Efficient Context-Sensitive Intrusion Detection

Jonathon T. Giffin Somesh Jha Barton P. Miller
Computer Sciences Department
University of Wisconsin, Madison
E-mail: {giffin,jha,bart}@cs.wisc.edu

Abstract

Model-based intrusion detection compares a process's execution against a program model to detect intrusion attempts. Models constructed from static program analysis have historically traded precision for efficiency. We address this problem with our Dyck model, the first efficient statically-constructed context-sensitive model. This model specifies both the correct sequences of system calls that a program can generate and the stack changes occurring at function call sites. Experiments demonstrate that the Dyck model is an order of magnitude more precise than a context-insensitive finite state machine model. With null call squelching, a dynamic technique to bound cost, the Dyck model operates in time similar to the context-insensitive model.

We also present two static analysis techniques designed to counter mimicry and evasion attacks. Our branch analysis identifies between 32% and 64% of our test programs' system call sites as affecting control flow via their return values. Interprocedural argument capture of general values recovers 32% to 69% more arguments than previously reported techniques.

1. Introduction

Host-based intrusion detection seeks to identify attempts to maliciously access the machine on which the detection system executes. Remote intrusion detection identifies hostile manipulation of processes executing in a distributed computational grid [10]. These intrusion detection systems monitor processes running on the local machine and flag unusual or unexpected behavior as malicious. In model-based detection [8], the system has a model of acceptable behavior for each monitored process. The model describes actions that a process is allowed to execute. A monitor compares the running process's execution with the model and flags deviations as intrusion attempts.

Model-based intrusion detection can detect unknown attacks with few false alarms. Such a system detects new and novel attacks because the model defines acceptable process behavior rather than the behavior of known attacks. Yet, false alarms are low to non-existent for a properly constructed model because the model captures all correct execution behaviors.

Constructing a valid and precise program model is a challenging task. Previous research has focused on four basic techniques for model construction: human specification [14], training [5, 7, 17, 23, 34], static source code analysis [31, 32], and static binary code analysis [10]. Of these, we use static binary code analysis since it requires no human interaction, no determination of representative data sets, and no access to a program's source code, although it is unsuitable for interpreted-language analysis. It constructs models that contain all possible execution paths a process may follow, so false alarms never occur. However, an imprecise model may incorrectly accept attack sequences as valid. We use static binary analysis to construct a finite state machine that accepts all system call sequences generated by a correctly executing program.

Models constructed from static program analysis have historically traded precision for efficiency. The most precise program representations, generally context-sensitive push-down automata (PDA), are prohibitively expensive to operate [10, 31, 32]. For example, Wagner and Dean suggested the use of their less precise digraph model simply because more precise models proved too expensive. Our earlier work used regular language overapproximations to a context-free language model, again due to cost. This paper presents a new model structure that does not suffer

*This work is supported in part by Office of Naval Research grant N00014-01-1-0708, Department of Energy grants DE-FG02-93ER25176 and DE-FG02-01ER25510, Lawrence Livermore National Lab grant B504964, and NSF grant EIA-9870684. 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.

from such drawbacks. Our Dyck model is a highly precise context-sensitive program representation with runtime behavior only slightly worse than a cheap, imprecise regular language model.

The Dyck model is as powerful and expressive as the full PDA model. An early result by Chomsky proved that every context-free language is a homomorphism of the intersection of a Dyck language with a regular language [2]. Chomsky's result implies that our Dyck model is as powerful as the PDA model, so the efficiency gains we observe come at no loss in correctness.

The Dyck model can detect a broad class of attacks. Generally, the model detects attacks that execute arbitrary code, as this code will not match the expected behavior of the process. For host-based intrusion detection, this includes:

- Attempts to exercise a race condition that uses invalid control flow to repeatedly execute a code sequence.
- Attempts to bypass security checks via impossible paths (see Appendix A).
- Attempts to execute programs via command insertion in unsanitized arguments to subshells.
- Changing a symbolic link target before an `exec` call.
- Buffer overruns, heap overflows, or format string attacks that force a jump to injected code.

The Dyck model is further suited for *remote intrusion detection*. This detection technique identifies hostile manipulation of remotely executing programs that send certain system calls to a different, local machine for execution. Successful remote manipulation means the local system executes malicious system calls. This is a stronger threat model than the host-based intrusion detection setting. Attackers do not exploit vulnerabilities at specific points of execution but can replace the entire image of the remote process with their attack tool at any arbitrary execution point. By modeling the remote job with a Dyck model and monitoring the stream of remote system calls arriving at the local machine, we can detect remote manipulation that produces invalid call sequences.

This paper makes three primary contributions:

The Dyck model, enabling efficient context-sensitive program modeling. The Dyck model represents a substantial improvement in statically constructed program models. Our Dyck model exposes call stack changes to the monitor. Model operation is highly efficient because the monitor explores only the exact call path followed by the application.

Experiments bear out these claims. All our test programs show an order of magnitude improvement in precision when using the Dyck model rather than a context-insensitive model. For example, the model precision for `procmail` improved from 14.2 with a context-insensitive model to 0.8 with the Dyck model as measured by the average branching factor metric. Excluding recursive call sites, impossible paths [31, 32] do not exist in the Dyck model. The only sequences of system calls it accepts are those that the program could actually produce.

Null call squelching, a dynamic method to limit null call generation. We have developed *null call squelching* to prevent excessive null call generation without reducing security. Squelching combines both static and dynamic techniques to generate only those null calls that provide context for a system call. With squelching enabled, the worst-case number of null calls generated per system call is bounded by $2h$, where h is the diameter of the program's call graph. We present the Dyck model and null call squelching in Section 4.

Efficiency gains demonstrate the value of squelching. Previous experiments using a context-sensitive PDA could not even be completed because the model update failed to terminate in reasonable time [31, 32]. With the Dyck model, operational cost nears that of a context-insensitive nondeterministic finite automaton (NFA) model.

Data flow analyses to counter mimicry attacks. We use interprocedural data flow analysis to model arguments passed to and return values received from system calls. In combination, these analyses hinder mimicry and evasion attacks [26, 27, 28, 33] by restricting the paths in the program model that accept an attack sequence. We discuss data flow analysis in Section 5.

2. Related Work

In human-specified model-based intrusion detection, a security analyst manually specifies correct behavior for each program of interest [14, 24] or annotates the source code to describe security properties [1]. A runtime monitor enforces the manually described model. Alternative systems check behavior against a specification of malicious activity [18]. Such systems are reasonable for very small programs; however, as programs grow, human specification becomes overly tedious.

Static and dynamic program analysis scale better by automatically constructing models. Wagner and Dean statically analyzed C source code to extract both context-insensitive and context-sensitive models [31, 32]. Unfortunately, the cost to operate their precise context-sensitive *abstract stack* model was prohibitively high and unsuitable for practical use. We observed similar expense when using context-sensitive push-down automata constructed via static analysis of SPARC binary code [10]. These papers recommended using imprecise context-insensitive models to achieve reasonable performance. The Dyck model presented in this paper significantly improves upon

these works, providing a precise context-sensitive model with excellent performance characteristics.

Wagner and Dean also introduced the impossible path exploit. A context-insensitive model includes paths originating from one function call site but returning to a different call site. A correctly executing program could never follow such a path due to its call stack; however, an attacker could force impossible control flow via an exploit. Our Dyck model is context-sensitive and detects impossible path exploits.

Dynamic analysis, based upon the seminal work of Forrest *et al*. [7], constructs program models from observed behavior during repeated training runs [8, 9, 12, 13, 16, 17, 19, 29, 35]. Feng *et al*. [5] extended the work of Sekar *et al*. [23] to learn sequences of system calls and their calling contexts. Their *VtPath* program model is a database of all pairs of sequential system calls and the stack changes occurring between each pair, collected over numerous training runs. The VtPath language is the regular language expansion of a context-free language with bounded stack. This is equivalent to our Dyck model, where the stack bound is the maximum depth of the program's call graph when ignoring recursion. However, our work differs from that of Feng *et al*. in four important aspects:

- The Dyck model is fundamentally more expressive than VtPath. For efficiency, the Dyck model treats recursion as regular. However, this is not a limitation of the model. The Dyck model can correctly express context-sensitive recursive calls and accept a strictly context-free language. VtPath cannot model recursion because all possible recursive depths would need to be learned during training. It must accept a regular language.

- The Dyck model, via its null call instrumentation, detects attacks that VtPath cannot. Null calls reduce non-determinism, better enabling the monitor to track process execution. Appendix A presents an example.

- The static analyzer constructing our Dyck model analyzes system call arguments and return values to prevent mimicry attacks [26, 27, 28, 33]. The Dyck model includes restrictions on valid arguments and acceptable execution directions based upon system call return values. The VtPath model, and, indeed, all but one learned model [25], ignore these arguments and return values.

- Our context-free Dyck model is a compact program representation. In the worst case, a regular language expansion of a bounded context-free language, such as VtPath, may grow exponentially large.

We view static and dynamic analysis techniques as complementary. Static analysis overapproximates acceptable program behaviors and generates a model that may miss attacks. Conversely, dynamic analysis underapproximates acceptable behaviors, leading to a high false alarm rate. Ultimately, a hybrid model based upon both approaches could be advantageous by minimizing the drawbacks of each technique. Although we chose to present the Dyck model in the context of static analysis, it appears equally well suited for use in dynamic analysis or a hybrid approach.

3. Model Construction Infrastructure

For completeness of presentation, we have included a summary of infrastructure work in this area. Readers familiar with such work can skip to this paper's major new contributions: the Dyck model in Section 4 and mimicry attack defenses in Section 5.

Our tool features two components: the *binary analyzer* and the *runtime monitor*. The analyzer reads a SPARC binary program and uses static program analysis to construct a model of the program. Additionally, it rewrites the binary program code to enable more precise and efficient modeling. The user then executes the rewritten binary in their security-critical environment. The runtime monitor tracks the execution of the rewritten binary to ensure that it follows the analyzer's constructed model. Deviation from the model indicates that a security violation has occurred.

Our program model is a finite state machine whose language defines all possible sequences of system calls that an application may generate during correct execution. Model construction progresses through three stages.

1. We read the binary program and construct a *control flow graph* (CFG) for each procedure in the application. Each CFG represents the possible control flows in a procedure.

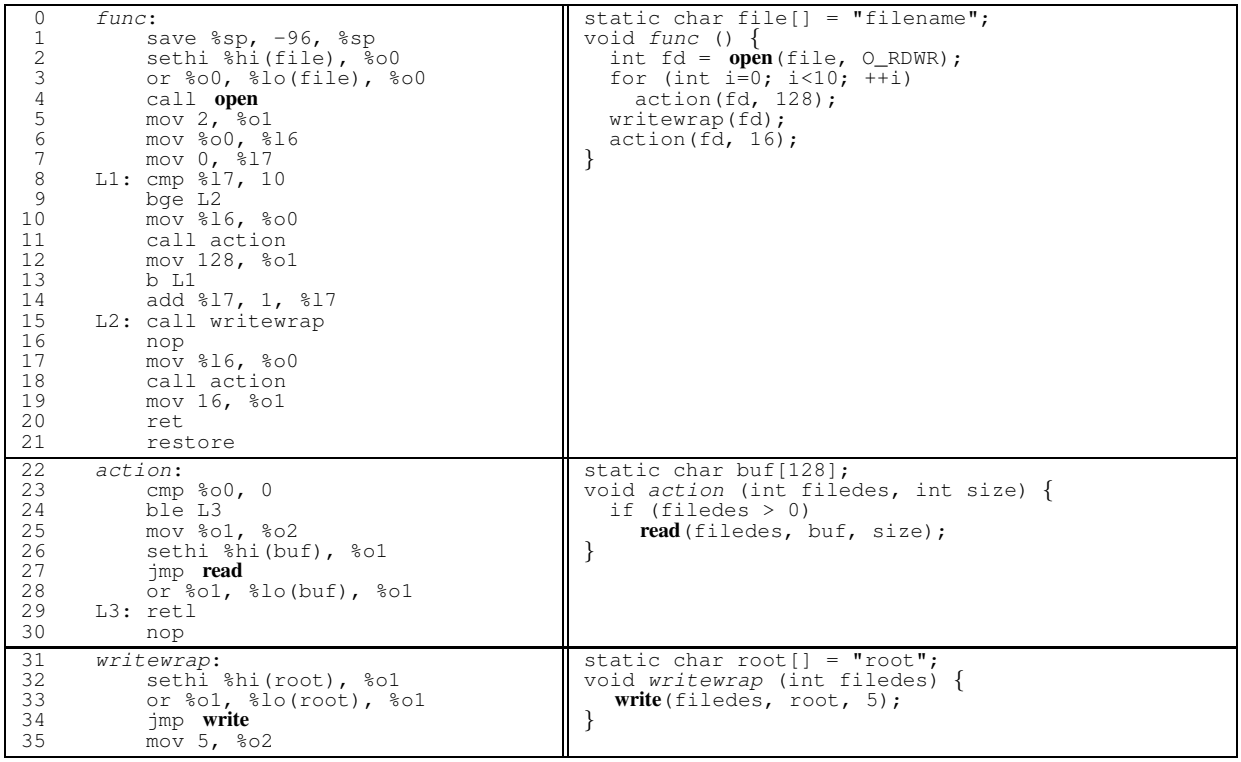
2. We convert each control flow graph into a nondeterministic finite automaton (NFA) that models all correct call sequences that the function could produce.

3. We compose the collection of local automata at points of internal user function calls to form a single interprocedural automaton modeling the entire application.

The runtime monitor enforces the program model by operating the interprocedural automaton at runtime.

Figure 1 contains the SPARC assembly code for three example functions, with system calls in boldface. Figure 2 presents the NFA constructed for each function.

Note that system call transitions include arguments. We analyze the data flow of the program to reconstruct an expression graph for each argument. By simulating execution of the machine instructions in the expression graph, the analyzer recovers statically known argument values. This



```
 0    func:
 1        save %sp, -96, %sp
 2        sethi %hi(file), %o0
 3        or %o0, %lo(file), %o0
 4        call open
 5        mov 2, %o1
 6        mov %o0, %l6
 7        mov 0, %l7
 8    L1: cmp %l7, 10
 9        bge L2
10        mov %l6, %o0
11        call action
12        mov 128, %o1
13        b L1
14        add %l7, 1, %l7
15    L2: call writewrap
16        nop
17        mov %l6, %o0
18        call action
19        mov 16, %o1
20        ret
21        restore
```

```
static char file[] = "filename";
void func () {
  int fd = open(file, O_RDWR);
  for (int i=0; i<10; ++i)
    action(fd, 128);
  writewrap(fd);
  action(fd, 16);
}
```

```
22    action:
23        cmp %o0, 0
24        ble L3
25        mov %o1, %o2
26        sethi %hi(buf), %o1
27        jmp read
28        or %o1, %lo(buf), %o1
29    L3: retl
30        nop
```

```
static char buf[128];
void action (int filedes, int size) {
  if (filedes > 0)
    read(filedes, buf, size);
}
```

```
31    writewrap:
32        sethi %hi(root), %o1
33        or %o1, %lo(root), %o1
34        jmp write
35        mov 5, %o2
```

```
static char root[] = "root";
void writewrap (int filedes) {
  write(filedes, root, 5);
}
```

Figure 1. SPARC assembly code and C source code for three example functions, *func*, *action*, and *writewrap*. We analyze binary code and include this source code only to aid comprehension of the code behavior.

recovery prevents an attacker from passing arbitrary arguments to system calls. Observe that the first argument to `read` in Figure 1, the file descriptor returned by `open`, is a dynamic value and cannot be statically recovered with this technique. Section 5.1 presents a new technique for recovery of such values.

These automata have a desirable property for system call modeling: in the absence of indirect function calls, the model is safe; i.e., if there exists an input to an underlying function f such that f produces a sequence of calls $a_1 ... a_n$, then the language of the automaton accepts this sequence. Hence, *the monitor will not raise false alarms*. To maintain the safety property at indirect call sites, we first attempt argument recovery on the jump register to find all possible targets. For the six test programs used in Section 6, our analysis recovers between 70% and 80% of indirect targets. In the remaining cases, we mark the call-site as targeting any function whose address is taken.

Call-site replacement constructs a model of the entire application by splicing local automata together at function call edges. This models the program's execution at points of function calls, i.e. control flow shifts into the called procedure. Previous work constructed either an NFA or PDA global model [10, 31, 32]; unfortunately, neither model is entirely satisfactory.

The NFA model (Figure 3) is an imprecise but efficient context-insensitive model. An NFA offers excellent runtime performance, but suffers from impossible path exploits. Impossible paths exist when multiple different call sites to the same target procedure exist. The language accepted by the model is then a superset of the program's actual language and includes paths not possible in actual program execution. These paths are important: an attacker may use the existence of such edges to attack a process without detection. The bold path in Figure 3 is an impossible path accepting repeated `read` and `write` calls.

A PDA model adds context-sensitivity for greater precision, but suffers from extremely high runtime overheads. Figure 4 shows how the PDA includes a model of the program's call stack. The monitor will only traverse matching call and return transitions, so impossible paths do not exist in the model. This stack model adds complexity to the operation of the PDA. Straightforward execution fails in the presence of left recursion. The `post*` algorithm [4], designed to terminate even in a left recursive grammar, has worst-case complexity that is cubic in the number of au-

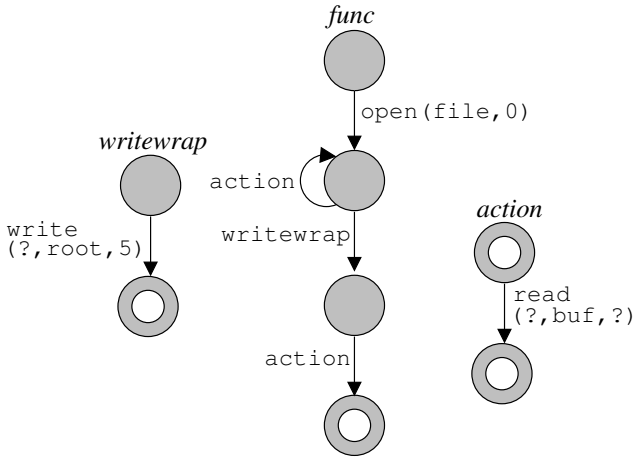


Figure 2. Local function models.

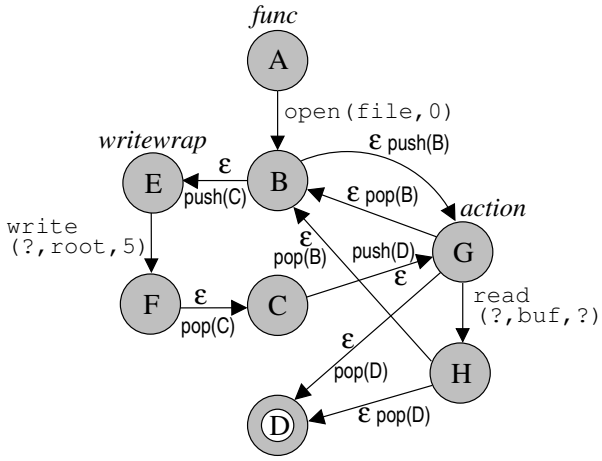


Figure 4. PDA program model.

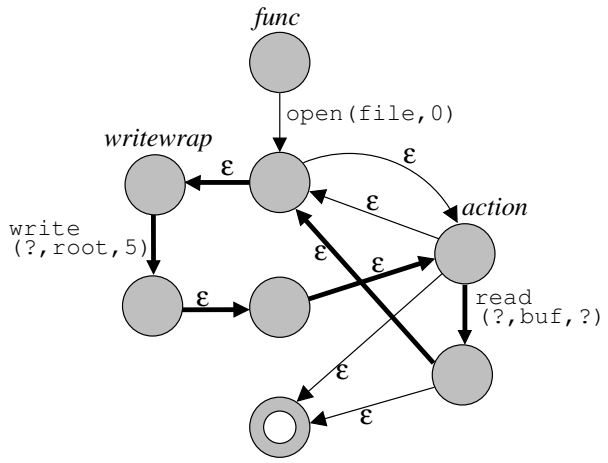


Figure 3. NFA program model. The bold cycle is an impossible path.

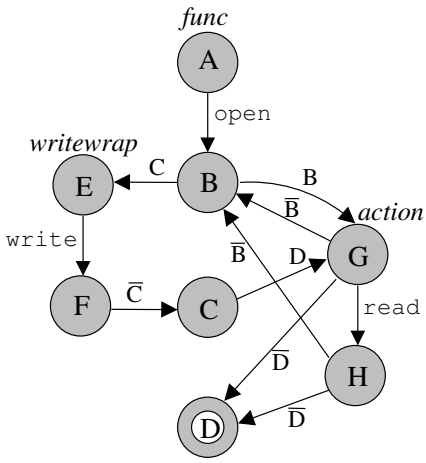


Figure 5. Dyck model without squelching.

tomaton states [22] and leads to unreasonably high runtime overheads [31, 32].

Binary rewriting can somewhat mitigate the cost of PDA operation via *null call insertion*. Null calls, or dummy system calls, observed by the monitor indicate the path of execution followed by the process. This limits runtime exploration of the PDA to the states dominated by the null call transition. Unfortunately, this naive null call insertion has two shortcomings. First, we cannot statically compute the cost of a particular null call insertion point [20], possibly leading to high cost. Second, the execution context information is accurate only until an attacker takes control of the application. Our Dyck model addresses these shortcomings by providing an attack-resilient context-sensitive model that dynamically controls null call cost.

4. Dyck Model

We have developed the Dyck model, the first efficient statically-constructed context-sensitive model. The Dyck model achieves much greater efficiency than a PDA by limiting state exploration. Like a PDA, the Dyck model includes a stack to record function call return locations. *In a Dyck model, however, all stack update transitions are also symbols in the automaton alphabet*. The monitor then updates the Dyck stack precisely when that update reflects actual program behavior. To produce these stack update symbols, we insert two null calls at selected function call sites in the program. A *precall*, immediately before the function call, notifies the monitor of the calling location. When the call returns, the program generates a *postcall*. The null calls inserted at each call site are different, so each call and return path to the same target function is distinguishable. Any postcall not matching the corresponding precall in-

```
1  void  func () {
2    int fd = open(file, O_RDWR);
3    for (int i=0; i<10; ++i) {
       null_call(B);
4      action(fd, 128);
       null_call(B̄);
     }
     null_call(C);
5    writewrap(fd);
     null_call(C̄);
     null_call(D);
6    action(fd, 16);
     null_call(D̄);
   }
```

Figure 6. Code example with Dyck instrumentation. Inserted null calls appear in boldface. Each user call has a null call indicating call and return. Line numbers correspond to those in Figure 1. Although this figure shows C code for readability, we instrument SPARC binary code.

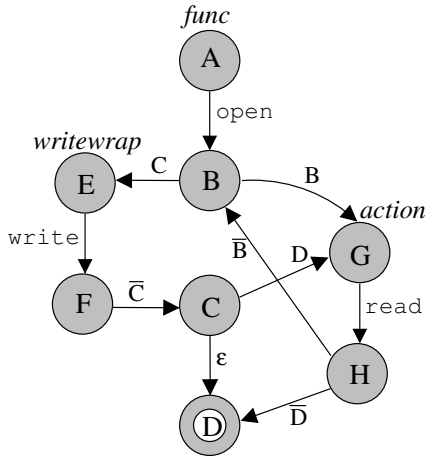


Figure 7. Dyck model with squelching.

dicates that the program is attempting to force execution through an impossible path.

The language accepted by the Dyck model is a *bracketed context-free language* originally developed by Ginsberg and Harrison [11]. The precall and postcall inserted at each call site correspond to parenthesis symbols in the language and form a Dyck language [3, 30]. The monitor accepts only sequences that correctly match paired pre- and postcalls. Note that this forced pairing is a stricter use of null calls than in previous work and prevents the introduction of impossible paths even when under attack. *An attacker is free to insert or change the null calls as he or she wishes; however, the manipulations must match some correct program execution path.*

Figure 5 shows the Dyck model. Null calls link the entry and exits of a target function's model with the call sites to that function. Edges labeled α are precalls that insert α onto the Dyck stack. Edges labeled $\overline{\alpha}$ are postcalls that pop α. When reaching state B in the Dyck model, the monitor will follow only the transition corresponding to the observed symbol in the call stream. Conversely, when operating a PDA, the monitor must replicate its state and follow both stack push transitions to states E and G, suffering greater overhead. Figure 6 shows how the program rewriter inserts the Dyck null calls into the existing program (recall that we instrument binary code). Each precall and postcall is inserted immediately before and after each call site.

Appendix B gives the formal definition of the Dyck model.

4.1. Selecting Instrumentation Points

Naive instrumentation may lead to excessive run-time overhead if program execution generates a null call with high frequency. *Recursion* and *loops* exacerbate the number of null calls produced. In these cases, execution follows a backedge in a function's control flow graph or in the program's call graph and leads to repeated null call site execution. Other execution patterns do not correspond to backedge traversal and thus do not affect the rate at which execution encounters a particular null call.

We do not insert Dyck calls naively. Our selection algorithm statically chooses function call sites to *avoid* instrumenting. First, it will not instrument recursive call sites. Each strongly connected component (SCC) in the program's call graph represents a recursive cycle. This rule flattens each SCC into a single node. We lose context sensitivity at points of recursion, but limit the cost of instrumentation.

Second, we do not instrument call sites that never execute a system call. Note that a function f will not execute a system call if the entire subgraph of the program's call graph rooted at f never reaches a system call. This prunes portions of the call graph that are uninteresting for system call monitoring. The monitor need not follow the program's execution through such functions because they cannot generate a system call.

4.2. Null Call Squelching

A strictly static technique cannot adequately address the looping problem. We have developed *null call squelching*, a dynamic technique that restricts null call generation. Squelching produces only the meaningful null calls indicating the call stack state when reaching a system call. Null calls around a function call that returns without generating a system call provide no security information and are discarded. We show two important results: first, the number of null calls generated is bounded by $2hn$ where h is the diameter of the program's call graph and n is the number

of system calls generated. Second, we show that the model resists attacker manipulation. We begin by describing the squelching algorithm.

We do not change the selection of null call instrumentation points; rather, we modify the semantics of instrumentation. First, we create a *squelch stack* in the program's data space. The precall instrumentation pushes the call site identifier onto the squelch stack, but does not send the identifier to the monitor.

We modify system call sites to send the squelch stack along with the system call. The precall identifiers on the squelch stack represent the calling context at the system call. The squelch stack is then cleared.

The postcall code examines the state of the squelch stack. If the stack is empty, then some system call site sent all symbols to the monitor, including the precall at this call site. Thus, the postcall is meaningful and is sent to the monitor. If the stack is not empty, then this call site generated no system call. The application pops the top element from the call stack. Rather than inserting irrelevant null calls into the call stream, this algorithm discards them at the slight expense of stack activity in the application.

Note that a postcall that pops an element should match the popped element. A mismatch indicates program manipulation not visible to the monitor has occurred. The program could kill itself, although an attacker could prevent the termination. We instead observe that the manipulation is uninteresting because it generated no system calls. Moreover, the squelch stack has entered a bad state that may be revealed at the next system call event.

As an example of squelching, consider Table 1. Line (a) shows one path through the Dyck model of Figure 5 without null call squelching. Every function call that does not generate a system call produces a matched Dyck pair $\alpha, \overline{\alpha}$ in the call stream. Clearly, such pairs provide no system call context and can be removed. Line (b) shows the same call string with such pairs removed. Every remaining Dyck pair envelops some system call and indicates the application's stack context at the point of that system call. With squelching, the runtime cost of null call insertion is notably reduced with no loss of security.

We change model construction to incorporate null call squelching. In particular, any precall-postcall sequence must be converted to an ϵ-transition. We describe this as a language transformation. Let L be the language accepted by the Dyck model without null call squelching and L' be the language accepted with squelching. Let $h : L \rightarrow L$ replace all precall-postcall strings with ϵ. Then $h^* : L \rightarrow L'$, denoting recursive calls to h terminating when no precall-postcall strings exist in L', generates the squelched language L'. Figure 7 shows the Dyck model transformed to accept a squelched language. Note that the pair $D, \overline{D}$ from C to G to D has been replaced with an ϵ-transition directly from C to D.

We finally show that null call squelching imposes a strict upper bound on the cost of instrumentation.

THEOREM. Let C be the call graph for program P. Denote by $\widetilde{C}$ the graph obtained from C with each strongly connected component collapsed to a single state. Let h be the maximum diameter of $\widetilde{C}$. If P generates n true system calls during execution, then the worst-case number of null calls generated is $2hn$.

PROOF. See Appendix B. □

4.3. Resilience to Attacker Manipulation

The Dyck model relies upon state kept with the application: the squelch stack and the rewritten call sites that produce null calls. Since this state is in the memory image of the process and not of the monitor, an attacker may arbitrarily modify the state. We claim that the Dyck model is resilient to any such modification. That is, modifications are successful only if they represent possibly legitimate program behavior.

First, the attacker could modify the stack. The monitor will detect added elements before a system call if the call path represented by the stack is not legitimate. By the same argument, element deletion will be detected if it attempts to introduce an impossible path. We note that although a denial-of-service attack is possible by releasing the memory used by the stack to produce a memory fault at the next stack reference, the process could be killed by a myriad of simpler means.

Second, the attacker could modify the code. The attacker could prevent null call generation, generate a large number of null calls, or send erroneous null calls. These are equivalent to the stack manipulations previously discussed and will be detected if they attempt to introduce an impossible path. Again, generating a large number of null calls may terminate the process if the squelch stack space becomes exhausted.

The monitor stores the program model in a separate process space, so an attacker cannot modify the model. Simply put, any modifications to the state kept in the application still must produce valid call sequences to be accepted by the monitor. Thus, the attacker gains nothing by modifying this state.

5. Data Flow Analysis

We have designed two advanced data flow analyses to counter the mimicry and evasion attacks described in recent literature [26, 27, 28, 33]. These papers stress the need to monitor system call arguments and return values to prevent an attacker from using system calls as *nop*s in a mimicry attack. We have added a new object to the analysis infrastructure that enables such analyses. The *data*

	Monitored Call String:	Number of Null Calls:
(a)	**open**, B, $\overline{\text{B}}$, B, $\overline{\text{B}}$, B, **read**, $\overline{\text{B}}$, B, $\overline{\text{B}}$, B, **read**, $\overline{\text{B}}$, B, $\overline{\text{B}}$, B, $\overline{\text{B}}$, C, **write**, $\overline{\text{C}}$, D, $\overline{\text{D}}$	18
(b)	**open**, B, **read**, $\overline{\text{B}}$, B, **read**, $\overline{\text{B}}$, C, **write**, $\overline{\text{C}}$	6

Table 1. System call strings accepted by the Dyck model. These strings correspond to possible paths in Figure 5 and Figure 7. (a) A possible path accepted by the context-free Dyck model. (b) The string in (a) with null call squelching. Note the large drop in observed null calls.

dependence graph (DDG) represents complex interprocedural data flows and is described in Appendix C. Section 5.1 presents argument capture, a method to recover statically-known arguments. Branch analysis, explained in Section 5.2, uses the DDG to identify branch conditions dynamically set by system call return values. With both argument and branch analysis, we reduce the opportunities for a successful mimicry attack.

5.1. Argument Capture

To prevent an attacker from manipulating arguments passed to a system call, we use the DDG to recover statically-known arguments. Our analysis recovers statically-known data values using a two step process. First, it follows paths in the data dependence graph to collect the expression graph for the value. Second, it simulates the execution of the instructions in the expression graph to determine the value. If analysis cannot reliably construct the expression graph or if a value is not statically known, the analyzer marks it as unknown. Multiple execution paths may set arguments differently, so we recover sets of integers, set of regular expressions for string arguments, and dependencies upon a return value from a previous system call. This interprocedural approach is more general than the constant-valued intraprocedural capture described in previous work, further restricting the possibilities for successful attacker manipulation.

Importantly, these argument recoveries help prevent mimicry and evasion attacks [26, 27, 28, 33]. Consider the `read` system call transition in Figure 2. With argument recovery, we can replace the transition `read(?,buf,?)` with `read(=open,buf,{16,128})`. The first argument is the return value from `open`, and the third argument is the set of values {16,128}. An attacker could not transform this `read` call into a *nop* because argument recovery prevents the necessary manipulation.

5.2. Branch Analysis

A mimicry attack works well because the attacker can easily generate *nop* system calls to steer model operation as needed. These *nop* calls use invalid arguments to force the call to fail and not change system state. Failed system calls return an error indicator so that legitimate programs may take any necessary corrective action. If the monitor does not track these return values and some system call arguments are unknown, the attacker can undetectably cause the system calls to fail. Branch analysis detects such manipulation.

Our analysis determines the expected subsequent process execution based upon the return value of a system call. We insert *predicate transitions* into the automaton that indicate control dependencies upon return values. At runtime, the monitor records return values and traverses any edge with a predicate that evaluates to *true* as if it were an ϵ-transition. It ignores any edge evaluating to false. If an attacker uses a *nop* call to steer execution, that call must be followed by system calls that match the error case behavior in the actual application.

For example, the DDG reveals that the branch instruction in line 24 of Figure 1(a) is based upon the return value of `open`. We insert predicate transitions into *action*'s model corresponding to the branch behavior (Figure 8). Should an attacker use the `open` call as a nop by specifying an invalid argument, the monitor would detect an intrusion if `read` were the next symbol. The failed `open` call blocks the path to the `read` call via its return value. Thus, branch analysis helps prevent development of successful attacks.

6. Evaluation

We evaluate our program models with two criteria: *precision* and *efficiency*. Precise models present an attacker with little opportunity to insert malicious system calls. An efficient model adds only a small runtime overhead to the existing process execution. Only efficient models will be deployed, and only precise models add security value. Precise models generally have higher runtime overhead. We demonstrate that the Dyck model with squelching presents an excellent tradeoff between precision and efficiency.

6.1. Metrics

We use standard techniques to measure these criteria. The *average branching factor* metric, originally developed by Wagner and Dean [31, 32], measures model precision. Average branching factor is a dynamic measure of an adversary's opportunity to insert dangerous system calls into a running process's call stream. As the monitor operates

				Call Sites	
Program	*Workload*	*Functions*	*Instructions*	*System*	*User*
procmail	Filter one 1 MB message to a local mailbox.	1,619	112,951	203	8,166
gzip	Compress a 13 MB text file.	884	56,710	96	2,746
eject	Open the CD-ROM drive tray.	1,039	70,177	159	3,903
fdformat	Format a high-density floppy disk.	957	67,874	197	3,767
ps	Report process status of all processes.	963	59,814	96	3,301
cat	Concatenate 38 files totaling 500 MB to a file.	838	52,028	108	2,615

Table 2. Test programs, workloads, and statistics.

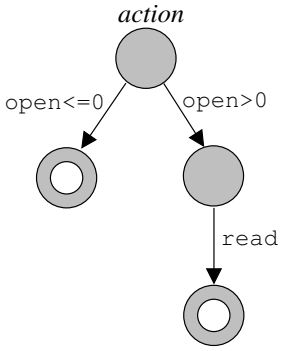


Figure 8. The model for *action* with branch analysis.

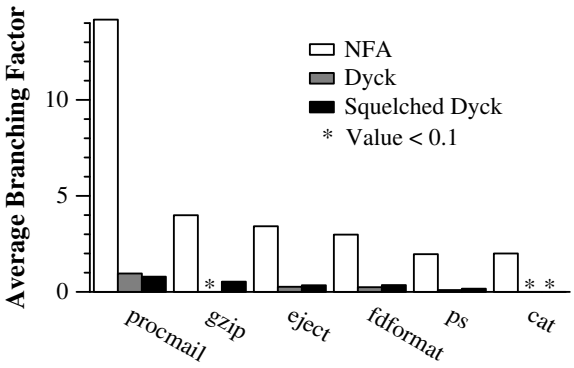


Figure 9. NFA and Dyck precision. Lower bars indicate greater precision.

the automaton model, it records all potentially dangerous system calls that it would accept as the next call. The average branching factor is then the total number of these calls divided by the number of automaton updates performed by the monitor. A low average branching factor indicates that an attacker has little opportunity to undetectably insert malicious system calls into the call stream.

Efficiency measurements are straightforward and take two forms. First, we time the length of process execution with and without model operation. Second, we measure each process's runtime memory usage increase due to binary code instrumentation and the model state kept in the monitor.

6.2. Experimental Design

We include precision and efficiency results for six test programs. Table 2 shows the workloads used for each program. Note that experiments using `ps` are not reproducible because its execution depends upon constantly changing system state. Table 2 also gives statistics for the binary code of each program. The number of user function call sites indicates the level of interprocedural control flow transfers and the worst-case number of Dyck instrumentation points. We currently analyze statically-linked binaries, so these statistics include linked library code.

These test programs and our runtime monitor run on Solaris 8 on a Sun Ultra 10 440 Mhz workstation with 640 MB of RAM. The monitor and test processes run simultaneously on the same machine. We have not yet implemented support for kernel trap monitoring, so the application communicates with the monitor with a shared message queue for the purposes of our experimentation. The collection of Solaris libc kernel trap wrapper functions defines our set of system call events.

We measured precision and efficiency for all six programs. The monitor calculates the average branching factor for every test program using the method described above. To determine runtime overheads, we use the UNIX `time` program to measure the wall time elapsed during execution of the test program. The test program and the monitor execute on the same machine, so this time includes test process execution, monitor execution as it operates the automaton, and context switches between the processes. The time does not include setup time in the monitor, in which it parses the program model from a file. We measure memory usage by recording the maximum process image size, observed at every return from the `brk` kernel trap. We ran experiments on a lightly loaded multi-user machine with no other active users.

6.3. Effects of the Dyck Model

We analyzed how the Dyck model influenced precision and efficiency. We compared the Dyck model with and

Program	*Base*	*NFA*	*%*	*Dyck*	*%*	*Squelched Dyck*	*%*
procmail	0.42	0.37	0	0.58	38	0.40	0
gzip	7.02	6.61	0	610.64	8600	7.16	2
eject	5.14	5.17	1	5.19	1	5.22	2
fdformat	112.41	112.36	0	112.22	0	112.38	0
ps	0.05	0.05	0	0.14	180	0.09	80
cat	54.65	56.32	3	895.67	1539	80.78	48

Table 3. Program execution times in seconds. The base execution time has no automaton operation. Percentages compare against base execution. Models had no argument recovery or branch analysis.

Program	*Unmonitored*	*(a) Infrastructure*	*(b) Instrumentation*	*(c) State Machine*	*(d) % Increase*
procmail	3272	600	104	840	29 %
gzip	600	288	56	296	59 %
eject	576	400	64	248	54 %
fdformat	600	368	80	408	81 %
ps	520	360	56	264	62 %
cat	496	328	32	72	21 %

Table 4. Memory use (KB) due to instrumentation and monitoring. Unmonitored is base-case execution of the unmodified programs. Columns (a)–(c) show additional use due to the rewriting infrastructure, our instrumentation, and the state machine structure in the monitor. Column (d) shows percentage increase compared to the base case.

without squelching against the NFA model used in our previous work [10]. Figure 9 shows the precision of the three models for all six test programs. Note that the Dyck models improve precision by an order of magnitude. For example, `procmail` improves from an average branching factor of 14.2 using the NFA model to 0.79 with the squelched Dyck model.

The squelched model appears to be slightly less precise than the unsquelched Dyck model. However, this is a side effect of the average branching factor calculation. Recall that the monitor divides the number of potentially dangerous system calls that could be accepted during execution by the number of automaton operations. The average branching factor is then inversely proportional to the number of events passed to the monitor. A squelched program will generate fewer null calls than an unsquelched program, leading to a slight increase in the average branching factor.

Table 3 presents execution times for the various models. Measurement noise accounts for slight timing variations. Note the marked improvement when the Dyck model includes squelching. The squelched Dyck model produced 2–5 times more system calls than the NFA model, depending upon the program. With the exception of `cat`, the performance impact of the additional calls is not significant. For a system-call-bound program such as `cat`, the additional time consumed by null calls becomes noticeable. We expect that performance could be markedly improved by batching Dyck calls and sending them with actual system calls to minimize the number of user-to-kernel transitions. Squelching is critical: the unsquelched `gzip` model generated 12,800 times more system calls than base execution due to loop iteration.

We also measured the memory overhead of monitoring with the squelched Dyck model (Table 4). This overhead has two parts: the memory needs of the monitor process and the increased size of the instrumented executable. The monitor is the same across all processes, differing only in the state machine read from file. Thus, the monitor's code and static data, 1736 KB, is a one-time cost shared across multiple executions. Approximately 1 MB of this code resides in shared libraries likely already used by other processes on the system. State machines are not shared, and their memory sizes are shown in column (c).

Instrumented binaries use additional memory for two reasons. First, program size increases as an artifact of our current rewriting infrastructure (column (a)). This overhead will disappear as we transition to our new rewriting environment (already in use of other areas of our project). Second, null call insertion adds code to the program, as shown in column (b). Column (d) shows the percentage increase due to our instrumentation and to the state machine structure in the monitor. Memory demands become more critical when we wish to monitor large numbers of processes on a system. We have identified several areas in which we can make substantial optimizations in our memory usage. For example, column (c) might be reduced by more efficient encodings of our state machines. Although we are unaware of published memory needs in related projects, we believe our results would compare favorably.

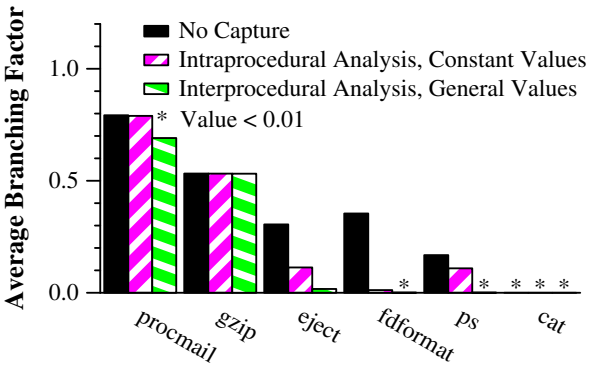


Figure 10. Effects of argument capture. Lower bars indicate greater precision. All bars use the squelched Dyck model. The black bars correspond to the black bars in Figure 9. Note the average branching factor scale has changed by an order of magnitude from Figure 9.

6.4. Effects of Argument Capture

We believe the squelched Dyck model represents the best tradeoff between precision and efficiency. We used this model to investigate the effects of improved argument capture. We tested argument capture in three forms. First, all argument capture was turned off. Second, we recovered only arguments set intraprocedurally with a single constant value, corresponding to our previous work [10]. Finally, we enabled the complete recovery technique that uses interprocedural analysis to recover general representations of call arguments. Figure 10 shows the model precision at each level of capture for all test programs.

6.5. Data Flow Analysis in Support of Mimicry Attack Detection

Mimicry and evasion attacks exploit some deficiency of a program's model so that the monitor accepts an attack system call sequence as valid [26, 27, 28, 33]. Tan *et al.* and Wagner and Soto stress the need to monitor system call arguments and return values for mimicry attack prevention. As our first hardening against mimicry attacks, we have implemented branch analysis and have extended argument capture to general values passed interprocedurally. Figure 10 shows the argument capture improvement, and Table 5 shows the results for branch analysis. Average branching factor is poorly suited to measurement of branch analysis, so the number of call sites affecting branching is advisory only.

The results appear promising. System call sites that set branches are those whose return value affects program control flow. Our branch analysis identifies between 32% and 64% such system calls in the test programs. Constant-valued intraprocedural argument capture corresponds to previous work. Our capture recovers general arguments passed interprocedurally. This stronger analysis recovers between 32% and 69% more arguments, depending upon the test program.

Program	*System Call Sites Affecting Branches*
procmail	97
gzip	54
eject	101
fdformat	103
ps	44
cat	45

Table 5. Branch analysis results. Table 2 lists the total number of system call sites per program. Here, the data indicates the size of the subset of system call sites whose return value affects program branching.

These are clearly partial results providing only an early indication of effectiveness against mimicry attacks. Our current work in mimicry attack detection and prevention is based upon analyzing the attacks as a language containment problem [33]. Formally, given the language L of system call sequences accepted by the monitor, we must determine if L contains one or more attack sequences. This study requires further investigation and is one component of our continuing research.

7. Conclusions

The Dyck model is an efficient context-sensitive program representation. Our experiments show that such context-sensitive models significantly improve the strength of an intrusion detection system. With null call squelching, the Dyck model operates with efficiency only slightly worse than an imprecise context-insensitive NFA. This makes context-sensitive models usable. Lastly, interprocedural argument capture and branch analysis based upon system call return values limit attacker manipulation, reducing opportunities for successful attacks.

Acknowledgements

We thank the anonymous referees and the other members of the WiSA security group at Wisconsin for their valuable feedback and suggestions.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[2] 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.

[3] N. Chomsky and M. Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, Studies in Logic and the Foundations of Mathematics, pages 118–161. North-Holland Publishing Company, Amsterdam, 1963.

[4] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *12th Conference on Computer Aided Verification (CAV), LNCS #1855*, pages 232–247, Chicago, Illinois, July 2000. Springer-Verlag.

[5] 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.

[6] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[7] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

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

[9] 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.

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

[11] S. Ginsberg and M. Harrison. Bracketed context-free languages. *Journal of Computer and System Sciences*, 1:1–23, 1967.

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

[13] 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.

[14] 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.

[15] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–218, Williamsburg, Virginia, January 1981.

[16] 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.

[17] 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.

[18] U. Lindqvist and P. Porras. eXpert-BSM: A host-based intrusion detection solution for Sun Solaris. In *17th Annual Computer Security Applications Conference (ACSAC)*, pages 240–251, New Orleans, Louisiana, December 2001.

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

[20] S. Maheshwari. Traversal marker placement problems are NP-complete. Technical Report CU-CS-09276, Department of Computer Science, University of Colorado, Boulder, Colorado, 1976.

[21] K. Ottenstein. *Data-Flow Graphs as an Intermediate Program Form*. Ph.D. dissertation, Purdue University, August 1978.

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

[23] 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.

[24] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Security Symposium*, Washington, DC, August 1999.

[25] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, New York, October 2003.

[26] 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.

[27] K. Tan and R. Maxion. "Why 6?" Defining the operational limits of stide, an anomaly based intrusion detector. In *IEEE Symposium on Security and Privacy*, pages 188–201, Oakland, California, May 2002.

[28] 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.

[29] 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 1999.

[30] W. von Dyck. Gruppentheoretische studien. *Mathematische Annalen*, 20:1–44, 1882.

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

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

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

[34] 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.

[35] 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. VtPath Attack

The VtPath model fails to detect impossible path attacks based upon non-determinism that the Dyck model can detect due to null call insertion. Consider the code in Figure 11. The function `security_check` verifies the current process id and allows the root user to access a file but denies access to all others. The function `log` writes activity to a log file and has a buffer overrun at line 13.

The attack works as follows: an attacker without root privilege enters `log` via the call at 7. They overflow the buffer in `log` to set the return address to the return of line 4. `log` will then return to line 5 and execute the privileged actions.

VtPath will not detect this attack. VtPath observes return addresses at each system call point, here, after the return address has been modified. The `if` at line 3 is a point of non-determinism, leading VtPath to incorrectly believe that the call to `log` originated at line 4. A push-down automaton model would similarly miss the attack.

The Dyck model detects the attack. Both calls to `log` would be instrumented with different pre- and postcalls. In particular, at the point of the buffer overrun, the correct return address has already been stored. The null calls before each call site thus aid attack detection by reducing nondeterminism when such coding patterns arise. We note that the VtPath model would detect this attack if it learned a behavioral database from a program with Dyck instrumentation previously inserted.

B. Formal Definitions and Proofs

DEFINITION 1. Denote local NFA models by $A_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, F_i)$ where i ranges over all functions in the program. Let τ be the entry point function. For each i, let S_i be the set of system calls and U_i the set of user functions called by i. Then $\Sigma_i = S_i \cup U_i$. Define the Dyck model as $D = (\cup_i Q_i, \Sigma, \Gamma, \delta, q_{0,\tau}, \emptyset, F_\tau)$ for $\cup_i Q_i$ the set of states, Σ the input alphabet, Γ the stack alphabet, δ the transition relation, $q_{0,\tau}$ the unique entry state, $\emptyset$ the initial stack configuration, and F_τ the set of accepting states, with:

- $\Gamma = \left\{ q \,\middle|\, p \xrightarrow{\beta} q \in \delta_i \text{ and } \beta \in U_i \right\}$
- $\Sigma = \left(\bigcup_i S_i \right) \cup \Gamma \cup \overline{\Gamma}$ where $\overline{\Gamma} = \{ \overline{q} \,|\, q \in \Gamma \}$.
- *System call transition*: $\delta(q, \alpha, \epsilon) = (p, \epsilon)$ if $q \xrightarrow{\alpha} p \in \delta_i$
- *Precall before* β; pushes p onto stack: $\delta(q, p, \epsilon) = (r, p)$ if $q \xrightarrow{\beta} p \in \delta_i$ and r is the entry state of β
- *Postcall after* β; pops p from stack: $\delta(r, \overline{p}, p) = (p, \epsilon)$ if $q \xrightarrow{\beta} p \in \delta_i$ and $r \in F_\beta$

Then D models the system call sequences generates by the application with a bracketed context-free language. The subsequences of D consisting entirely of symbols from $\Gamma \cup \overline{\Gamma}$ form a Dyck language.

LEMMA 1. In the squelched Dyck model, a postcall follows either a true system call or a postcall.

PROOF. Suppose a postcall t follows a precall r. Then r was at the top of the application's squelch stack. By construction, squelching removes both r and t from the call stream. □

LEMMA 2. Let $c_0, \ldots, c_n$ be an observed sequence of calls where c_0 is a true system call, c_n is a true system call, and $c_1, \ldots, c_{n-1}$ are null calls.

Let c_i, $1 \leq i \leq n-1$ be the first precall. Then c_j is a precall for all $i < j \leq n-1$.

PROOF. Suppose not. Then $\exists i < k \leq n-1$ such that c_k is the first postcall in $c_i, \ldots, c_{n-1}$. Then c_{k-1} is a precall, contradicting Lemma 1. □

THEOREM. Let C be the call graph for program P. Denote by $\widetilde{C}$ the graph obtained from C with each strongly connected component collapsed to a single state. Let h be the maximum diameter of $\widetilde{C}$. If P generates n true system calls during execution, then the worst-case number of null calls generated is $2hn$.

PROOF. From Lemmas 1 and 2, it follows that the observed call pattern is a repeating sequence of a string of precalls followed by a system call followed by a string of postcalls. We claim the precall string and the postcall string each have length at most h.

For a given system call, suppose the precall string has length $l > h$. Then there exists a directed path in $\widetilde{C}$ of length l, which cannot occur.

Suppose the postcall string has length $m > h$. Then there exists a directed reverse path in $\widetilde{C}$ of length m, which similarly cannot occur.

Therefore, the number of null calls generated is $\leq 2h$ per system call. □

C. Data Dependence Graph

The data dependence graph (DDG) is a common program analysis structure representing interprocedural flows of data through a program [15, 21]. The DDG is a subgraph

```
 1  void security_check (char *file) {
 2    uid_t uid = getuid();
 3    if (uid == 0) {
 4      log("Accessing %s", file);
 5      restricted_access(file);
 6    } else {
 7      log("Invalid access %s", file);
 8      exit(SECURITY_ERROR);
 9    }
10  }
11  void log (char *msg, char *file) {
12    char buf[100];
13    sprintf(buf, msg, file);            <-- Buffer overflow
14    write(LOG_FD, buf, strlen(buf));
15  }
```

Figure 11. Code for VtPath attack.

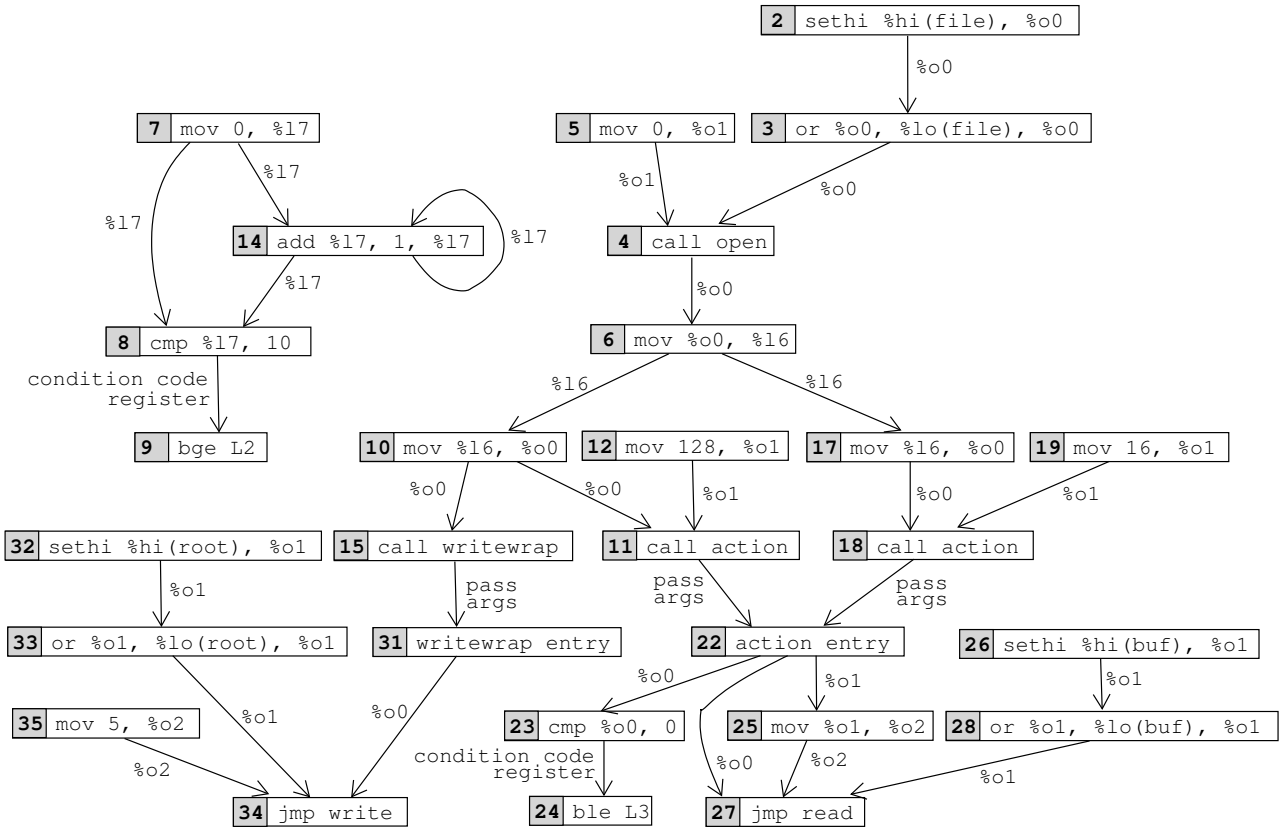


Figure 12. Data dependence graph. Our analysis constructs this data dependence graph for the code in Figure 1(a). The shaded numbers correspond to the line numbers in Figure 1(a). In SPARC code, the rightmost register in an instruction is the written register. Registers `%o0`–`%o5` contain call arguments. `%o0` contains the return value of a call.

of a program dependence graph [6] that includes only data flow dependence edges. The graph abstracts away procedure and basic block boundaries, so each instruction is a node in the graph. Edges indicate data flowing from an instruction P_i that may write to a data location L to instructions P_j that may read from L. Such a flow exists only when there is a def-clear path from P_i to P_j with respect to L. For convenience, each edge label indicates the data location creating the dependency. Furthermore, a DDG includes interprocedural data flow edges. Interprocedural edges indicate data dependencies between the definition of arguments and the entry point of a function that uses those arguments and between the exit point of a function and a use of the return value.

DEFINITION 2. Let I be the set of instructions in a program P and N be the set of function entry points. Define the data dependence graph G for P to be $G = \langle I \cup N, E \rangle$ where $P_i \xrightarrow{L} P_j \in E$ if there is a def-clear path from P_i to P_j with respect to L.

Consider an example. Figure 12 shows the DDG constructed for the program code in Figure 1(a). Shaded node numbers correspond to line numbers in Figure 1(a). SPARC delay slots are unwound, so node 5 precedes node 4 in the graph.

With this DDG, argument capture becomes straightforward. The subgraph of the DDG rooted at a system call instruction reachable by following reverse edges for the dependent data location is the expression graph setting the argument value. For example, argument 2 (register `%o1`) to the `read` instruction in node 27 has nodes 26 and 28 in its expression graph. By simulating the execution of these instructions, we can identify the buffer passed to `read`. Similarly, the expression graph for the third argument (register `%o2`) to `read` includes nodes 12, 19, 11, 18, 22, and 25. Note that this represents an interprocedural data flow. Simulated execution recovers both values 128 and 16 for this argument.

Branch analysis and argument capture for system call return values requires a slight change to this procedure. In particular, discovery of the expression graph stops at the return value of a system call. The expression graph for argument 1 (register `%o0`) to the `read` instruction in node 27 reaches back to node 4. Here, analysis recognizes that `open` is a system call and marks the argument as using `open`'s return value.

Analysis of the branch in node 24 proceeds similarly. The expression graph reveals that the return value from `open` in node 4 is compared against 0 in node 23. Given the branch condition *branch less or equal*, the predicates added to the program model are `open` ≤ 0 for the branch-taken control flow path and `open` > 0 for the fall-through path, as shown in Figure 8.