

Proxi-Annotated Control Flow Graphs: Deterministic Context-Sensitive Monitoring for Intrusion Detection

Samik Basu¹ and Prem Uppuluri²

¹ Dept. of Computer Science
Iowa State University Ames IA 50011-1040
sbasu@cs.iastate.edu

² Dept. of Computer Science and Electrical Engineering
University of Missouri Kansas City MO 64110
uppulurip@umkc.edu

Abstract. Model or specification based intrusion detection systems have been effective in detecting known and unknown host based attacks with few false alarms [12, 15]. In this approach, a model of program behavior is developed either manually, by using a high level specification language, or automatically, by static or dynamic analysis of the program. The actual program execution is then monitored using the modeled behavior; deviations from the modeled behavior are flagged as attacks. In this paper we discuss a novel model generated using static analysis of executables (binary code). Our key contribution is a model which is precise and runtime efficient. Specifically, we extend the efficient control flow graph (CFG) based program behavioral model, with context sensitive information, thus, providing the precision afforded by the more expensive push down systems (PDS). Executables are instrumented with operations on auxiliary variables, referred to as *proxi* variables. These annotated variables allow the resulting context sensitive control flow graphs obtained by statically analyzing the executables to be deterministic at runtime. We prove that the resultant model, called *proxi-annotated control flow graph*, is as precise as previous approaches which use context sensitive push-down models and in-fact, enhances the runtime efficiency of such models. We show the flexibility of our technique to handle different variations of recursion in a program efficiently. This results in better treatment of monitoring programs where the recursion depth is not pre-determined.

1 Introduction

Intrusion detection systems (IDS) have shown promise in detecting a large number of host based attacks [4, 12, 15]. They can be categorized into (a) misuse based systems [4, 13], which detect previously known attacks by monitoring the system behavior, (b) anomaly based systems [9, 2, 11] in which machine learning or expert systems learn a system's behavior and attacks are detected as deviations of actual program behavior from learnt behavior, and, (c) specification/model based systems [7, 12] in which the *intended* program behavior is modeled and attacks are detected as deviations from this behavior. Out of these approaches, misuse based approaches cannot detect unknown

attacks as they depend on a rule base of known attacks. On the other hand, while, anomaly based approaches can detect unknown attacks, they may result in large number of false alarms, since legitimate but previously not-learned behavior can be flagged as an attack. Specification based approaches seek to combine the advantages of both by modeling program behavior. However, traditional specification based approaches [14, 12], in which a domain expert specifies legitimate program behavior using a high level specification language, are not scalable: manual specifications are tedious to write for large programs [8]. Moreover, just as any manually written software can have semantic errors, so can specifications. To counter the problems due to manually written specifications, recent research efforts [16, 8] have focused on automatically developing program behavioral models (PBMs) through static analysis of programs.

From a practical perspective, since the source codes of programs are not always available, such research efforts have focused on developing PBMs by analyzing the executables. These models are typically represented in terms of the system calls executed by the programs [8, 16]. The key requirements of such models are *precision* and *runtime efficiency*. The former requires that the models capture as much context information about the programs as possible in order to prevent false alarms. Moreover, without context information, models represent a superset of the corresponding programs' behavior. Consequently, attacks which exploit the gap between the modeled superset and the actual program behavior (called *impossible path* attacks) escape detection. In addition, the model should be efficient at runtime to be able to detect attacks before they can cause damage to the system. In this paper, we present such a program behavioral model.

Driving problem. PBMs can be classified based on their precision in capturing program behavior. Control flow graphs (CFG) or finite state machines (FSM) represent the simplest form of such models. While being efficient [1], they are imprecise: CFG/FSM's accept strings in regular languages while programs with procedure calls/returns fall in the class of context-free languages. Hence, they allow the impossible path attacks [16].

To increase precision, push-down system (PDS) have been considered. In such a model, the *context information* of each call, i.e., the return location is recorded explicitly in a pre-defined stack. However here, the precision comes at the cost of poor runtime efficiency [16, 8]. Inefficiency is due to the presence of conditional control paths in the programs; it is not possible to judge statically which of the many paths a program will take and as such the conditional controls in a program are treated as non-deterministic branch points in the program.

To get both precision and efficiency, [8] proposed the Dyke model which removes non-determinism in PDSs. Specifically, [8] instruments the executables by flanking each call site with two distinct *null system calls*. Program behavior is then modeled as a CFG in terms of the system/procedure calls made by the program – the CFGs now include the null system calls. At runtime, execution of a null call precisely identifies the call site, thus determinizing the monitoring mechanism. Intuitively, a Dyke model can be viewed as a PDS with additional calls at each call site.

While the Dyke model is precise, in the worst case if the program path is of size h (size measured in terms of number of system calls made), it requires execution of an additional $2 * h$ number of system calls. Since, interception of system calls in the kernel (the best case) causes about 20% [7] overhead over the actual execution time

of the system call, increasing their number in a program will increase this overhead. Moreover, being a variation of the PDS model, Dyke model requires pre-defining the size of a stack to record null call information. In the event of recursion, the stack may overflow resulting in the loss of context information leading to imprecise monitoring.

Our solution. In this paper, we present a novel model for capturing program behavior in terms of system and procedure calls. The central theme of our approach is to add precision to simplicity. Specifically, we use CFGs to represent programs. To add context to CFGs as well as determinism (and hence efficiency), we instrument the executables with updates to a set of variables called *proxi* variables. We call such a model, *proxi-annotated control flow graph* (PCFG). We prove that a PCFG is as precise as a PDS model and is equivalent to a Dyke model. We claim that as PCFG does not insert extra calls in the program, it is more efficient than the corresponding Dyke model. Furthermore, PCFGs make more efficient use of stacks (runtime memory usage) than PDS/Dyke models by using an array of proxi variables. In particular, except for certain types of recursion, these arrays degenerate to simple variables.

Organization. In Section 2, we present a detailed study of existing modeling mechanisms: CFG (Section 2.1), PDS (Section 2.2) and Dyke (Section 2.3). PCFG model is introduced in Section 3 followed by its comparative study with Dyke model (Section 4), with specific emphasis on stack usage by the two models.

2 Background & Related Work

2.1 Control-Flow Graphs

A control flow graph captures the behavior of a procedure. In the context of host-based intrusion detection systems, a control flow graph is typically represented by a set of states, and transition relations between pairs of states labeled by system calls and procedure calls [16]. It has a start state corresponding to the the entry point of the procedure. It can have multiple exit states due to the presence of multiple return statements in the procedure.

Definition 1 A *control flow graph (CFG)* for a procedure p is a tuple $CFG_p = (S, s_0, S_E, \longrightarrow, \mathcal{L})$ where S is the set of states, $s_0 \in S$ is the start state of p , $S_E \subseteq S$ is the set of its exit states, \longrightarrow is the set of labeled transitions $\subseteq S \times \mathcal{L} \cup \{\epsilon\} \times S$ and \mathcal{L} ranges over set of system calls and procedure invocations. \square

Figure 1(a) shows CFGs for a program with a *main*, *line* and *end* procedures. The procedure *line* can be invoked from two different call sites, one each in *main* and *end*. Note that the CFGs for individual procedures do not represent the inter-procedural control flow of the program; the reason being transitions from caller to the callee and vice versa are not explicitly present in the CFGs. A naive approach to overcome this problem is to in-line all the called procedures; a call transition is replaced by the CFG of the corresponding called procedure. Typically, such in-lining mechanism results in large global CFG leading to significant increase in space usage. Further, in-lined CFG fails to represent behaviors of recursive programs.

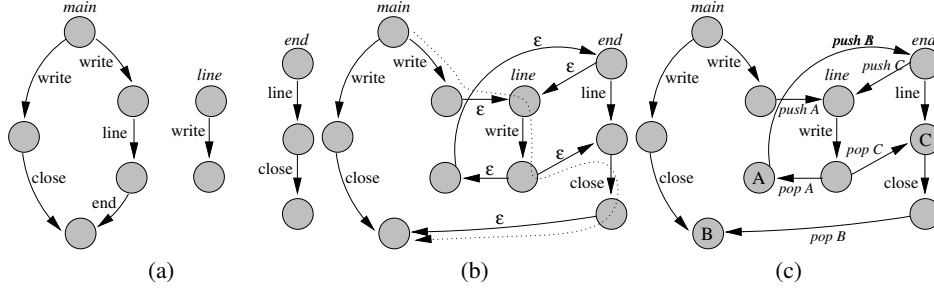


Fig. 1. (a) Local CFG, (b) global CFG with inter procedural transitions and (c) *push* and *pop* operations for PDS model

Another technique, which is employed often, is to connect the local CFGs by introducing new transitions (a) from the caller state (with outgoing call transition) to the start state of the called procedure and (b) from the exit states of a procedure to all the states which has a incoming transition labeled by call to the procedure under consideration. This technique overcomes the problem of in-lining; size of the global CFG is on the order of the sum of sizes of local CFGs. Figure 1(b) presents global CFG constructed by introducing inter-procedural ϵ -transitions and discarding call-transitions from the call site to the return locations in the local CFGs in Figure 1(a).

Context insensitivity in CFG. In a global CFG model, impossible paths can occur since it does not keep track of the location to which a program control should return once a procedure exits, i.e., *context* information of a call is lost. Such context insensitivity incorrectly classifies paths with unmatched calls and returns as valid execution sequence in the program. An example of such an impossible path in the global CFG is illustrated using *dotted* lines in Figure 1(b). Call from *main* to *line* is followed eventually by a return from *line* to *end* without any intermediate returns to *main* and calls to *end*. CFG model fails to classify the transition from exit state of *line* to return location at *end* as incorrect and hence erroneous program behavior goes un-noticed. Observe that, an impossible path results from one/more *bad* edges from the callees exit state to the return location of one of its potential callers.

A malicious user, manipulating the executing program, can use impossible paths in the model as an exploit which cannot be detected by the model [10].

2.2 Context Sensitive Model: Push-Down Systems

Push-down systems [3, 5, 6] (PDS) are deployed to detect impossible paths in a program model. A PDS captures, in addition to the intra-procedural control structure, the correct call-return pattern (context) of the program under normal circumstances. This is achieved by explicitly keeping track of the execution stack of the program whose behavior is being modeled by the PDS. Unlike CFG transition, which is between a pair of states, a PDS transition represents the change in the execution stack of the program.

Specifically, given a top-of-stack, a PDS transition relation shows the possible ways the stack changes once the statement at the current top-of-stack is executed.

Definition 2 A push-down system for a procedure p is a tuple $PDS_p = (S, s_0, S_E, \hookrightarrow, \mathcal{L})$ where S is the set of states in p (also referred to as stack symbol set), $s_0 \in S$ is the start state, $S_E \subseteq S$ is the set of exit states, \hookrightarrow is the labeled push-down transition relation $\subseteq S \times \mathcal{L} \cup \{\epsilon\} \times S^*$ and \mathcal{L} is the set of system calls and procedure calls. \square

A PDS transition can be classified as follows (s is the current top-of-stack):

- $s \xrightarrow{\epsilon} \{\}$: $s \in S_E$ is the exit state of a procedure (*pop-transition*).
- $s \xrightarrow{\text{call}_p} \{t, r\}$: $s \in S \wedge \overline{S_E}$ is call site of a procedure p . The top-of-stack, s , is replaced by start state t of p and the return location r in the caller (*push-transition*).
- $s \xrightarrow{a} \{t\}$: $s \in S \wedge \overline{S_E}$ is the top-of-stack and t is the new top-of-stack.

[10] proposed a technique for run-time monitoring using PDS. Whenever the monitored program makes a jump from one procedure to another via a call, the return location in the caller is pushed in a stack (we will refer to this as *monitor-stack*). On the other hand, if the monitored program exits a procedure and goes to a state in another procedure, the execution is deemed correct only when the destination state is present in the top of the monitor-stack. In the event the transition is allowed, the top of monitor-stack is popped out. Going back to the example in Figure 1(c), every inter-procedural transition is labeled by the operation on the monitor stack ($\text{push}(A)$, $\text{pop}(A)$ etc).

Non-determinism in PDS monitoring. PDS representation of static models suffers from the major drawback of space and time complexity, specifically, owing to the presence of non-determinism. It can be shown [16] that for a given input string of system calls, both the time and space complexity for monitoring via PDS representation technique is cubic to its size. As an illustration, see the example program in Figure 2(a) (do not consider the statements at Lines 4a, 5a, 8a and 9a). The system/function calls that are monitored are shown in bold fonts. The corresponding PDS representation is presented in Figure 2(b-I). Observe that as the valuation of conditional expression at Line 4 is not evaluated in static models, the PDS representation includes a non-deterministic inter-procedural transition: $\text{push}A/\text{push}B$ due to the presence of function invocations at Lines 5 and 9.

2.3 Dyke Model & Stack-Determinism

Recently, [8] proposed Dyke model representation based on code instrumentation to counter the problem of non-deterministic stack operations in monitoring. The central theme of this technique is to instrument the code with *null* system calls at appropriate call sites in order to determinize the stack operations in the PDS. These null calls are executed at runtime. Whenever a call is made in the program, the appropriate null call determinizes the call transitions in the model. The resultant model is equivalent to Stack-Deterministic PDS (sDPDS). The distinguishing feature of sDPDS is that, unlike PDS, there exists exactly one transition in the model corresponding to a call to a procedure in the monitored sequence. The instrumentation of code, proposed by [8], to

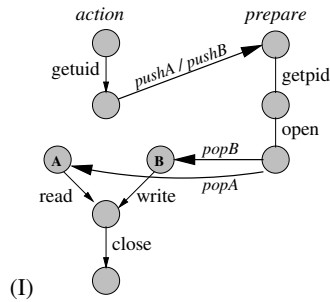
```

0:char* fname; pid_t[2] pid;
1:void action () {
2:  uid_t uid = getuid();
3:  int handle;
4:  if (uid != 0) {
4a:   precall(A);
5:   handle = prepare(1);
5a:   postcall(A);
6:   read(handle, ...);
7:  }
8:  else {
8a:   precall(B);
9:   handle = prepare(0);
9a:   postcall(B);
10:   write(handle, ...);
11:  }
12:  close(handle);
13:}

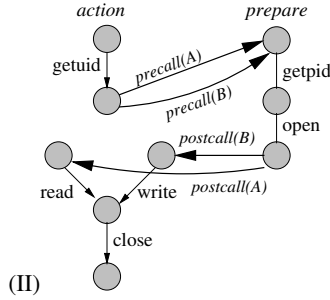
14:int prepare(int index) {
15: char buf[20];
16: pid[index] = getpid();
17: strcpy(buf, fname);
18: return open(buf, 0_RDWR);
19:}

```

(a)



(I)



(II)

(b)

Fig. 2. (a) Source-code (b-I) Non-deterministic PDS (b-II) Dyke Model

achieve this is shown in Figure 2(a) at Lines 4a, 5a, 8a and 9a. Each call is flanked by distinct “pre” and “post” calls to identify which branch of the conditional block is being executed by the monitored program. In this model, a Dyke stack is maintained which records the pre-calls that are invoked with the last pre-call being at the top-of-stack. Valid or feasible inter-procedural paths are the ones that have matching pre and post null calls. The Dyke model is shown in Figure 2(b-II).

Disadvantages of Dyke model. There are two key issues when using the Dyke model for monitoring: (a) overhead in monitoring and (b) Dyke stack size. The usage of two extra (null) calls (pre-/post-calls) per procedure-call site adds to monitoring overhead. Since, system call interception incurs ~20% overhead [7], doubling the number of intercepted system calls can raise overhead by two-fold. Secondly, note that, the defining factor for runtime memory usage is the size of the stack used by the Dyke model. The stack usage depends on the type of procedure calls: standard vs. recursive. In standard, the maximum depth of the stack can be determined statically and the Dyke stack size is set to that value. However, for recursion, the depth depends on the program’s runtime behavior. If the recursion depth is such that it overflows the Dyke stack, then the model fails to correctly identify the inter-procedural feasible paths and as such the monitoring mechanism becomes vulnerable to impossible path attacks.

3 Proxi-Annotated Control Flow Graphs

In this section, we propose a new technique based on CFG representation which determinizes stack operations without incurring overheads due to null calls. In addition, our approach performs better in terms of memory usage when handling recursion compared to the Dyke model. The central tenet of our technique is that by appropriately updating and checking valuations of auxiliary integer variables introduced at each call site we can detect impossible paths and resolve stack non-determinism efficiently. The auxiliary variables are referred to as *PROcedure context Indicator* (Proxi) and the corresponding CFG as *proxi-annotated CFG*.

Definition 3 (Proxi-Annotated CFG) A *proxi-annotated CFG* of a procedure p is a tuple $PCFG_p = (S, s_0, S_E, \longrightarrow, \mathcal{L}, \mathcal{V})$, where $CFG_p = (S, s_0, S_E, \longrightarrow, \mathcal{L})$ and \mathcal{V} is the set of proxi variable arrays where $|\mathcal{V}| = \text{number of possible return locations of } p$. \square

Notations. We introduce here the notational convenience used in the rest of the paper. Array names in \mathcal{V} for $PCFG_q$ are denoted by v_s^q and are associated with the called procedure q and the return location/state s of the callee. The maximum index of the array v_s^q holding a non-zero value is denoted by I_s^q . The pre-specified size of a proxi variable array is denoted by N_s^q .

Realizing PCFG monitoring. The monitoring mechanism via PCFG proceeds by updating the proxi variables in the following fashion:

1. *Initialization.* All proxi variables are initialized to zero, i.e., $\forall v_s^q. \forall i < N_s^q. v_s^q[i] = 0$. Furthermore, each I_s^q is set to zero.
2. *Call to procedure: incrementing proxi variables.* As eluded before, elements in array v_s^q records a call to procedure q with the return location s in the caller.
 - (a) If there exists a non-zero $v_r^q[I_r^q]$ less than $v_s^q[I_s^q]$ with $r \neq s$, then $I_s^q = I_s^q + 1$. $v_s^q[I_s^q]$ is incremented by one and all non-zero $v_r^q[I_r^q]$ where $r \neq s$ are also incremented by one.
 - (b) Else, $v_s^q[I_s^q]$ are incremented by one and all non-zero $v_r^q[I_r^q]$ where $r \neq s$ are incremented by one.
3. *Return from a procedure: decrementing proxi variables.* If a procedure q returns, the correct return location s in the caller is determined by the array v_s^q . The conditions to be satisfied on return are that the valuation of $v_s^q[I_s^q]$ (a) is greater than zero and (b) is minimum among all non-zero $v_r^q[I_r^q]$. $v_s^q[I_s^q]$ is decremented by one. If $v_s^q[I_s^q] = 0$ and $I_s^q \neq 0$ then decrement I_s^q by one. All other proxi variables associated with q , i.e., $v_r^q[I_r^q]$ are decremented by one.

Observation. The updates to the proxi variables ensure that the element of the proxi variable array corresponding the last call in the execution sequence is always less than the proxi variable elements for any prior calls.

Theorem 1. A sequence of steps, in terms of procedure calls, is feasible in a PDS model iff it is also feasible in a PCFG.

Calls to procedure q with return locations r and s					
Stack operation	PDS	Proxi variable values			
		push(s)	$v_s^q[0]=1$	$v_r^q[0]=0$	$I_s^q=0$
push(r)	$v_s^q[0]=2$	$v_r^q[0]=1$	$I_s^q=0$	$I_r^q=0$	
push(s)	$v_s^q[0]=2, v_s^q[1]=1$	$v_r^q[0]=2$	$I_s^q=1$	$I_r^q=0$	
push(s)	$v_s^q[0]=2, v_s^q[1]=2$	$v_r^q[0]=3$	$I_s^q=1$	$I_r^q=0$	
pop(s)	$v_s^q[0]=2, v_s^q[1]=1$	$v_r^q[0]=2$	$I_s^q=1$	$I_r^q=0$	
pop(s)	$v_s^q[0]=2, v_s^q[1]=0$	$v_r^q[0]=1$	$I_s^q=0$	$I_r^q=0$	

Fig. 3. Updates to proxi variables with sample operation sequence to monitor stack

Proof: The proof proceeds by showing that proxi variables correctly record the monitor stack of a PDS model. Recall that a call to procedure q with return location s results in *push*-ing s to the top of the monitor stack of PDS model (see Section 2.2).

Assume that a procedure q can be invoked multiple times with two different return locations s and r . Let q be a new call, seen with the return location s . Following PDS monitoring mechanism, s is pushed to the top-of-stack due to the new call to q .

Case I: s is already present in the stack and the top-of-stack is r , i.e. in the execution sequence there are at least two prior calls to q with two different return locations. This implies that $v_r^q[I_r^q]$ is less than $v_s^q[I_s^q]$. The corresponding updates to the proxi variable $v_s^q[I_s^q]$ follows the rule 2(a) (see above). In other words, I_s^q is incremented and $v_s^q[I_s^q]$ is incremented by one. Also $v_r^q[I_r^q]$ is incremented by one. As such, $v_s^q[I_s^q]=1$ and $v_s^q[I_s^q] < v_r^q[I_r^q]$. The return location for the new call to q is correctly identified as s .

Case II: s is present in the top of the stack. This implies the new call to q is a recursive call with the same return location. The updates to the proxi variables follow the rule 2(b) (see above). This ensures that number of recursions to q with same return location s is equal to the valuation of $v_s^q[I_s^q]$. \square

Figure 3 shows a sample session of execution monitoring using PDS model and PCFG. The sequence of operations on monitor stack are presented along with the corresponding changes in the proxi variables in PCFG.

Theorem 2. *A PCFG is equivalent to Dyke model.*

Proof. Theorem 1 states that PCFG only allows feasible inter-procedural sequences as per the PDS model. Here we give the proof sketch showing that PCFG also resolves non-determinism in PDS model monitoring.

Recall that a Dyke model instrumentation involves inserting distinct pre-/post-calls at each call site. This ensures determinism in stack operations as each call site is distinguished by its pre-call. A Dyke stack records the pre-call and a path is deemed feasible if each return leads to post-call matching with the last pre-call at the top of Dyke stack.

In identical fashion, PCFG inserts updates to the proxi variables at each call site. In other words, a pre-call of a Dyke model is replaced by incrementing operations of proxi variables and the post-call is replaced by assertions that must be satisfied followed by

Call to q with return state s	
Dyke Model	Proxi-annotated CFG
precall(s)	<pre> if $\exists r. v_r^q[I_r^q] \neq 0 \ \&\& \ v_r^q[I_r^q] < v_s^q[I_s^q]$ then I_s^q++; $v_s^q[I_s^q]++$; $\forall v_r^q[I_r^q] > 0 \ \&\& \ r \neq s \ v_r^q[I_r^q]++$; </pre>
postcall(s)	<pre> assert($v_s^q[I_s^q] > 0$); $\forall r \neq s. \text{if } v_r^q[I_r^q] \neq 0 \text{ then assert}(v_s^q[I_s^q] < v_r^q[I_r^q])$; $\forall r \neq s. \text{if } v_r^q[I_r^q] > 0 \ v_r^q[I_r^q]--$; $v_s^q[I_s^q]--$; if $v_s^q[I_s^q] == 0 \ \&\& \ I_s^q \neq 0$ then I_s^q--; </pre>

Fig. 4. Code Instrumentation

decrementing the proxi variables. The variables are distinguished by the return locations of the called procedure. As such, all procedure calls (i.e. stack operations in the context of Dyke model) are determined in PCFG with respect to their return locations. \square

Figure 4 presents the instrumentations for PCFG corresponding to pre- and post-calls in Dyke model. Feasible path in PCFG must satisfy all the assertions corresponding to the post-call.

4 Discussion

There are two important distinguishing aspects of our technique that can make it potentially more efficient than the existing (PDS/Dyke) stack-based techniques.

First, we replace null calls in Dyke model with updates to variables. Note that the number of arrays of proxi variables $|\mathcal{V}|$ is equal to the number of pre-calls introduced by the Dyke model. This will reduce runtime overhead caused by monitoring extra null calls at each call site. The operations on the proxi variables (see Figure 4) can be executed in time linear to the number of proxi variables by clever arrangement of proxi variables in an ascending order.

Secondly, observe that, the bound on recursion depth of the execution sequence that can be correctly monitored is equal to the pre-defined size of the Dyke stack. In PCFG, the recursion bound depends on the *kind* of recursion. Specifically we identify two kinds: *uni-valent* and *multi-valent*. Uni-valent recursion corresponds to the case where the same procedure is invoked with the same return location recursively. In this situation, the proxi variable element corresponding to the concerned procedure is simply incremented (Rule 2(b) for proxi variable incrementation) with each recursion as opposed to Dyke model where pre-calls are pushed in the stack. The bound on recursion depth, therefore, is determined by the integer domain of the proxi variable element.

Multi-valent recursion occurs when there are at least two alternating recursive calls to the same procedure with two different return locations. For example, let q be called

with return location s followed by a recursive call to q with r as return location (with no intermediate returns from q). We say that the *depth of alternation*, in this example, is 1. Note that, for a subsequent new call to q with return location s (new alternation depth becomes 2) leads to incrementation of the index I_s^q . As such, the maximum depth of alternation in an execution sequence that can be monitored correctly is determined by the ranges (e.g. N_s^q , N_r^q etc.) of the proxy variable array.

In summary, unlike PDS/Dyke models, PCFG model monitors recursion in two different dimensions. The domain of individual elements in a proxy variable array determines the maximum depth of the uni-valent recursion while the range/size of the array limits the depth of alternation in multi-valent recursion. Such separation of uni- and multi-valent recursive patterns adds to the flexibility and efficiency of monitoring. For example, if it is statically determined that multi-valent recursions are not possible in an execution sequence, then the array sizes of proxy variables are set to 1.

References

1. A.V. Aho. *Handbook of Theoretical Computer Science Vol A*. Elsevier Science Publishers B.V., 1990.
2. D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Next-generation intrusion detection expert system: A summary. Technical Report SRI-CSL-95-07, SRI International, 1995.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCURR*, 1997.
4. S. Eckmann, G. Vigna, and R. Kemmerer. Statl. Technical report, UCSB, 2000-19.
5. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247. Springer-Verlag, 2000.
6. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Computer-Aided Verification (CAV)*, pages 324–336. Springer-Verlag, 2001.
7. T. Bowen et al. Building survivable systems: An integrated approach based on intrusion detection and confinement. In *Darpa Information Security Symposium*, 2000.
8. H. Feng, J. Griffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, May 2004.
9. S. Forrest, R. Henning, J. Reed, and R. Simonian. A neural network approach towards intrusion detection. In *National Computer Security Conference*, 1990.
10. J. T. Griffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Usenix Security Symposium*, August 2002.
11. K. Ilgun. A real-time intrusion detection system for unix. In *IEEE Symposium on Security and Privacy*, 1993.
12. C. Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach*. PhD thesis, University of California, Davis, December 1996.
13. J. Pouzol and M. Ducasse. From declarative signature to misuse intrusion detection systems. In *RAID*, 2001.
14. R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 99.
15. P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In *RAID*, 01.
16. D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, May 2001.