution indicative of an intrusion attempt has occurred.

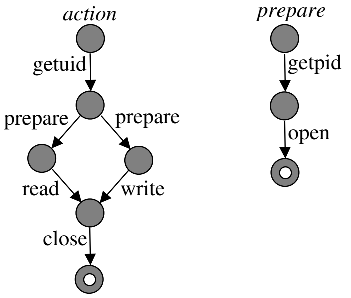

Figure 5. Local Dyck models.

5. The Dyck Model: Determinizing via Instrumentation

As shown in Section 3, adding stack determinism to a PDA requires additional alphabet symbols to make stack-modifying transitions deterministic. Statically constructed program models use the PDA stack to model the running process's call stack. Stack operations then occur at function call sites and returns. The *Dyck model* [7] uses binary rewriting to insert code before and after each function call site to generate the extra symbols needed for stack-determinism.

5.1. Static Model Construction

The Dyck static analyzer reads a binary program image and produces both an Dyck model and an instrumented version of the binary. This requires four steps:

1. For each function, construct a control flow graph (CFG).
2. Convert each CFG into a *local model*: a non-deterministic finite automaton that accepts all sequences of function calls and kernel traps that the function could generate under correct execution.
3. Classify function calls and insert code around function call sites to generate symbols necessary for stack-determinism. This instrumentation adds new events into the call stream, so we update local models to match.
4. Combine the collection of modified local models into a single sDPDA modeling the entire rewritten program.

Recall that Figure 2 shows code for two example functions, `prepare` and `action`. Although we show C source code for readability, we analyze SPARC binary code.

We convert each function's CFG into a local model. This is straightforward: a CFG is already a non-deterministic finite state machine with all edges unlabeled. If a basic block contains a user call or kernel trap site, we label all outgoing

```
void action () {
  uid_t uid = getuid();
  int handle;
  if (uid != 0) {
    precall(A);
    handle = prepare(1);
    postcall(A);
    read(handle, ...);
  } else {
    precall(B);
    handle = prepare(0);
    postcall(B);
    write(handle, ...);
  }
  close(handle);
}
```

Figure 6. Example Code With Dyck Instrumentation. Inserted code appears in boldface.

edges of that block with the call name. We label all other edges ϵ and convert all basic blocks into automaton states. The ϵ-reduced and minimized local automata for the example code are shown in Figure 5. Appendix C.1 gives the formal definition of a local model.

Next, we add edges to the local models around function call transitions that model the call stack changes occurring at runtime. An edge before each call transition pushes a unique identifier onto the PDA stack kept in the runtime monitor; an edge after the call pops that identifier off. Each call site has a unique push and pop symbol, so the monitor can differentiate between different call sites to the same function. The NFA local models are now PDAs.

To add stack-determinism to these PDA models, we must add symbols to the event stream that distinguish each stack operation. The analyzer rewrites the binary image of the program by inserting a *history stack* into the program's data space and adding code immediately before and after each call site. The history stack records stack changes occurring since the last kernel trap. *Precall* code before call site A pushes the symbol $f_{\epsilon,A}$ onto the history stack. If the call generates a kernel trap before returning, then the monitor reads all collected symbols from the history stack to identify the execution path followed in the program. If the call returns without generating a kernel trap, then the *postcall* code pops $f_{\epsilon,A}$ from the history stack and discards it. Otherwise, it adds the symbol $g_{\epsilon,A}$ to the history stack. Figure 6 shows the rewritten code for `action` with instrumented call sites to `prepare`.

Adding code instrumentation at recursive call sites has potentially high runtime cost. We add neither stack transitions to the local models nor code to the binary image around call sites that may recurse.

Lastly, we compose the collection of modified local automata at points of function calls to form the global model

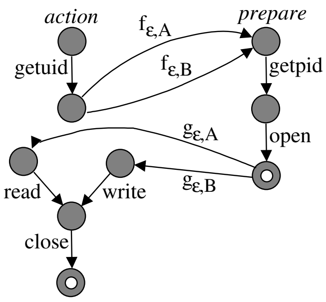

Figure 7. Dyck model.

of the entire program. The analyzer replaces each function call transition with ϵ-edges entering and returning from the model of the target function. Figure 7 shows the completed Dyck model for the example functions. Note the similarity to the VPStatic model described earlier. Here, the input symbol $f_{\epsilon,A}$ additionally pushes the identifier A onto the PDA stack. The symbol $g_{\epsilon,A}$ is an input symbol that pops A. Appendix C.1 formally defines the model using language theory. The Dyck model is an sDPDA (see Appendix C.2).

5.2. Runtime Monitoring

The user executes the rewritten binary in her security-critical environment, with the *runtime monitor* tracing its execution at system calls. The monitor enforces the model, guaranteeing that process execution does not deviate from the possible sequences of system call streams.

Dyck model operation is more straightforward than VPStatic operation. Although the model is a PDA, the monitor keeps only one PDA stack due to stack-determinism. When the traced process generates a kernel trap, the monitor reads all saved symbols from the process's history stack. Each symbol is an input to the automaton that modifies the stack state and corresponds to a return from or a call to another function. These symbols are equivalent to the virtual path symbols calculated from stack walks in the VPStatic model. The monitor then processes the kernel trap symbol, permitting execution only if the symbol has a valid transition in the model.

6. Performance Measurements

To compare the performance of the VPStatic and Dyck models, we measured two costs of execution monitoring. First, we measured the increase in execution time when monitoring. Second, we calculated the increased memory use due to the program models and Dyck instrumentation.

We analyzed performance for three test programs. Our tools currently build models only for statically-linked programs. As a result, the set of test programs is not representative of those with greatest security concern, although they do contain a mix of computation-intensive and syscall-intensive programs. Table 2 lists the three programs and workloads and statistics for each. `htzipd` is a proprietary implementation of `httpd` which is the only httpd implementation we successfully compiled statically under Solaris.

Execution time overheads are calculated by subtracting a base execution time from monitored execution time. All times are averaged over several runs. Execution times do not include setup time in the monitor during which the program model is read from disk. The current implementation of both the VPStatic and Dyck monitors execute in user space and detect system call events via Solaris process tracing. To better evaluate the cost of operating each model, the base execution time is measured with process tracing enabled. At each system call stop, the monitor does nothing but resume the execution of the stopped process. The difference between base time and monitored time then captures exactly the overhead of model operation.

We calculated memory usage similarly. The value of interest is the increase in state required for each process. In particular, the static code of the monitoring process is of little meaning as it may be shared among all audited processes. For the VPStatic model, we compute the per process state by taking the difference between memory use with full auditing enabled and with an empty profile loaded with auditing disabled. The memory used by the Dyck model also includes the cost of binary code inserted into the original application.

This section does not include measurements with the previously used average branching factor metric [23]. This metric is poorly suited for measurements of context-sensitive languages as stack transitions entering system call wrapper functions obscure the reachable system calls. Lacking an appropriate metric, we rely instead upon our theoretical discussions in the previous sections as an evaluation of the strength of our models. Strength metrics may be applied in the future if new research develops reasonable measurement algorithms.

6.1. Execution Time Overhead Results

Table 3 contains execution time overheads for the VPStatic and Dyck models. Base execution times are presented twice because differences in monitor implementations result in somewhat different base times. Due to the high cost of the stack walk operation in the VPStatic model, we separate the model's runtime into two components: the time to operate the automaton and the time to perform the stack walk. The Dyck model does not walk the call stack, so no such separation is presented.

Program	*Workload*	*Instructions*	*Functions*	*Call Sites*
htzipd	Service 500 client requests, transferring 151.7 MB in total.	110,096	1455	6928
gzip	Compress a 23.5 MB tar file.	57,271	884	2844
cat	Concatenate 38 files totalling 500 MB to a file.	52,601	838	2728

Table 2. Test programs, workloads, and statistics.

		VPStatic					*Dyck*		
Program	*Untraced*	*Base*	*Automaton*	*%*	*Stackwalk*	*%*	*Base*	*Automaton*	*%*
htzipd	16.66	22.06	3.80	17	17.54	80	20.50	27.72	135
gzip	13.72	14.69	0.03	0	0.17	1	13.99	1.17	8
cat	46.20	59.14	2.56	4	15.81	28	54.84	30.60	56

Table 3. Model execution times in seconds. Base execution time includes system call tracing without automaton operation. Percentages compare against base execution.

Program	*VPStatic*	*Dyck*
htzipd	225	300
gzip	389	415
cat	170	289

Table 4. Average system call verification time, in microseconds.

Table 4 shows the average monitor execution time, in microseconds, per system call event received. Each system call requires the monitor to update its calling context information and to verify that the system call is a valid operation in the program model. The times to perform these operations remained relatively constant even as the number of stack symbols read from the monitored process changed, although outlying points did occur.

The tables show two interesting results. First, these deterministic or stack-deterministic models are efficient to operate. Automaton operations in the deterministic VPStatic model are extremely fast. Second, the VPStatic model is more efficient to operate than the Dyck model.

This second result occurs for two reasons. First, it illustrates the operational differences between deterministic and stack-deterministic automata. The DPDA used with the VPStatic model operates in constant time, but the sDPDA underlying the Dyck model requires linear time operation (Table 1). This effect is clearly visible in the respective runtimes of the two models. Second, the Dyck model has additional execution at many function call sites due to the injected code. This cost arises even if the process follows an execution path that does not generate system calls. The VPStatic model incurs monitoring cost only at system call events.

6.2. Memory Use Overhead Results

Table 5 presents the memory needs of execution monitoring for the two models. We divide the memory costs of the Dyck model into the cost of the current rewriting infrastructure, which doubles the size of each program's code segment, and the cost of our code insertions and state machine representation. The infrastructure cost is excessive, but could be significantly reduced by shifting to a more efficient rewriter.

The VPStatic state machine cost is greater than the corresponding Dyck models. Again, this highlights differences between DPDA and sDPDA models. An automaton allowing non-determinism in state transitions naturally has a more compact representation. Hence, the Dyck model will produce smaller automaton structures than the VPStatic model. Moreover, we have not yet optimized the VPStatic model size. For example, we could remove all the function entry and exit nodes using techniques similar to the automaton reduction used for Dyck model. We kept the original format of the model since it is a recent development and is conceptually clearer this way.

6.3. Discussion

We draw two primary conclusions from this work. First, the formalisms of deterministic and stack-deterministic push-down automata result in highly accurate and highly efficient program models. Non-deterministic context-sensitive models produced overheads orders of magnitude worse than base execution [6, 23, 24] and would never be suitable for real-world operation. Our automaton operation overheads, while not yet as low as we would like, show that context-sensitivity and precise program models need not be sacrificed for performance.

Second, the differences in these models suggests that hybridization of the two construction and monitoring tech-

		VPStatic		Dyck				
Program	*Unmonitored*	*State Machine*	*%*	*Infrastructure*	*%*	*Instrumentation*	*State Machine*	*%*
htzipd	568	1040	183	504	89	48	168	38
gzip	600	560	93	288	48	48	232	47
cat	280	544	194	272	97	32	104	49

Table 5. Memory use in KB due to monitoring. Percentages are increases over unmonitored execution.

niques may be beneficial. The Dyck model produces no context information at points of recursion or dynamic linking to non-instrumented binaries. The VPStatic model can identify this missing information by inspecting existing program state. If instrumented libraries are available, the Dyck model can more easily use these libraries at runtime as memory offsets of return addresses are not an issue. The Dyck model can also successfully reveal context information in optimized binaries where stack walking may be difficult or impossible. On the other hand, binary rewriting does occasionally fail. We can then rely on the stack walk technique to recover state information. Likewise, we may wish to limit instrumentation to some set of critical program points and rely upon stack walking elsewhere. A hybrid model would combine both state recovery mechanisms to capture the complete context of a system call. The hybrid would gain from the strengths of both models while minimizing the drawbacks of each.

7. Limitations

Although our approaches produce more sensitive and more accurate models than other approaches, there are still limitations. It is well documented [16, 17, 25] that attackers can exploit weaknesses and limitations of intrusion detection models to avoid detection. Short of complete instrumentation, which amounts to essentially interpreting the program, our statically-generated models do not have complete information about the state of the executing program. An attacker can exploit incomplete information in the model to evade the HIDS.

7.1. Incomplete Sensitivity

Models discussed in this paper incorporate information about the stack activity of the program. Thus, our models are context sensitive. However, since our model does not track predicates used in branches, they are neither flow nor path sensitive. This incompleteness can result in our model allowing extraneous behavior. For example, consider the following code fragment:

```
char *str, *user;

str = (char *) calloc (...);
user = (char *) calloc (...);
...

if (strncmp (user, "admin", 5)) {
   sys_1 ();
} else {
   sys_2 ();
}
strcpy (str, someinput);
if (strncmp (user, "admin", 5)) {
   sys_3 ();
} else {
   sys_4 ();
}
```

There are two possible system call sequences for the code fragment:

`sys_1, sys_3`, and

`sys_2, sys_4`.

The sequences correspond to the predicate `strncmp(user,"admin",5)` being true or false respectively. Notice that the predicates used in both the branches are the same. However, since our models do not track the values of branch predicates, they will allow the following four sequences:

`sys_1, sys_3`,

`sys_1, sys_4`,

`sys_2, sys_3`, and

`sys_2, sys_4`.

An attacker can exploit this limitation to avoid detection. In the example given above, an attack uses a large `someinput` in `strcpy` to overflow `str` on the heap to change the value of `user`. If `user` is "`guest`" and the overflow in `strcpy` changes `user` to "`admin`", then the illegal sequence `sys_1, sys_4` is executed, which is accepted by our model and hence the attack is not detected.

7.2. Incomplete Set of Events

Events monitored by our model are system calls. As pointed out by Wagner and Soto [25] and Tan *et al.* [20,21], an attacker can evade detection if it generates sequence of events accepted by our model. We previously presented such an attack using the following code segment [2]:

```
...
f(); //f has no system calls, a buffer-overflow occurs
     // so that after f the program jumps to IP

if (regular_user) {
  return ();
}

IP: //super user privileges
  execve (``/bin/sh'');
```

An attack that uses a buffer overflow in `f` to force the program to jump to `IP` can (illegally) obtain a root shell. Without inserting code instrumentation before and after `f`, our models will miss this attack because there is no system call in the code segment between the call to `f` and `IP`. In other words, no matter how the program control flow is illegally modified within the code segment, there is no observable events to our models. However, when code instrumentation is added right before and after the call to `f`, the attack will be detected by our model.

An attack can also evade detection if it simply replaces the system call parameters [25]. We recover some arguments of system calls using static analysis, but there are several system calls where we have incomplete information about the arguments.

7.3. Playing Inside the Sandbox: Mimicry Attacks

We assume that attackers have complete knowledge about our model-construction algorithm. In *mimicry* attacks, an adversary transforms her attack in such a way that the resulting sequence is accepted by the detection model [25]. For example, the attacker can mimic the legal program behavior by generating (legal) system calls and inserting them in the original attack sequence. An attacker can use a different attack sequence that is semantically equivalent to the original attack sequence. These attacks are very serious and can easily evade simple detection models, such as the n-gram model proposed by [3]. However, incorporating additional information about the program in the model makes it difficult to mount mimicry attacks. Our models monitors information about system calls, program counter, and call stack. Therefore, to mount a successful mimicry attack an adversary is required to produce correct call stack and program counter information along with the sequence of system calls which is equivalent to the attack.

8. Conclusions

We formally described statically-constructed context-sensitive program models for host-based intrusion detection. Seeking to add efficiency to the precision of these models, we examined deterministic PDAs and introduced stack-deterministic PDAs. The proofs of language equivalence between the homomorphic image of a DCFL or sDCFL and a CFL give rise to monitoring techniques that make these models possible. The VPStatic model walks a process's call stack to harvest return addresses revealing context information enabling a deterministic model. Using program instrumentation, the Dyck model eliminates stack non-determinism. Experiments demonstrate that context-sensitivity and efficiency can coexist in these program models, benefiting all such intrusion detection systems.

Acknowledgments

We thank the anonymous referees for their useful comments. The members of the WiSA security group at Wisconsin provided ongoing feedback to this work. We also acknowledge Dr. Weibo Gong at the University of Massachusetts for discussion and support. Mr. Prahlad Fogla and Mr. Oleg Kolensnikov at Georgia Institute of Technology provided ongoing feedback to this work. We use EEL [13] to analyze SPARC binary code.

This work is supported in part by Army Research Office contract DAAD19-01-1-0610, Office of Naval Research grant N00014-01-1-0708, NSF grants CCR-0133629 and CCR-0208655, and Department of Energy grants DE-FG02-93ER25176 and DE-FG02-01ER25510. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

References

[1] N. Chomsky. Context-free grammars and pushdown storage. In *Quarterly Progress Report No. 65*, pages 187–194. Massachusetts Institute of Technology Research Laboratory of Electronics, April 1962.

[2] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2003.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, California, May 1996.

[4] T. Garvey and T. Lunt. Model-based intrusion detection. In *14th National Computer Security Conference (NCSC)*, Baltimore, Maryland, June 1991.

[5] A. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999.

[6] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, California, August 2002.

[7] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium (NDSS)*, San Diego, California, February 2004.

[8] S. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection system using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[9] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[10] K. Ilgun, R. Kemmerer, and P. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[11] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *10th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, December 1994.

[12] T. Lane and C. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2(3):295–331, August 1999.

[13] J. Larus and E. Schnarr. EEL: Machine independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, California, June 1995.

[14] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

[15] T. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *11th National Computer Security Conference (NCSC)*, Baltimore, Maryland, October 1988.

[16] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), December 1999.

[17] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc., January 1998. http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps.

[18] S. Schwoon. *Model-Checking Pushdown Systems*. Ph.D. dissertation, Technische Universität München, June 2002.

[19] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[20] K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID) 2002, LNCS #2516*, pages 54–73, Zurich, Switzerland, October 2002. Springer-Verlag.

[21] K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *5th International Workshop on Information Hiding, LNCS #2578*, Noordwijkerhout, Netherlands, October 2002. Springer-Verlag.

[22] H. Teng, K. Chen, and S.-Y. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1990.

[23] D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. Ph.D. dissertation, University of California at Berkeley, Fall 2000.

[24] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[25] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, November 2002.

[26] C. Warrander, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

[27] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Recent Advances in Intrusion Detection (RAID) 2000, LNCS #1907*, pages 110–129, Toulouse, France, October 2000. Springer-Verlag.

A. Proof for Section 3

Theorem 3 *The language L accepted by an sDPDA is a DCFL.*

Proof: *First, remove ϵ transitions. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be an sDPDA. Recall that an sDPDA does not have stack activity on ϵ-transitions. Given states q and q' and a stack symbol $z \in \Gamma$, we say that $(q, z) \Rightarrow_\epsilon (q', z)$ if there exists an ϵ-transition such that $(q', z) \in \delta(q, \epsilon, z)$. Let $\Rightarrow_\epsilon^\star$ be the reflexive and transitive closure of $\Rightarrow_\epsilon$. Notice that $\Rightarrow_\epsilon^\star$ can be computed in polynomial time using standard graph reachability algorithms. We will transform the transition relation δ of P to δ' to remove ϵ transitions. First, δ' will contain all the non-ϵ transitions $\delta(q, a, z)$. For each transition, $(q', z) \in \delta(q, a, z)$ (where $a \neq \epsilon$), $\delta'(q, a, z)$ contains all (q'', z) such that $(q', z) \Rightarrow_\epsilon^\star (q'', z)$.*

Second, remove state non-determinism. Due to the step given above, we can assume that the sDPDA P does not contain ϵ-transitions. Next we can remove the "state" non-determinism of P using the standard subset construction used in determinizing a NFA. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ be an sDPDA without ϵ-transitions. We will construct a DPDA $DP = (Q_1, \Sigma, \Gamma, \delta_1, q_0^1, z_0, F_1)$, where $Q_1 = 2^Q$, $q_0^1 = \{q_0\}$, F_1 are all subsets of Q such that $F_1 \cap F \neq \emptyset$, and δ_1 is defined as follows: $\delta_1(q_1, a, z)$ contains (q_2, z'), where q_2 is the following set:

$$\{q' \mid \exists q \in q_1 : (q', z') \in \delta(q, a, z)\}$$

Recall that because of condition 2 in the definition of sDPDA the stack activity is completely determined by the input a and top of the stack z. Therefore, the definition of δ_1

is well defined. It is easy to see that DP is a DPDA and accepts the same language as the sDPDA P. $\square$

B. Definition of VPStatic Model

We expand the notation in Section 4. $SysCall(S)$ is the system call made at S, and $Target(C) = Target(C^{'})$ is the target function of the call site C/C'.

The computation model is a DPDA $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$. We still use the simple program fragment, shown in Figure 2, as a running example. Its corresponding automaton is shown in Figure 3.

Q is the set of states. The example program has 14 states. Note we have five different kinds of states in the automaton: function entry states ($Entry(action)$ and $Entry(prepare)$), function exit states ($Exit(action)$ and $Exit(prepare)$), system call states (S_getuid, S_read and so on), call site entry states ($C1$ and $C0$) and call site exit states ($C1^{'}$ and $C0^{'}$).

Σ is the input alphabet. The input symbols of this DPDA have the forms in the proof of Theorem 1, with slight modifications. Namely, for system call and call site states, we use $Addr(p)$ instead of p to expose the state. For example, $g(a, Addr(p), z)$ or $g(a, p, z)$ means the automaton consumes the input symbol, pops z from the stack, and transitions to state p. This is equivalent to the transition $g_{a,p,z}$ used in the proof of Theorem 1. Other symbols can be similarly explained using the proof of Theorem 1. During online detection, we monitor the call stack and program counter to expose address information $Addr(p)$. We use $none$ as the placeholder when no system call is involved for the transition. All three forms of input symbols $e(\ldots)$, $f(\ldots)$ and $g(\ldots)$ appear in Figure 3.

Γ is the stack alphabet. For our model, Γ is

$$\{Addr(p) | p \in Q \text{ and } p \text{ is a function call site state}\} \cup \{z_0\}$$

where z_0 is the initial stack start symbol. We use the automaton stack of the DPDA to simulate the program call stack.

The start state q_0 is the entry state of the program entry function. F is the set of accepting states. If we require the program to exit normally, F is the set of all states for `exit` system calls. If the program can be killed anytime, F is the set of all states, or $F = Q$. Since the example program is only a program fragment, the start and the accept state set are not shown in Figure 3.

δ is the transition function. The automaton is constructed by interconnecting the transformed control flow graph of each function. If both the states connected by a transition e are in the same function, we call e an *intra-function* transition. Otherwise, we call e an *inter-function* transition. Intra-function transitions are always marked with input symbols of the form $e(...)$ since they do not deal with automaton stack. For example, in Figure 3, transition $e6$: $e(close, Addr(S\_close))$ means that if the current state is S_read, the program issues a system call $close$, and we observe that its program counter is $Addr(S\_close)$, then the current state is moved to S_close.

Inter-function transitions modify the automaton stack. They only exist between a call site entry state and its target function entry state, and between a target function exit state and a corresponding call site exit state. If T_1 is a call site entry state and T_2 is the corresponding target function entry state, we add a transition from T_1 to T_2 and label it with $f(none, T_2, Addr(T_1))$, which means the program is calling a function, and we push the address of the corresponding call site into the automaton stack. In Figure 3, transitions e_{12} and e_{13} belong to this case. If T_2 is a call site exit state, and T_1 is the corresponding target function exit state, we add a transition from T_1 to T_2 and label it with $g(none, Addr(T_2), Addr(T_2))$, which means we only follow this transition if the address of the call site the program is returning to matches the top symbol on automaton stack, and we pop this address. In Figure 3, transitions e_{14} and e_{15} belong to this case.

This completes our model definition. The formal models section proved this pushdown automaton is deterministic. Note recursive function call and return transitions are handled just like non-recursive ones.

C. Definitions and Proofs for Section 5

C.1. Definition of Dyck Model

Let $\mathcal{S}$ be the set of system call sites (traps to the operating system) and $\mathcal{C}$ be the set of function call sites. Let $\theta(c)$ denote the target function of call site c. Note that two different call sites $c_1, c_2 \in \mathcal{C}$ are unique, even if $\theta(c_1) = \theta(c_2)$.

Definition 2 *[Local Model]*
Let $G = \langle V, E\rangle$, $E \subseteq V \times V$ be the control flow graph of program Pr. Let $a \lhd v$ indicate that vertex $v \in V$ contains call site a. The local model for each function $i \in Pr$ is $A_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, F_i)$, where:

- $Q_i = V$
- $\Sigma_i = S_i \cup C_i \cup \{\epsilon\}$ *where* $S_i \subseteq \mathcal{S}$ *and* $C_i \subseteq \mathcal{C}$
- $q_{0,i} \in V$ *is the unique CFG entry state*
- $F_i = \{v \in V \setminus q_{0,i} | v$ *is a CFG exit* $\}$
- $q \in \delta_i(p, a)$ *if* $a \lhd p$ *and* $(p, q) \in E$
- $q \in \delta_i(p, \epsilon)$ *if* $\forall a \in S_i \cup C_i : a \ntriangleleft p$ *and* $(p, q) \in E$

This definition simply labels CFG edges as described in Section 5.1. All local models are ϵ-reduced and minimized to reduce their storage requirements.

The definition of the global Dyck model depends upon a classification of function call sites. Let $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, and $\mathcal{C}_4$ partition $\mathcal{C}$ as follows:

- $a \in \mathcal{C}_1$ if a does not recurse and $\theta(a)$ must generate at least 1 system call before returning.
- $a \in \mathcal{C}_2$ if a does not recurse and $\theta(a)$ may conditionally generate a system call before returning.
- $a \in \mathcal{C}_3$ if a does not recurse and $\theta(a)$ will never generate a system call before returning.
- $a \in \mathcal{C}_4$ if a may recurse.

We write $\mathcal{C}_{12}$ to denote $\mathcal{C}_1 \cup \mathcal{C}_2$.

Definition 3 *[Dyck Program Model]*
Let i range over all functions in Pr with τ the entry point function. Let f and g be the symbols used in the proof of Theorem 2, with $\mathcal{F} = \{f_{\epsilon,\gamma} : \gamma \in \mathcal{C}_{12}\}$ and $\mathcal{G} = \{g_{\epsilon,\gamma} : \gamma \in \mathcal{C}_{12}\}$. Then D is a Dyck model if there exists $D_\epsilon = (Q, \Sigma \cup \{\epsilon\}, \Gamma, \delta_\epsilon, q_0, z_0, F)$ with:

1. $Q = \bigcup_i Q_i$
2. $\Gamma = \mathcal{C}_{12}$
3. $\Sigma = \mathcal{S} \cup \mathcal{F} \cup \mathcal{G}$
4. $q_0 = q_{0,\tau}$
5. $z_0 = \epsilon$
6. $F = F_\tau$
7. $(q, z) \in \delta_\epsilon(p, a, z)$ *if* $a \in \mathcal{S}$ *and* $\exists i : q \in \delta_i(p, a)$
8. $(q, z) \in \delta_\epsilon(p, \epsilon, z)$ *if*
   (a) $\exists a \in \mathcal{C}_2 \cup \mathcal{C}_3 \, \exists i : q \in \delta_i(p, a)$ *; or*
   (b) $\exists a \in \mathcal{C}_4 \, \exists i \, \exists r \in Q_i : r \in \delta_i(p, a) \wedge q = q_{0,\theta(a)}$ *; or*
   (c) $\exists a \in \mathcal{C}_4 \, \exists i \, \exists r \in Q_i : q \in \delta_i(r, a) \wedge p \in F_{\theta(a)}$
9. $(q, za) \in \delta_\epsilon(p, f_{\epsilon,a}, z)$ *if* $a \in \mathcal{C}_{12} \wedge \exists i \, \exists r \in Q_i : r \in \delta_i(p, a) \wedge q = q_{0,\theta(a)}$
10. $(q, \epsilon) \in \delta_\epsilon(p, g_{\epsilon,a}, a)$ *if* $a \in \mathcal{C}_{12} \wedge \exists i \, \exists r \in Q_i : q \in \delta_i(r, a) \wedge p \in F_{\theta(a)}$

and $D = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ is D_ϵ under ϵ-reduction.

Several properties of the definition require explanation. Property 3 adds push and pop symbols to the alphabet of system calls. Property 7 maintains the system call transition property of the local automata: a system call will not modify stack state. Property 8(a) adds ϵ-edges around call sites that may not generate a system call. Properties 8(b) and (c) link automata at recursive call sites with ϵ-edges rather than with edges that update the PDA stack. Properties 9 and 10 describe transitions for precalls and postcalls that modify stack state.

C.2. Stack-determinism of Dyck Model

Theorem 4 *The Dyck model is an sDPDA.*

Proof: *Clearly, the Dyck model is a PDA.*

$\epsilon \notin \Sigma \Rightarrow \forall z \in \Gamma \ \nexists p \in Q : \delta(p, \epsilon, z) \neq \emptyset$, *so sDPDA condition 1 is satisfied.*

Suppose $\exists q_1, q_2 \in Q$ so that for some $\sigma \in \Sigma$ and $z \in \Gamma$, $\delta(q_1, \sigma, z) = (p_1, w_1)$ and $\delta(q_2, \sigma, z) = (p_2, w_2)$ with $w_1 \neq w_2$. Proof by contradiction in three cases:

1. *If $\sigma \in \mathcal{S}$, then $w_1 = z = w_2$ by Property 7.*
2. *If $\sigma \in \mathcal{F}$, then $\exists \gamma \in \mathcal{C}_{12} : \sigma = f_{\epsilon,\gamma}$ and $w_1 = z\gamma = w_2$ by Property 9.*
3. *If $\sigma \in \mathcal{G}$, then $\exists \gamma \in \mathcal{C}_{12} : \sigma = g_{\epsilon,\gamma}$. Then $z = z'\gamma$ and $w_1 = z' = w_2$ by Property 10.*

Thus, sDPDA condition 2 holds. □