Model-Based Intrusion Detection Design and Evaluation

Jonathon T. Giffin

September 21, 2004

1 Introduction

Host-based intrusion detection identifies attempts to maliciously access the machine on which the detection system executes. Remote intrusion detection identifies hostile manipulation of processes executing in a distributed or remote execution environment. These intrusion detection systems monitor processes and flag unusual or unexpected behavior as malicious. In model-based detection [48, 75], the system has a model of acceptable behavior for each monitored process. A monitor compares a running process's execution with the model and flags deviations as intrusion attempts.

My research examines two aspects of model-based intrusion detection. In previous work, I first investigated model construction techniques. I developed a new model constructed from static binary analysis that accurately represents process behavior [52]. In this proposal, I add new data flow analyses to further restrict allowable behavior. Second, I turn to attacks. I will design a formal framework that will automatically construct attacks and their variants. I can then evaluate the effectiveness of an intrusion detection system by identifying attacks that are undetected by the system.

Constructing a valid and precise program model is a challenging task. Previous research has focused on four basic techniques for model construction: human specification [75, 136, 154], training [44, 91, 97], static source code analysis [161], and static binary code analysis [51]. Of these, I use static binary code analysis because it requires no human interaction, no training data, and no access to a program's source code, although it is poorly suited for interpreted-language analysis. It constructs models that contain all possible execution paths that a process may follow, so false alarms never occur. However, an imprecise model may incorrectly accept attack sequences as valid. I use static binary analysis to construct a finite state automaton 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, context-sensitive push-down automata (PDA), are prohibitively expensive to operate [51, 160, 161]. For example, Wagner and Dean suggested the use of their less precise digraph model simply because more precise models proved too expensive. My earlier work used regular language overapproximations to a context-free language model, again due to cost. This proposal describes a new model structure that does not suffer from such drawbacks. My 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 further suited for *remote intrusion detection* [51]. 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, I can detect remote manipulation that produces invalid call sequences.

Development of an intrusion detection system means little without proper techniques to evaluate the strength of the system. Previously published evaluations have been ad hoc. Wagner and Soto demonstrated that launching existing attacks against processes protected by a detection system system is insufficient as it reveals nothing about the system's responsiveness to novel and previously unseen attacks [162]. Their *mimicry attack* modifies a known, detected attack into a semantically equivalent sequence that mimics normal process behavior and is undetected.

A gap exists between these theoretical discussions of attacks that intrusion detection systems should detect and actual demonstration of a system's detection capability. In the second half of my proposed research, I will design formal methodologies for security evaluation of model-based intrusion detection systems. The operating system will be modeled as a collection of variables that together describe its current state. Certain configurations of the variables are undesirable or unsafe and reflect the effects of malicious activity. System calls act as transformers of the operating system state. A process begins execution when the system is in some known, initial configuration. A successful attack against the process forces it to generate a sequence of system calls that takes the operating system into an unsafe state. The formal notions of state variables and system call transformers enable construction of and formal reasoning about attacks.

Together, these research directions will provide the following contributions to the security field:

- **The Dyck model: efficient, context-sensitive program modeling.** The Dyck model is a substantial improvement in statically constructed program models. Exposing call stack changes to the monitor makes operation highly efficient by limiting state exploration to only the exact execution path followed by the application. By correctly modeling function call and return semantics, the model can detect an important class of attack, the impossible path exploit [160, 161]. Section 3 describes how static binary analysis constructs program models and presents the Dyck model.

- **Interprocedural data-flow analyses for binary code.** Data-flow analysis identifies values computed during program execution, with applications to fields such as program understanding and security policy design. I previously developed new interprocedural analyses to model: (1) arguments passed to system calls, (2) return values received from system calls, and (3) predicates used at branch instructions. These analyses are designed to eliminate mimicry attacks by restricting the paths in the program model that accept an attack sequence. I discuss data-flow analyses in Section 4.1.

- **Incorporating environment information into statically-constructed models.** Static data-flow analysis can recover only values and dependencies present in the static program. However, a process's execution may be restricted by environment state external to the process, such as command-line parameters, configuration file settings, and environment variables. In a program model, these values may control system call arguments and branch predicates. External configuration information is safe to include in a statically constructed model because such information remains constant for all program executions. Analysis that incorporates external features produces a program model that more accurately represents possible execution behavior.

  I will show how a new analysis technique: *differential dynamic analysis*, can enhance existing static analysis with data- and control-flow restrictions imposed by the environment. I show how to compose differential analysis with models constructed using static analysis in Section 4.2.

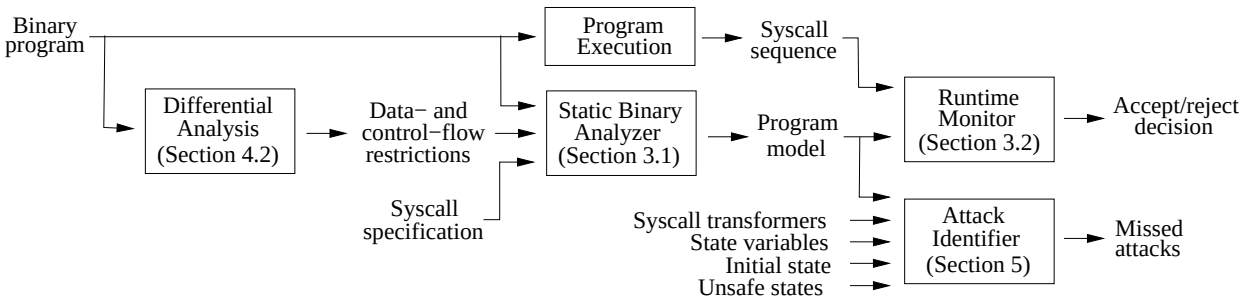

Figure 1: Connections among proposed research components.

- **A formal attack model for system-call-based intrusion detection systems.** Security is too often a responsive research field. After a new exploit is discovered, researchers respond with new detection techniques. For example, I originally designed the Dyck model to counter the new impossible path attack [161]. Research becomes an undesirable escalating cycle between attack discovery and detection system design.

  I will develop a formal framework for attack creation and IDS evaluation that will break this cycle. By formally modeling system calls and their effects upon stored operating system state, I can generate sequences of system calls that take the system from a safe initial state to an unsafe state satisfying some attack goal. These attacks are linked to goals rather than particular exploit methods and thus describe attacks that may be yet undiscovered. Flaws in intrusion detection systems can then be identified well before an exploit opportunity ever occurs. Section 5 describes my framework for attack creation and system evaluation.

- **Automatic attack obfuscation.** The formal attack generation framework produces obfuscated attacks. Obfuscations modify an attack by changing the system call sequence while maintaining the original attack semantics. Intrusion detection systems must detect all obfuscated versions of an attack.

  There are two general classes of obfuscations. First, any system call sequence that does not interfere with attack state may occur at any arbitrary point of the attack. This class includes failed system calls. Second, obfuscations may include system call sequences that alter attack state provided that the attack adjusts to use the altered state. For example, a file used by the attack could be closed and reopened before its next use. The attack must change to use a new file descriptor after the file is reopened. The model-checking algorithms with system call argument unification, described in Section 5, automatically generate all possible attack obfuscations.

These contributions produce a unified research goal: formal analysis of program models and attacks for evaluation of system-call-based intrusion detection systems. Figure 1 illustrates the connections among the proposed pieces of research. The model builder statically analyzes a binary program to produce a model, here the Dyck model, of its system call sequences. External data flow analysis further restricts the model. A runtime verifier checks the execution of the program at every system call against the model. The attack identifier uses the formal framework defining system call transformations of operating system state to find attacks accepted as valid execution by the program model. Section 6 summarizes these proposed works and includes a timeline for completion.

2 Related Work

Host-based intrusion detection is a highly active research area. Section 2.1 presents projects in the complementary fields of misuse and anomaly detection [9] and shows how the Dyck model, an anomaly detector, advances the existing research. In contrast, few studies of detection effectiveness and resilience to intelligent attackers have been conducted. Section 2.2 discusses the small body of existing work and some of the model-checking techniques I expect to use in my proposed evaluation.

2.1 Host-Based Intrusion Detection

Prior to 1980, intrusion detection was entirely a manual endeavor [69]. A site administrator would periodically review audit reports containing summaries of user behavior. Strange audit records, such as a user logging in locally when the administrator knew her to be on vacation, indicated that the account may have been compromised. Administrators could identify such obvious intrusions, but an intelligent attacker could often penetrate a system undetected [147]. Furthermore, user bases and networks grew large and manual review failed to scale.

2.1.1 Misuse Detection

Misuse detection automates the identification of intrusions in audit data. Misuse detectors (or *knowledge-based detectors* [29]) contain a database of known malicious event patterns and search event streams for occurrences of the patterns [11, 48, 55, 57, 82–84, 103, 104, 111]. STAT [34, 61–63, 144, 157, 158] and LAMBDA [57] specify attacks as a sequence of steps from an initial state to the malicious state produced by the attack but require an administrator to manually construct the sequences. Sekar *et al*. specified signatures as system calls sequences [134, 135]. They realized the importance of modeling system call arguments and modeled argument dependencies in each call sequence [137, 138]. However, these signatures could not encode the calling context of a system call and, again, were manually written.

Misuse detectors excel at identification of known attacks, but have important limitations:

- **They cannot detect new and novel attacks**. Misuse detectors are limited by their database of malicious event patterns. An emerging new attack or a variant of an existing attack will not match any existing pattern in the database, so the detector will miss the attack. Indeed, a DARPA-sponsored evaluation of intrusion detection systems suggests that "further research should focus on developing techniques to find new attacks" [106].

- **They require a second detection methodology**. This is a chicken-and-egg problem: detecting an exploit with a misuse detector requires a signature for the exploit, but building a signature requires that the exploit first be detected and given to a human for investigation. This circularity is broken only with a second detector that is not limited by a known signature database.

- **The system audit information available to the detector is often insufficient for accurate detection** [123]. A detected pattern may represent an exploit in some execution contexts and legitimate behavior in others. If the context is unavailable to the detector, it will generate incorrect responses.

2.1.2 Anomaly Detection

Anomaly detectors (or *behavior-based detectors* [29]) address the problems of misuse detection by removing the dependence upon the known-attack database. They instead characterize correct, expected behavior and

flag deviations from normal as possibly intrusive. Research in anomaly detection began with the insights that normal user behavior can be characterized as a statistical model and that intrusions often manifest as departures from the model [5]. The detector filters out non-anomalous audit data, reducing the manual workload upon the site administrator.

The model of normal behavior is the critical component of an anomaly detector. A too restrictive model does not capture all legitimate behavior and will produce false alarms. Relaxing the model admits more behavior and may fail to detect an intrusion. Early anomaly detection systems modeled user accounts [4, 27, 31, 32, 56, 65, 66, 87–89, 109, 110, 112, 116, 121, 139, 142, 156] and attempted to accurately model normal behavior [27, 28, 32, 85, 90, 148, 153]. As anomaly detection shifted from detecting masquerading users to detecting incorrectly executing processes, three methods to construct program models emerged: manual specification of correct behavior; dynamic analysis, or learning from past execution traces; and static analysis of the program's source or binary code.

In human-specified modeling, a security analyst manually specifies correct behavior for each program of interest [73, 75–77, 136, 154] or annotates the source code to describe security properties [7, 35]. These systems are reasonable for small programs; however, as programs grow, human specification becomes tedious. Dynamic and static program analysis scale better by automatically constructing models.

Dynamic program analysis constructs program models from observed behavior during repeated training runs [91, 108]. Previous work has viewed learning in an immunology framework [43, 44, 58, 59, 79, 145, 164] or as a data mining problem [95–101, 167]. Models used information-theoretic measures such as conditional entropy and information gain [102], Bayes' networks [80], neural networks [50], Markov chains [67, 166], hidden Markov models [163], genetic programming [24, 26], and finite state automata [133].

Feng *et al*. [41] and Oka *et al*. [118] extended the work of Sekar *et al*. [133] to learn sequences of system calls and their calling contexts. Feng's *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 database defines the rule-set of a push-down system (PDS) that is operable by the efficient `post` algorithm [39]. Like the Dyck model, the VtPath model uses exposed function calls and returns to limit the state exploration at runtime.

Dynamic analysis suffers from several drawbacks:

- **Training**. Systems require training data to establish the model of expected behavior. Administrators could manually design and generate a data set based upon their expectations of correct usage. Alternatively, the system could learn from historical data, as in Lee *et al*.'s work [97]. However, if this data contains attacks, then the system will learn the attack behavior as normal [107]. An administrator must still review the data, identify attacks, and remove or mark them before training.

- **Adapting**. Training must be an ongoing process. Execution behavior may change over time, a problem called *concept drift* [86, 155]. To adapt, anomaly detectors must continually retrain to update the profile of expected behavior.

- **False alarms**. Anomaly detectors report anomalies, not intrusions [107]. Alerts raised by a learning-based anomaly detector may not have been caused by intrusive activity. Legitimate behavior not in the training set appears anomalous and produces false alarms that weaken the usefulness of the system [8].

- **Missed attacks**. Likewise, not all intrusions produce anomalous behavior. Anomaly detectors miss attacks similar to the profile of normal traffic [107].

Static code analysis addresses the first three problems by extracting a model of correct execution from the program's underlying structure. Research efforts attempt to accurately model the program and restrict

intrusive behavior to minimize the fourth problem of missed attacks. A model can be described as either context-sensitive or context-insensitive. The more accurate context-sensitive models monitor not only the system calls generated by the process but also the calling context in which each system call occurs. This comes at a cost: additional state slows the monitor and hence the execution of the process at runtime.

Wagner and Dean statically analyzed C source code to extract both context-insensitive and context-sensitive models [160, 161]. They introduced the impossible path exploit, a vulnerability in the context-insensitive models. A context-insensitive model includes paths originating from one function call site but returning to a different call site. The execution context of the function call is lost. A correctly executing program could never follow such a path due to its call stack; however an attacker could force impossible control flow via a program exploit. Context-sensitive models maintain calling context information to prevent impossible path exploits, and hence more precisely model the correct executions of the program.

Unfortunately, the cost to operate Wagner and Dean's precise context-sensitive *abstract stack* model was prohibitively high and unsuitable for practical use. I observed similar expense when using context-sensitive push-down automata (PDA) constructed via static analysis of SPARC binary code [51]. The additional state information maintained in the precise models required expensive algorithms to verify that each system call was valid. Both papers recommended using imprecise context-insensitive models to achieve reasonable performance.

2.1.3 Exploit Detection

Misuse and anomaly detectors identify the malicious effect of an exploit as intrusive. *Exploit detectors* attempt to identify the actual exploit of a program vulnerability such as a buffer overflow, heap overflow, or format string attack.

Buffer overflows or *stack smashing attacks* [1] that overwrite a return address enable code injection attacks and return-into-libc attacks [115]. Overwriting a stored function pointer allows the attacker to change the target of an upcoming indirect function call. Ad-hoc tools attempt to prevent execution of injected code [74, 143] or detect changes to the return address [10, 16, 23, 45, 122] or to a pointer value [22]. These protections require costly system calls, particular hardware architectures, or program recompilation.

Heap overflows corrupt heap management information [6, 68] to produce invalid control flow. Robertson *et al*. modify heap memory allocation routines so that they protect heap management information with encrypted data [130]. Unfortunately, the encrypting key resides in the application's memory space and can be accessed and read by an attacker.

A *format string attack* allows arbitrary writes to memory [117] using a `printf`-family function. FormatGuard replaces the `printf` functions with safe versions that restrict their use [21]. Unfortunately, many programs, including the often-vulnerable `wu-ftpd`, implement their own versions of formatted printing functions. FormatGuard offers no protection to these programs.

The `wu-ftpd` example highlights one of two major shortcomings of all ad-hoc solutions. By detecting particular methods of exploit, the detectors miss even slight variations of the exploit or of programming style. The expectation that all programs will be programmed unsafely in the same way is incorrect and leads to incomplete solutions. History reinforces this claim: once ad-hoc solutions are proposed, it is only a matter of time before new attacks arise and are immune to the proposals [12, 14, 165]. Second, these techniques miss entire classes of attacks, such as integer overflows [13] and C++ dynamic dispatch table (vtable) smashes [129].

2.2 Intrusion Detection Evaluation

"How do we test an intrusion detection system and measure its effectiveness?" [142]. This question, posed in 1988, remains open sixteen years later. The recent body of work in model-based intrusion detection obscures a critical detail. Rigorous methods to evaluate the security provided by each system do not exist.

Systems are predominantly tested against known, actual attacks. DARPA [70, 78, 105, 106] and AFRL [33] tested systems with a mix of simulated normal data and malicious data produced by the execution of known attacks. Puketza *et al*. simulated both normal and attack activity and offered ad hoc guidelines for attack data design [124, 125]. Debar *et al*. designed a tool to generated simulated normal data that can be mixed with attacks from a known-attack database [30].

Detection of known attacks is an obvious necessity for a detection system but is insufficient for two reasons. First, it does not demonstrate how the system will respond to new attacks or intelligent attackers. This echoes a problem of misuse detectors: the evaluation is limited by the database of known attacks. Second, the database must be updated when a new attack emerges. One research group estimates that each new attack requires one person-week to understand the attack and add it to an existing evaluation tool [113]. Removing the dependence upon the known-attack database can eliminate these two problems.

Tan *et al*. observed that an intelligent attacker can evade immunology-based anomaly detectors [149–151]. Although these detectors successfully identified known attacks, they failed to detect modified but equivalent versions of the attacks. Tan's work was incomplete and only discovered attacks that a human could manually construct. Moreover, the ideas are specific to the implementations of the immunology-based models and cannot be abstracted to general models of system call sequences.

Wagner and Soto extended Tan's work to all detection systems that model sequences of system calls [162]. They intersect a regular language defining an attack sequence of system calls with the context-free language of calls accepted by the model as valid execution. A non-empty intersection reveals attacks missed by the detector. Wagner and Soto called such attacks *mimicry attacks* because they mimic correct program execution. An attack sequence detected by the monitor may be transformed into a mimicry attack via two alterations:

1. insert non-interfering system calls into the attack sequence;
2. change the attack system calls into semantically equivalent calls accepted by the model.

As with the work of Tan *et al*., Wagner and Soto manually apply these transformations and cannot prove that they produce all possible attacks.

Chen and Wagner use similar ideas to identify unsafe and erroneous coding practices [15]. They encode rules of safe programming as finite state machines and use model checking techniques to determine the presence or absence of those rules in a PDA modeling a program. As in the earlier mimicry attack work, the state machines are manually constructed.

Recent work suggests that model checking offers the opportunity to automatically identify all possible attacks and obfuscated attacks missed by a detector without prior knowledge of attack sequences. Multi-stage attacks can be discovered by composing known atomic attacks [120, 152]. For example, Sheyner *et al*. [140, 141] use model checking methods to identify potential attacks against a network of computers. Sheyner characterizes a known vulnerability in a critical system or defense by the necessary preconditions and the resulting postconditions of an attack against the vulnerability. Given an attack goal, symbolic model checking composes these atomic attacks into all sequences that achieve the goal. Ramakrishnan and Sekar [126] model components of a single computer system and similarly model check the composition of the components to identify new vulnerabilities. Although my end goal differs from these prior investigations, I expect to use similar model checking methods to compose obfuscated attacks from the list of available system calls.

My goal, evaluation of model-based intrusion detection, is similar to a network-based intrusion detection system study by Rubin *et al*. [131], though their approach differs from my own. They formally model allowed transformations of TCP and IP data and exhaustively enumerate all possible variants of a known attack. They start from a known attack and encode obfuscation transformations known *a priori*. My proposed work targets a desired malicious program state and encodes the known state transformations of system calls. Attacks and obfuscations are discovered automatically by the model checker.

3 Intrusion Detection Using the Dyck Model

The compromise between precision and performance that plagued earlier statically-constructed program models is not a fundamental limitation of static analysis. My Dyck model, constructed using static analysis, is a precise, context-sensitive model with excellent performance characteristics [52]. It is a push-down automaton (PDA), but achieves greater efficiency than prior PDA models by eliminating stack nondeterminism. The inefficient abstract stack and PDA models suffer high cost because they model function calls and returns nondeterministically. The runtime monitor must explore *all* possible execution paths reachable from the previously observed system call point. The Dyck model exposes function calls to the monitor, allowing it to deterministically follow only the execution path taken by the process. This approach was later duplicated by Feng's VPStatic model [40].

The Dyck model addresses other problems encountered in earlier work. The model uses process call stack information, allowing for precise and contextual modeling of processes. It characterizes correct process execution behavior rather than specific vulnerability or exploit types. A monitor operating a Dyck model should thus detect attacks exploiting classes of vulnerabilities not previously known.

I assume attackers have broad abilities. In both host-based and remote execution environments, attackers can insert new code and remove or alter existing data or code of an executing process. Attackers can overwrite disk images of programs and, in remote execution, replace the entire memory image of the remote process. This attack model encompasses a large set of attacks, including impossible path attacks [160, 161], file system changes before an `exec` call, stack and heap overflows, and format string attacks. Additionally, attackers can construct the same Dyck model used by the security monitor. The strength of the detection system must then come from the resilience of the model to attack rather than from obfuscation of the program model used. Attackers must be unable to alter the execution of the monitor or the stored Dyck model.

It is critical to understand the limitations of a security technology. I have not attempted to use the Dyck model to detect exploits in the interactions of multiple processes, so it is unclear if the model is suitable for detection of these vulnerabilities. Static binary analysis alone cannot build reasonable models of programs written in an interpreted language, such as Java or SML. The Dyck model would still be suitable for such programs if static analyzers for each interpreted language were available.

The Dyck model extends a regular security automaton [2, 132] to a context-sensitive automaton describing valid process execution behavior. Each edge of the automaton is labeled with an alphabet symbol; here, a system call or stack update symbol. The automaton has final states corresponding to program points at which the process may exit. The ordered sequences of symbols on all connected sequences of edges from the entry state to a final state define the language accepted by the automaton. For a given application, the language defined by a perfect model of the application is precisely all possible sequences of system calls that could be generated by the program in correct execution with an arbitrary input, but no others.

3.1 Static Model Construction

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

1. For each function, construct a control flow graph (CFG). Currently, the Executable Editing Library (EEL) constructs CFGs in my work [93].

2. Convert each CFG into a *local model*: a non-deterministic finite automaton (NFA) that accepts all sequences of function calls and kernel traps that the function could generate under correct execution.

3. Classify function calls and insert code around function call sites to generate symbols necessary for stack-determinism. This instrumentation adds new events into the call stream, so I update local models to match.

4. Combine the collection of modified local models into a single automaton modeling the entire program. The runtime monitor enforces the program model by operating the interprocedural automaton.

Figure 2 shows code for three example functions, *mylink*, *myexec*, and *log*. Although I include C source code for readability, I analyze SPARC binary code.

The analyzer first converts each function's CFG into a local model. This is straightforward: a CFG is already a representation of a non-deterministic finite state machine with all edges unlabeled. If a basic block contains a user call or kernel trap site, I label all outgoing edges of that block with the call name. I label all other edges ϵ and convert all basic blocks into automaton states. The automaton mirrors the points of control flow divergence and convergence in the CFG and the possible streams of calls that may arise when traversing such flow paths. The ϵ-reduced and minimized local automata for the example code are shown in Figure 3a. Appendix A gives the formal definition of a local model.

Next, the analyzer adds edges to the local models around function call transitions that model the call stack changes occurring at runtime (Figure 3b). An edge before each call transition pushes a unique identifier onto the PDA stack kept in the runtime monitor; an edge after the call pops that identifier off. Each call site has a unique push and pop symbol, so the monitor can differentiate between different call sites to the same function.

Adding symbols to the event stream that distinguish each stack operation determinizes stack updates. If the binary can be rewritten, then the analyzer inserts a *history stack* into the program's data space and adds code immediately before and after each call site. The history stack records stack changes occurring since the last kernel trap. *Precall* code before call site A pushes the symbol A onto the history stack. If the call generates a kernel trap before returning, then the monitor reads all collected symbols from the history stack to identify the execution path followed in the program. *Postcall* code then adds the symbol $\overline{\mathsf{A}}$ to the now-empty history stack. If the call returns without generating a kernel trap, then the postcall code pops A from the history stack and discards it. For binaries that cannot be rewritten, the pre- and postcall automaton edges will be labeled with the return address of the function call instruction. The runtime monitor then reads return addresses from the process's call stack to generate events that traverse these edges (see Section 3.2). Currently, my static analyzer requires the binary to be rewritten.

Last, the analyzer composes the collection of local PDA at points of function calls to form the global model of the entire program. It replaces each function call transition with ϵ-edges entering and returning from the model of the target function (Figure 3c). Figure 3d shows the completed Dyck model for the example functions, after ϵ-reduction. Appendix A formally defines the model using language theory.

```
 0   mylink:
 1       save  %sp, -96, %sp
 2       sethi %hi(dest), %o0
 3       or    %o0, %lo(dest), %o0
 4       call   unlink
 5       mov   %o0, %o1
 6       call   link
 7       mov   %i0, %o0
 8       sethi %hi(relinked), %o0
 9       or    %o0, %lo(relinked), %o0
10       call  log
11       mov   9, %o1
12       ret
13       restore
```

```
/* Command line parameter */
extern char *dest;

/* Wrapper around link syscall */
void mylink (char *src) {
  unlink(dest);
  link(src, dest);
  log("Relinked", 9);
}
```

```
14   myexec:
15       save  %sp, -96, %sp
16       sethi %hi(allow_exec), %l2
17       ld    [%l2+%lo(allow_exec)], %l2
18       cmp   %l2, 0
19       be    L1
20       mov   8, %o1
21       sethi %hi(execing), %o0
22       call  log
23       or    %o0, %lo(execing), %o0
24       sethi %hi(mailconf), %o0
25       call  exec
26       or    %o0, %lo(mailconf), %o0
27   L1: ret
28       restore
```

```
/* Command-line flag */
extern int allow_exec;

/* Wrapper around exec syscall */
void myexec (void) {
  if (allow_exec) {
    log("Execing", 8);
    exec("/sbin/mailconf");
  }
}
```

```
29   log:
30       save  %sp, -96, %sp
31       sethi %hi(log_fd), %o0
32       ld    [%o0+%lo(log_fd)], %o0
33       mov   %i0, %o1
34       call  write
35       mov   %i1, %o2
36       cmp   %o0, 0
37       bg,a  L2
38       nop
39       call  exit
40       mov   1, %o0
41   L2: ret
42       restore
```

```
/* File descriptor for log file */
extern int log_fd;

/* Write messages to log file */
void log (char *msg, int len) {
  if (write(log_fd, msg, len) <= 0)
    exit(1);
}
```

Figure 2: SPARC assembly code and C source code for three example functions, *mylink*, *myexec*, and *writewrap*. I analyze binary code and include the source code only to aid comprehension of the code behavior. The variables `dest` and `allow_exec` are command-line parameters set by code (not shown) that parses command-line arguments. The variable `log_fd` is the file descriptor for the log file, set by an earlier call to **open** (not shown).

The language accepted by the Dyck model is a *bracketed context-free language* [53]. The inserted symbols at each call site correspond to parenthesis symbols in the language and form a Dyck language [17, 159]. The monitor accepts only sequences that correctly match paired pre- and postcalls. Note that this forced pairing prevents the introduction of impossible paths even when under attack. *An attacker is free to insert or change the null calls as they wish; however, the manipulations must match some correct program execution path.*

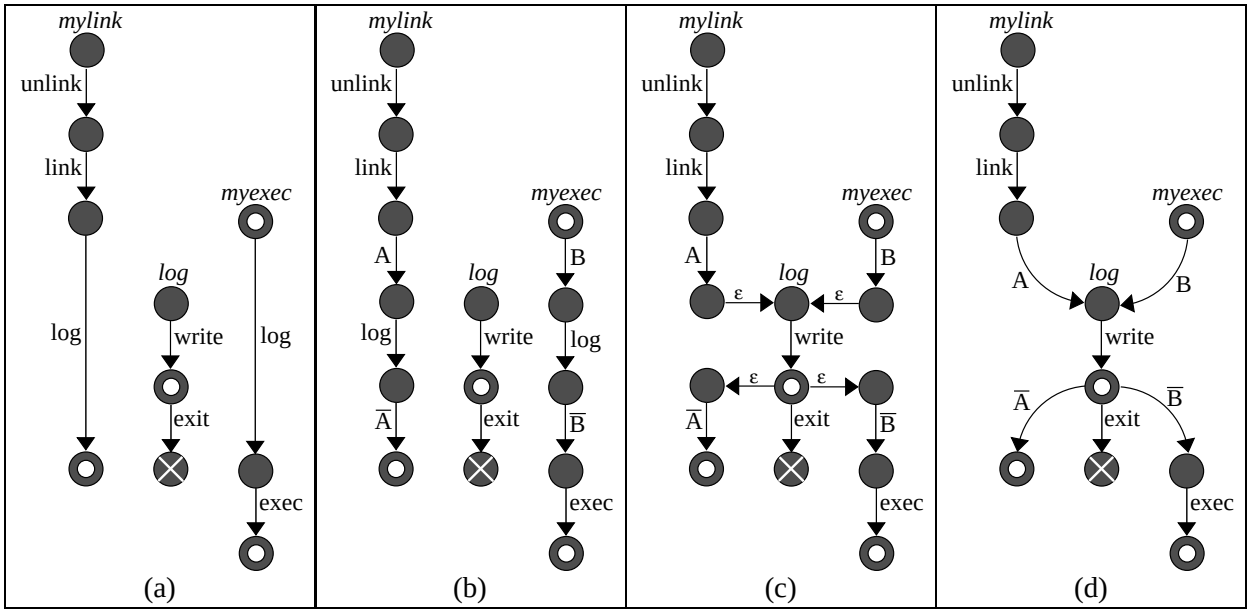

Figure 3: (a) Local function models. States with a white center represent function return points, and states with a white X represent program termination points. (b) Local models with edges modeling call stack changes. A symbol A pushes a return location; a matching closing symbol $\overline{\mathsf{A}}$ returns to the correct call site. (c) The global Dyck model after call site replacement. (d) The Dyck model after ϵ-reduction.

3.2 Runtime Monitoring

A *runtime monitor* traces execution of the process using system call interpositioning [54, 64]. When the process executes a kernel trap, the operating system invokes the monitor. The monitor generates stack update symbols and inputs the symbols to the Dyck model. Although the model is a PDA, the monitor updates only one saved stack due to deterministic stack operations. The monitor then processes the kernel trap symbol and permits execution only if the automaton still contains a valid state. Process execution thus cannot deviate from the model. Though not a focus of my research, alternative monitoring infrastructures, such as inline reference monitors [36–38], user-level delegation [47], kernel modules [46, 49, 114], or secure sandboxes [71, 72], could be used.

The monitor generates stack update symbols by reading the history stack from a rewritten binary or the call stack of an unmodified binary. The symbol A from a process's history stack pushes the identifier A onto the Dyck model's PDA stack. The symbol $\overline{\mathsf{A}}$ is an input symbol that pops A. Unmodified binaries contain no history stack, so the monitor will generate stack events by walking the program's call stack at each system call invocation to recover the saved return address in each stack frame. Differencing this list of return addresses with the list recovered at the previous system call reveals the addresses popped from and pushed onto the call stack [40, 41]. A popped return address corresponds to a postcall at a function return, and a pushed address corresponds to a precall immediately before a function call.

3.3 Preliminary Results

I have previously evaluated the Dyck model with two criteria: *precision* and *efficiency* [40, 52]. 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. The Dyck model presents an excellent trade-off between precision and efficiency.

The *average branching factor* metric [160, 161] measures model precision. It is a dynamic measure of an adversary's opportunity to insert dangerous system calls into a running process's call stream. The average branching factor is the average number of potentially dangerous system calls that the monitor would accept as a valid next event following each observed system call. A low average branching factor indicates that an attacker has little opportunity to undetectably insert malicious system calls into the call stream. For six standard UNIX programs tested, the Dyck model improved precision by an order of magnitude when compared to the weaker NFA model [52]. Efficiency tests measured process' increased execution time due to system call monitoring. For the same programs, execution slowed by 0% to 80%. These measurements are preliminary and, importantly, do not compute the level of security offered by the intrusion detection system. Section 5 proposes new evaluation methods that identify attacks missed by the program models.

3.4 Proposed Work

Although this stage of the work is nearly complete, I plan to carry out two additional tasks. First, I will improve the static analyzer so that a greater collection of programs can be analyzed. In particular, I will add support for dynamically-linked binaries and for multi-threaded programs. Second, I will investigate alternative formalizations of the Dyck model, such as a tree automaton [20] or a visibly pushdown automaton [3]. The primary goal of these two tasks is to gain a better understanding of program modeling, although the alternative formalizations may lead to more efficient monitor implementations.

4 Data-Flow Analysis

Data-flow analysis improves the Dyck model by further limiting acceptable system call sequences. The Dyck model's context-sensitivity restricts control flow paths to only those having correct call and return semantics, but it allows an attacker to undetectably manipulate data values along these execution paths. Recently developed attacks [149–151, 162] exploit this deficiency. By changing arguments to system calls on a valid system call sequence, the sequence can become malicious. These attacks demonstrate the need to model system call arguments and return values so that the program monitor can detect data manipulation.

I have designed analyses to address this need. Using both static and dynamic analysis, the algorithms identify flows of data through an executing program. The binary analyzer statically evaluates these flows to recover values or encodes them into the Dyck model for dynamic evaluation by the program monitor. Section 4.1 describes two analyses, already implemented, that use static analysis to identify data flows. Section 4.2 proposes new techniques that incorporate dynamic analysis into the statically constructed Dyck model.

4.1 Static Data-Flow Analysis

Static data-flow analysis identifies dependencies between points in the program that use and set the data. I have implemented two analyses that use this static dependency information to limit an attacker's ability to construct undetected attacks. *Value capture* recovers system call arguments when the argument value can be statically calculated. If the value flowing on a data dependency can only be calculated at execution time, then *dependency capture* encodes the dependency into the Dyck model. The program monitor will restrict allowed system call sequences based on its dynamic computation of the data value.

The *data dependence graph* (DDG) [81, 119] enables these analyses. Readers unfamiliar with a DDG may see Appendix B for details.

4.1.1 Value Capture

The analyzer recovers statically-known values to prevent system call argument manipulation. With the DDG, static value capture becomes a straightforward two-step process. First, the analyzer follows paths in the DDG to determine the expression graph for the value. The expression graph setting the value is the subgraph of the DDG rooted at a system call instruction reachable by following reverse edges for the dependent data location. Second, the analyzer simulates the execution of the instructions in the expression graph to determine the value.

Multiple execution paths may set arguments differently, so the analyzer joins the recovered values via set union. The analyzer recovers sets of integers for numeric arguments and sets of regular expressions for string arguments. Consider the **write** system call at line 34 of Figure 2. The string value and length are passed as arguments from both call-sites to *log*. Value capture unions these values, producing the restricted system call transition

```
write(?, {"Relinked", "Execing"}, {8, 9}).
```

If analysis cannot reliably construct the expression graph or if a value is not statically known, the analyzer allows all possible values. The file-descriptor argument to **write** is dynamically generated and cannot be statically computed.

Value capture helps prevent attacks. For example, the filename argument passed to **exec** at line 25 of Figure 2 can be statically computed. By restricting the allowed argument, the model prevents the attacker from executing an arbitrary program without directly altering the filesystem.

4.1.2 Dependency Capture

The statically-constructed DDG may reveal data dependencies that cannot be evaluated until the program executes. The analyzer encodes these dependencies into the Dyck model so that the program monitor can compute the data values dynamically. For example, **write** often uses a file descriptor returned by a previous call to **open**. Although the returned file descriptor cannot be statically known, the dependency between the argument and return value can be encoded in the model. The program monitor can then restrict allowed arguments based on the actual return values observed during execution.

Similarly, a system call's return value may control program branching. The analyzer encodes a dependency between a return value and a branch with *predicate transitions*. The program monitor will only follow a transition whose predicate evaluates to true. The predicates restrict the automaton path explored to exactly the execution path taken by the program.

For example, the DDG reveals a dependence between the branch at line 37 and the **write** system call at line 34. Given the branch condition "branch if greater than 0", the analyzer adds the predicate transitions shown in Figure 4 to the model of the *log* function. If an attacker attempts to steer execution through the **write** call by specifying invalid arguments, as suggested by Wagner and Soto [162], then the call will return an error code less than 0. The future behavior accepted by the model is then restricted to a program exit.

4.2 Environment-Sensitive Analysis

A class of data dependencies requires cubic-time partial evaluation algorithms [25] to be recovered statically: dependencies to the external environment.

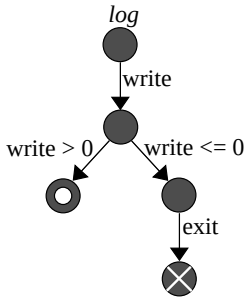

Figure 4: The model for *log* with predicate transitions encoding the branch based on the return value of **write**. The function can return allowing for continued execution only if **write** returns a positive value.

```
% ./a.out -F /home/me/tmp
System calls generated:
  unlink("/home/me/tmp")
  link("/tmp", "/home/me/tmp")
  write(5, "Linked", 8)

% ./a.out -F /progs/bin
System calls generated:
  unlink("/progs/bin")
  link("/bin", "/progs/bin")
  write(4, "Linked", 8)
```

Figure 5: Evidence that a command-line parameter restricts arguments to **unlink** and **link**.

DEFINITION 1. The *environment* is program input known at program load time and fixed for the entire execution of the program. This includes environment variables, command-line parameters, and configuration files. A *property* of the environment is a single variable, parameter, or configuration setting in a file. ■

An environment dependency captures the relation between environment properties and the program's execution behavior.

DEFINITION 2. Let E be the set of all environments containing property x. Let I be the set of all non-environment program inputs. Let $Value(p, d, e, i)$ denote the possibly-infinite set of values program point p may read from data location d given environment e and program input i. An *environment dependency* exists between x and p if

$$\exists f, d \ \left[\forall e \in E \ \forall i \in I \ \left[Value(p, d, e, i) = f(p, x)\right]\right]$$

In words: over all possible executions, a program data value at p depends only upon the value of x. ■

This initial definition of an environment dependency is overly restrictive because it requires environment properties to be independent. However, a data value may depend on a combination of multiple properties, such as a boolean combination of command-line flags. The restrictive definition will make the initial investigation easier as I can consider properties independently. As the work progresses, I can broaden the definition and recover additional dependencies.

An environment-sensitive program model restricts acceptable program behavior given the settings of environment properties. Let L_e denote the language containing all system call sequences that the program can generate in some actual environment e. A program model constructed from static analysis allows all possible environments E and recognizes the language $L_s = \bigcup_{e \in E} L_e$. A model that include environment dependencies recognizes a restricted language L_c, where $L_e \subseteq L_c \subseteq L_s$.

I propose a new analysis method, *differential dynamic analysis*, that identifies environment dependencies without the cost of partial evaluation. I will then illustrate how differential dynamic analysis may be used for system call argument recovery and branch analysis.

4.2.1 Differential Dynamic Analysis

Differential dynamic analysis learns environment dependencies by correlating changes in execution with changes to an environment property. The idea is not to learn behavior, but rather *changes* in behavior. It is

the experimental method: if varying one property produces a change in the process's execution, then this provides evidence of a dependency between the value of the property and the program behavior. The set of possible dependent behaviors contains all features visible in an execution trace. In Section 4.2.2, the dynamic analyzer observes system call arguments to learn environment dependencies. Section 4.2.3 shows how code instrumentation can add observable features to a trace and exposes program branching to the dynamic analyzer.

Like all learning-based techniques, differential dynamic analysis is not conservative. An incomplete training data set may fail to cover all execution paths that affect the dependency between an environment property and a program point. For example, a program may check a command-line string parameter against a special-case value. The program changes the parameter only when it matches the special-case. If I fail to test the special-case, then an environment dependency identified by differential analysis will be be correct only when the property does not take the special-case value. The execution monitor will then generate false alarms for any program execution using special-case values.

This raises research considerations:

- Differential analysis contributes only data-flow assertions to a conservative model constructed from static analysis. How do models with this limited use of learning compare to models constructed entirely from dynamic analysis? Intuitively, I expect that a model combining static and differential analysis will produce fewer false alarms than a dynamic model given the same training set size, or will require a smaller training set size to produce an equivalent false alarm rate. Can the false alarm rate be made sufficiently small, such as below 1 false alarm per 10^6 system calls?

- Adding dependencies learned from differential analysis for only properties that have no special case values prevents introduction of false alarms. Can static analysis verify that a dependency holds even for untested input? Is this verification cheaper than algorithms that build dependencies statically, such as partial evaluation?

I believe that static analysis can conservatively verify when a dynamically-recovered environment dependency holds for all input. Generally, environment values used in a program are stored in global memory, so verification would use an algorithm like the following:

1. Identify the data location d producing the dependency at program point p. The DDG used in Section 4.1 reveals this location.

2. Identify all program points $\mathcal{P}$ that may write to d. The static analyzer currently does not perform global memory analysis, although it is a simple extension of an existing algorithm that recovers possible indirect call targets during Dyck model construction.

3. Let $\mathcal{P}'$ be the union of $\mathcal{P}$ with all instructions in a backward slice of all $p \in \mathcal{P}$. Then $\mathcal{P}'$ is the set of instructions that affect the value of d. If the control dependence sets [60] of each $p' \in \mathcal{P}'$ are equal, then the dependency holds regardless of input. If any pair of sets are unequal, then the value d is conditionally set based upon some unknown input. The dynamically learned dependency violates soundness and cannot be safely added to the Dyck model in this case.

This algorithm requires control dependence information. Expanding the DDG to include control dependencies can be done with linear-time depth-first search. Computing slices in the DDG requires linear time, so verification is at worst a quadratic-time operation (due to the slice per element in $\mathcal{P}'$ in step 3). This is more efficient than cubic-time partial evaluation.

With either static verification or the simple assumption that no properties have special-case values, differential analysis works with as little as one piece of training data, although training sets covering more code paths allows the differential analysis to identify more environment dependencies. The recovered dependencies hold regardless of input data and require no retraining over time. Importantly, code paths not followed due to incomplete training sets simply maintain their conservative structure as constructed by static analysis and will not generate false alarms.

4.2.2 External Argument Recovery

System call arguments are observable program properties that may depend on the environment. External argument recovery uses differential analysis to identify dependencies between arguments and string values in the external environment. Evidence of a dependency exists when the string value of the property appears as a substring in the system call argument. If a variation in the property is echoed as a variation in the argument, then a dependency between the argument and the external data is clear (Figure 5).

Consider the `dest` argument passed to the **unlink** and **link** calls at lines 4 and 6, respectively, of Figure 2. This is a command-line parameter that cannot be recovered by static analysis. If the program contains a call to *mylink* followed by a call to *myexec*, an attacker can relink "/sbin/mailconf" to an arbitrary program and then execute the linked file. Static recovery of the string "/sbin/mailconf" (Section 4.1.1) does not prevent this attack. Differential analysis produces a model that detects the attack by limiting the `dest` argument to the filename specified on the command line. No other files can be relinked.

4.2.3 External Branch Analysis

Command-line flags and configuration file values usually invoke specific execution paths in a program. External branch analysis uses differential dynamic analysis to identify dependencies between these boolean flags and program branch behavior. Evidence of a dependency occurs when, regardless of execution context or other program input, a branch at program point p always branches in one direction when a boolean environment property takes one value, and always branches in the opposite direction when the property's value is reversed. A model incorporating external branch dependencies allows only the execution paths possible given the environment.

Dependency identification requires knowledge of the execution paths taken. Binary instrumentation offers a solution. Inserting code along both sides of a branch that logs the branch direction followed makes program branching observable. Optimizations may reduce the logging cost by removing instrumentation where the branch direction may be inferred [92].

The branch at line 19 of Figure 2 tests the boolean value `allow_exec` to set the branch destination. Static analysis cannot insert predicate transitions to restrict the model because the value is set by a command-line flag. Differential dynamic analysis can recover a dependency between the branch direction taken and the presence or absence of the flag. When monitoring the program, the model will not accept the **exec** system call in *myexec* if the `allow_exec` flag is not present in the environment. All attacks attempting to call **exec** will be detected as malicious.

Dynamic branch analysis opens interesting research questions. Boolean flags may not be independent: can differential dynamic analysis identify boolean combinations of flags that control branch behavior? Can the analyzer identify correlated branching?

4.2.4 The Combined Model

Environment dependencies identified by differential dynamic analysis limit the sequences accepted by the statically-constructed Dyck model. As with the statically-recovered data dependencies of Section 4.1, the actual data values flowing along environment dependencies can be recovered either by the static analyzer or at runtime by the program monitor:

- The static analyzer can compute values dependent upon the environment when the complete environment is known to the analyzer. The model produced is specific to this environment.
- The static analyzer can encode environment dependencies into the Dyck model. When the program monitor later loads the model to begin enforcement, it computes the dependent values given the current environment. This is a general model that adapts to the specific environment at every program load.

5 Security Evaluation

A model-based intrusion detection system identifies anomalies, or deviations from a model. The system is useful only when attacks appear anomalous to a program model. Formally, a sequence of system calls α is anomalous only when $\alpha \notin L(M)$, where $L(M)$ is the language of system call sequences accepted by model M. I propose that a formal model describing the operating system and system-call interface will enable evaluation of program models. I plan to investigate the following problems for any finite-state program model M:

1. Find an attack $\alpha \in L(M)$.
2. Prove that the absence of any attack $\alpha \in L(M)$ implies M detects all attacks.
3. Find all attacks $\alpha \in L(M)$.
4. *(optional)* Determine why M fails to detect an attack. In particular, does the model inaccurately represent the program or does the structure of the program code allow the attack? Like the refinement step of model checking and software verification [18, 19], refine an inaccurate model so that it detects the attack.

I believe that Problems 1 and 2 have feasible solutions, and the remainder of this section describes formal analysis methods to solve these problems. For Problem 3, I would like to produce a feasible solution or develop heuristics to find multiple attacks should no perfect solution exist. It is not readily apparent that a tractable solution exists to Problem 4 and, time permitting, the problem will receive additional investigation.

5.1 Mimicry Attacks

In a first step towards attack discovery, Wagner and Soto demonstrated that an undetected attack α' can be generated from a known, detected attack α by modifying α to appear as a sequence accepted by a program model [162]. These *mimicry attacks* mimic the expected program execution behavior defined by the model. The new sequence α' must still produce the goal, or bad operating system state, produced by the original attack. The authors defined an equivalence relation $\alpha \equiv_R \alpha'$ that holds whenever α can be "equivalently replaced" by α'. Any $\alpha' \in L(M)$ satisfying $\alpha \equiv_R \alpha'$ is a mimicry attack.

Although mimicry attacks demonstrate that a model must detect all possible variants of an attack, they are unsuitable for model evaluation.

- The phrase "equivalently replaced" is the foundation of the equivalence relation $\equiv_R$ but is not well defined. In practice, Wagner and Soto used their human knowledge of system calls to manually choose replacements. The new sequences were, in fact, not equivalent to the original attack because they produced different changes to the operating system.

- There is no rigorous method to find new sequences α'. The authors defined only the single *nop-insertion* obfuscation. A nop system call fails due to invalid arguments and hence does not change system state. Nop calls can be inserted between system calls in an existing attack without altering the attack's semantics. No other obfuscations were explored. In fact, without system call formalisms, obfuscation of existing attacks will be incomplete: limited only to a set of predefined obfuscations.

- Mimicry attack construction requires an existing attack sequence. Ideally, an evaluation tool can identify missed attacks not previously considered.

A formal approach should be able to automatically construct attacks without the above limitations. A formal model of the operating system enables construction of attacks undetected by a program model and does not require a pre-existing attack or database of obfuscations.

5.2 Formalizing the Operating System

Attacks against a process adversely affect the operating system on which the process runs. I plan to formally model the operating system as an infinite state transition system $\mathcal{G}$ that describes how system calls change the operating system's state.

DEFINITION 3. The operating system model is a five-tuple $\mathcal{G} = \langle V, \Sigma, P, S, \Delta \rangle$, where

- V is a finite set of state variables, where each variable may take a value from a possibly-infinite set. Each state variable describes a system attribute. For example, the currently executing process image is a variable taking a value from the set of all executable images on the system. The filesystem is a collection of objects, one per file. Each object is a set of state variables characterizing aspects of the file, such as its full name and mode value.

- Σ is an infinite set of operating system states. Each $\sigma \in \Sigma$ describes a possible state of the operating system, or equivalently, an assignment to each state variable in V.

- P is a set of predicates over V. Each predicate describes some assignment of values to state variables and returns the possibly infinite set of states in Σ where that assignment holds. For example, the predicate *executing_image(*"/bin/sh"*)* describes all states where the process executing is "/bin/sh". Predicates enable efficient description of the system call transition functions Δ, described below.

- $S \subseteq \Sigma$ is the possibly-infinite set of initial states. Each $s \in S$ describes a state that the operating system may be in when beginning a process's execution.

- Let A_i be the set of all possible system call arguments to system call i. Then $\Delta = \{\delta_i : i \text{ is a system call}\}$, where each $\delta_i : \Sigma \times A_i \rightarrow \Sigma$ describes how a system call transforms operating system state. ■

System calls determine the transition characterizing how operating system state changes. The predicates P make the problem of describing transformations of an infinite statespace tractable. Each system call is formalized as a set of preconditions and postconditions. If the model $\mathcal{G}$ is in a state s satisfying each predicate of a precondition, then the model can transition to all states satisfying the predicates in the postcondition.

exec (string *filename*, char **args*[]) returns int *R*		
Preconditions: *file_exists*(*filename*) *can_exec*(*filename*) **Postconditions:** *link_target*(*filename*, *dest*) *executing_image*(*dest*)	**Preconditions:** ¬*file_exists*(*filename*) **Postconditions:** *R* = -ENOENT	...

Figure 6: Partial specification for the **exec** system call. A complete specification would include all possible preconditions on system call arguments and their corresponding postconditions. ENOENT is the error code for a non-existent file.

Preconditions thus describe the required state before the system call, and postconditions describe the state produced by the call. For example, a postcondition could indicate that a file's size increased following a **write** call. In alternative terminology, each system call *requires* certain predicates to hold and *provides* new predicates after execution [152]. The set of preconditions for a system call must be disjoint because the modeled call executes deterministically given its arguments and operating system state.

Figure 6 shows a partial specification for the **exec** system call. A complete specification would include all possible preconditions on system call arguments and their corresponding postconditions.

A computation using the system model requires two additional elements specific to the computation.

- **Final states.** Successful attacks transform the operating system state into a state undesired by an administrator but advantageous to the attacker. The collection of these bad states defines the possible goals of an attacker and comprises the final states $F \subset \Sigma$ of the framework. Note that specific exploit methods or system call sequences that reach final states are irrelevant; the framework is not bound to known attacks.

- **System call sequence model.** A program model M, such as the Dyck model, defines allowed orderings of system calls in a call sequence. This model characterizes how system calls may be combined in the framework.

5.3 Model-Checking Program Models

Model-checking algorithms construct sequences of system calls that take the operating system from an initial state to a final state and respect the system call ordering specified by the program model. All discovered sequences are attacks missed by the model because they do not appear anomalous. The existence or non-existence of attack sequences allows evaluation of the program model's resistance to attack.

The evaluation tasks then become clear.

- **Model checking**: find an undetected attack α that transforms the operating system from a state $s_0 \in S$ to a state $s_n \in F$. This is similar to the model-checking problem of automated attack graph construction [140, 141]. An algorithm $\mathit{Check}_{\mathcal{G}}(M, F)$ returns the first sequence found in M that takes the operating system to an undesired state.

  The *Check* algorithm is complicated by the need to unify uninterpreted variables. Uninterpreted variables allow efficient description of system call transformers using variables that are not given values

until *Check* constructs an attack. For example, *filename*, *dest*, *args*, and *R* are uninterpreted variables in Figure 6. Unification binds uninterpreted variables to explicit values required by the pre- or post-conditions of other system calls. Existing model checking research has not addressed unification as the need does not arise in traditional applications of model checkers. Unification is implemented in algorithms used in the similar fields of deductive databases [94, 127, 128] and logic programming [146], so I believe that it can be feasibly added to existing model checking techniques as a new contribution to model checking.

Proving that failure to find any attack sequence missed by the model requires a proof that the *Check* algorithm is complete. This is a standard result of model checking, although I will need to prove that unification does not violate completeness. Naturally, *Check* can be complete only with respect to the abstractions of operating system state.

- **Exhaustive model checking**: find a model M' such that $L(M') \subseteq L(M)$, $L(M')$ contains only attack sequences, and $L(M) \setminus L(M')$ contains no attack sequences. Traditional model checkers are designed to identify only a single attack sequence rather than the entire set of attacks. Early efforts by Sheyner [140] indicate that model checkers can be extended to compute all missed attacks in his domain of attack graph construction. It is not immediately clear that Sheyner's work can translate to attacks against a program model due to the presence of unification and unique characteristics of the file system (e.g. file creation adds new objects to the file system). I plan to investigate this problem to contribute either a feasible solution or proof that no feasible solution exists in this problem domain.

- **Refinement** (optional): given an attack $\alpha \in L(M)$, compute a new model M' such that M' models the program, $L(M') \subset L(M)$, and $\alpha \notin L(M')$; or show that no such M' exists. At present, it is not clear that refinement is possible due to the need to verify or disprove that a new model M' is a valid program model. The problem unfortunately appears circular: verifying that a model M' conservatively represents the program requires checking against a model M'', with $L(M'') \subseteq L(M')$. However, if time permits, I will consider the refinement problem in an attempt to discover a different, workable solution.

The model-checking approach offers important benefits. It is fully automated and requires no human interaction. Obfuscated attacks, including attacks using *nop* insertion and semantically-equivalent calls, are automatically produced from the descriptions of system calls. The operating system and system call abstractions provide an analysis framework suitable to an arbitrary program model. Other work in model-checking shows that efficient implementations of the checking algorithm are possible [140, 141].

I expect to try both bottom-up and top-down evaluation algorithms [127, 128, 140, 141] with unification of uninterpreted variables. Bottom-up evaluation begins at all states in F and follows system call transitions in M in reverse when the system call's postconditions match the current operating system state. The configuration changes to match the preconditions required for the system call, and the algorithm continues. If a state in S is reached, then the path followed is an undetected attack. Top-down evaluation is similar, but progresses from S towards F.

The formal framework and model-checking approach show how various analyses improve program models. Consider the Dyck model in Figure 3d without data-flow analysis. Assume the program calls *mylink* followed by *myexec*. Define the final states F by the predicate *executing_image*("/bin/sh"). Unification will bind the **exec** argument to "/bin/sh" and **link** and **unlink** arguments to arbitrary files that do not change "/bin/sh". Complete evaluation will produce an attack sequence:

unlink(/home/giffin/prelim.ps)
link(/dev/zero, /home/giffin/main.c)
write(-1, 0, 0)
write(-1, 0, 0)
exec(/bin/sh)

When the Dyck model includes static data-flow analysis (Section 4.1), a more complex attack remains. The file "/sbin/mailconf" is relinked on the filesystem to point to "/bin/sh":

unlink(/sbin/mailconf)
link(/bin/sh, /sbin/mailconf)
write(-1, "Relinked", 9)
write(-1, "Relinked", 9)
exec(/sbin/mailconf)

An environment-sensitive model (Section 4.2) does not have an attack, provided "/sbin/mailconf" is not the file specified on the command-line.

6 Summary and Timeline

Fall 2004:

- Construct Dyck models for dynamically linked programs. Expand test suite to include network service programs and versions of programs with known vulnerabilities.
- Implement differential dynamic analysis and the two clients: dynamic branch analysis and argument analysis. Recover dependencies under the assumption that special-case environment property values do not exist. Measure the security provided by models incorporating differential analysis with the older average branching factor metric [160, 161].

Winter 2005:

- Expand the DDG to include control dependence edges.
- Implement static verification of learned environment dependencies using control dependence information.
- Investigate the relationship between the Dyck model, visibly push-down languages, and regular tree languages.

Spring 2005:

- Choose the model-checking or logic programming environment for implementation of the operating system model.
- Begin building the operating system model. Identify state variables and predicates.

Summer 2005:

- Write the system call transformers.

- If necessary, add unification to the model checker.

Fall 2005:

- Identify final states.
- Model-check the Dyck model.
- Acquire and test other model-based detection systems.

Winter 2006:

- Analyze the model-checking algorithm. Extend the algorithm to identify all missed attacks or show that this is infeasible.
- Caution: interview season.

Spring 2006:

- Writing.
- Caution: interview season.

Summer 2006:

- Writing.
- Graduation.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, Nov. 1996.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *ACM Symposium on Theory of Computing (STOC)*, pages 202–211, Chicago, Illinois, 2004.

[4] D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Safeguard final report: Detecting unusual program behavior using the NIDES statistical component. Technical report, Computer Science Laboratory, SRI International, Dec. 1993.

[5] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, Pennsylvania, Apr. 1980.

[6] Anonymous. Once upon a free()... *Phrack*, 57, Aug. 2001.

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

[8] S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *6th ACM Conference on Computer and Communications Security (CCS)*, pages 1–7, Kent Ridge Digital Labs, Singapore, Nov. 1999.

[9] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Department of Computer Engineering, Chalmers University, Mar. 2000.

[10] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, June 2000.

[11] S. M. Bellovin. There be dragons. In *3rd USENIX Security Symposium*, Baltimore, Maryland, Sept. 1992.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, pages 105–120, Washington, DC, Aug. 2003.

[13] Blexim. Basic integer overflows. *Phrack*, 60, Dec. 2002.

[14] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, May 2000.

[15] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[16] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, Apr. 2001.

[17] N. Chomsky and M. P. 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.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, Sept. 2003.

[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV)*, July 2000.

[20] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 1997. *http://www.grappa.univ-lille3.fr/tata*.

[21] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from `printf` format string vulnerabilities. In *10th USENIX Security Symposium*, Washington, DC, Aug. 2001.

[22] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, pages 91–104, Washington, DC, Aug. 2003.

[23] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, Texas, Jan. 1998.

[24] M. Crosbie and E. H. Spafford. Applying genetic programming to intrusion detection. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 1–8, Cambridge, Massachusetts, Nov. 1995. AAAI.

[25] M. Das. *Partial Evaluation Using Dependence Graphs*. Ph.D. dissertation, University of Wisconsin–Madison, Feb. 1998.

[26] D. Dasgupta and F. González. An immunity-based technique to characterize intrusions in computer networks. *IEEE Transactions on Evolutionary Computation*, 6(3):281–291, June 2002.

[27] H. Debar. *Application des Réseaux de Neurones à la Détection d'Intrusions sur les Systèmes Informatiques*. PhD thesis, de l'Université Paris, June 1993. (In French).

[28] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Research in Security and Privacy*, pages 240–250, Oakland, California, May 1992.

[29] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805–822, 1999.

[30] H. Debar, M. Dacier, A. Wespi, and S. Lampart. An experimentation workbench for intrusion detection systems. Technical report, IBM Research Division, Zurich Research Laboratory, Zurich, Switzerland, Mar. 1999.

[31] D. E. Denning. An intrusion-detection model. In *IEEE Symposium on Security and Privacy*, Oakland, California, Apr. 1986.

[32] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb. 1987.

[33] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Testing and evaluating computer intrusion detection systems. *Communications of the ACM*, 42(7):53–61, July 1999.

[34] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[35] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specivic, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2000.

[36] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Jan. 2004.

[37] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, Ontario, Canada, Sept. 1999.

[38] U. Erlingsson and F. B. Schneider. IRM enformcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2000.

[39] 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.

[40] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.

[41] H. H. Feng, O. M. 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.

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

[43] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10), Oct. 1997.

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

[45] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, pages 55–66, Washington, DC, Aug. 2001.

[46] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, May 1999.

[47] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *11th Network and Distributed Systems Security Symposium*, San Diego, California, Feb. 2004.

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

[49] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, 1998.

[50] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Santa Clara, California, Apr. 1999.

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

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

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

[54] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper application. In *USENIX Security Symposium*, 1996.

[55] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *7th System Administration Conference (LISA)*, Monterey, California, Nov. 1993.

[56] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, Sept. 1993.

[57] J. Hochberg, K. Jackson, C. Stallings, J. F. McClary, D. DuBois, and J. Ford. NADIR: An automated system for detecting network intrusion and misuse. *Computers & Security*, 12(3):235–248, 1993.

[58] S. A. Hofmeyer. *An Immunological Model of Distributed Detection and Its Application to Computer Security*. PhD thesis, University of New Mexico, May 1999.

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

[60] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 1992.

[61] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. Master's thesis, University of California, Santa Barbara, Nov. 1992.

[62] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *IEEE Symposium on Security and Privacy*, pages 16–28, Oakland, California, May 1993.

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

[64] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *7th Network and Distributed Systems Security Symposium*, San Diego, California, Feb. 2000.

[65] H. S. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *IEEE Symposium on Research in Security and Privacy*, 1991.

[66] H. S. Javitz and A. Valdes. The NIDES statistical component description and justification. Annual Report, SRI International, Mar. 1994.

[67] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Computer Security Foundations Workshop (CSFW)*, June 2001.

[68] M. M. Kaempf. Vudo–an object superstitiously believed to embody magical powers. *Phrack*, 57, Aug. 2001.

[69] R. A. Kemmerer and G. Vigna. Intrusion detection: A brief history and overview. *IEEE Computer*, pages 27–30, Apr. 2002.

[70] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, June 1999.

[71] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for java programs. In *First Workshop on Runtime Verification*, Paris, France, July 2001.

[72] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *2nd International Workshop on Run-Time Verification*, Copenhagen, Denmark, July 2002.

[73] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-Time Systems (ECRTS)*, pages 114–121, June 1999.

[74] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, San Francisco, California, Aug. 2002.

[75] 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, Dec. 1994.

[76] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1997.

[77] C. C. W. Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach*. PhD thesis, University of California, Davis, 1996.

[78] J. Korba. Windows NT attacks for the evaluation of intrusion detection systems. Master's thesis, June 2000.

[79] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, September/October 1997.

[80] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian event classification for intrusion detection. In *Annual Computer Security Applications Conference (ACSAC)*, Las Vegas, Nevada, Dec. 2003.

[81] 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, Jan. 1981.

[82] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, Aug. 1995.

[83] S. Kumar and E. H. Spafford. A pattern matching model for misuse intrusion detection. In *17th National Computer Security Conference*, pages 11–21, 1994.

[84] S. Kumar and E. H. Spafford. A software architecture to support misuse intrusion detection. Technical Report CSD-TR-95-009, COAST Project, Purdue University, Mar. 1995.

[85] T. Lane. Hidden markov models for human/computer interface modeling. In *IJCAI Workshop on Learning About Users*, pages 35–44, 1999.

[86] T. Lane and C. E. Brodley. Approaches to online learning and concept drift for user identification in computer security. In *Fourth International Conference on Knowledge Discovery and Data Mining*, pages 259–263, 1988.

[87] T. Lane and C. E. Brodley. An application of machine learning to anomaly detection. In *National Information Systems Security Conference*, Baltimore, Maryland, 1997.

[88] T. Lane and C. E. Brodley. Detecting the abnormal: Machine learning in computer security. Technical Report ECE-97-1, Department of Electrical and Computer Engineering, Purdue University, Jan. 1997.

[89] T. Lane and C. E. Brodley. Sequence matching and learning in anomaly detection for computer security. In *AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 43–49, 1997.

[90] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *5th ACM Conference on Computer and Communications Security (CCS)*, pages 150–158, 1998.

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

[92] J. R. Larus. Whole program paths. In *Programming Language Design and Implementation (PLDI)*, pages 259–269, 1999.

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

[94] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1988.

[95] W. Lee. *A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems*. PhD thesis, Columbia University, 1999.

[96] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *7th USENIX Security Symposium*, 1998.

[97] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI97 Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[98] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang. Real time data mining-based intrusion detection. In *DARPA Information Survivability Conference & Exposition II (DISCEX)*, pages 89–100, Anaheim, California, June 2001.

[99] W. Lee, S. J. Stolfo, and K. W. Mok. Mining audit data to build intrusion detection models. In *Knowledge Discovery and Data Mining*, pages 66–72, 1998.

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

[101] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.

[102] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[103] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

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

[105] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks: International Journal of Computer and Telecommunications Networking*, 34(4):579–595, Oct. 2000.

[106] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *2000 DARPA Information Survivability Conference and Exposition*, volume 2, 2000.

[107] E. Lundon and E. Jonsson. Some practical and fundamental problems with anomaly detection. In *4th Nordic Workshop on Secure IT Systems*, Kista, Sweden, Nov. 1999.

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

[109] T. F. Lunt. Detecting intruders in computer systems. In *1993 Conference on Auditing and Computer Technology*, 1993.

[110] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *IEEE Symposium on Security and Privacy*, pages 59–66, Oakland, California, Apr. 1988.

[111] T. F. Lunt, R. Jagannathan, R. Lee, and A. Whitehurst. Knowledge-based intrusion detection. In *Annual AI Systems in Government Conference*, pages 102–107, Washington, DC, Mar. 1989.

[112] J. A. Marin, D. Ragsdale, and J. Surdu. A hybrid approach to profile creation and intrusion detection. In *DARPA Information Survivability Conference and Exposition II*, pages 69–76, Anaheim, California, June 2001.

[113] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman. An overview of issues in testing intrusion detection systems. Technical Report NISTIR 7007, National Institute of Standards and Technology, July 2003.

[114] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Annual Computer Security Application Conference (ACSAC)*, Dec. 1997.

[115] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 58, Dec. 2001.

[116] P. G. Neumann and P. A. Porras. Experience with EMERALD to date. In *1st USENIX Workshop on Intursion Detection and Network Monitoring*, Santa Clara, California, Apr. 1999.

[117] T. Newsham. *Format String Attacks*. Guardent, Incorporated, Sept. 2000.

[118] M. Oka, H. Abe, Y. Oyama, and K. Kato. Intrusion detection system based on binary code and execution stack analysis. In *Japan Society for Software Science and Technology Annual Conference (JSSSP)*, Aichi, Japan, Sept. 2003.

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

[120] C. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, Charlottsville, Virginia, Sept. 1998.

[121] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *20th National Information Systems Security Conference (NISSC)*, Baltimore, Maryland, Oct. 1997.

[122] M. Prasad and T.-C. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX 2003 Annual Technical Conference*, pages 211–224, San Antonio, Texas, June 2003.

[123] K. E. Price. Host-based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, Dec. 1997.

[124] N. Puketza, M. Chung, R. A. Olsson, and B. Mukherjee. A software platform for testing intrusion detection systems. *IEEE Software*, 14(5):43–51, September/October 1997.

[125] N. J. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, Oct. 1996.

[126] C. R. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, 1998.

[127] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL–control, relations and logic. In *19th Very Large Data Bases (VLDB)*, Vancouver, British Columbia, Canada, 1992.

[128] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB Journal*, 2:161–210, 1994.

[129] Rix. Smashing C++ vptrs. *Phrack*, 56, May 2000.

[130] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *17th Systems Administration Conference (LISA)*, San Diego, California, Oct. 2003.

[131] S. Rubin, S. Jha, and B. P. Miller. Attack generation for NIDS testing using natural deduction. Technical Report 1496, University of Wisconsin, Madison, Jan. 2004.

[132] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[133] 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.

[134] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *USENIX Intrusion Detection Workshop*, 1999.

[135] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *21st National Computer Security Conference*, Oct. 1998.

[136] R. Sekar, A. Gupta, J. Frullo, T. Shanghag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: A new approach for detecting network intrusions. In *ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[137] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[138] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. 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, Oct. 2003.

[139] J. Shavlik, M. Shavlik, and M. Fahland. Evaluating software sensors for actively profiling windows 2000 users. In *Fourth International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2001.

[140] O. Sheyner. *Scenario Graphs and Attack Graphs*. Ph.D. dissertation, Carnegie Mellon University. In preparation.

[141] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[142] S. E. Smaha. Haystack: An intrusion detection system. In *4th Aerospace Computer Security Applications Conference (ACSAC)*, pages 37–44, Dec. 1988.

[143] Solar Designer. Non-executable stack patch. *http://www.openwall.com/linux*.

[144] S. Soman, C. Krintz, and G. Vigna. Detecting malicious java code using virtual machine auditing. In *12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[145] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. In *New Security Paradigms Workshop*, pages 75–82, Langdale, Cumbria, United Kingdom, 1997.

[146] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 2nd edition, Mar. 1994.

[147] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, Sept. 1989.

[148] K. Tan. The application of neural networks to UNIX computer security. In *International Conference on Neural Networks (ICNN)*, 1995.

[149] K. Tan, K. S. Killourhy, and R. A. 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, Oct. 2002. Springer-Verlag.

[150] K. Tan and R. A. 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.

[151] 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, Oct. 2002. Springer-Verlag.

[152] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *New Security Paradigms Workshop*, pages 31–38, Cork, Ireland, 2000.

[153] H. S. Teng, K. Chen, and S. C.-Y. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *IEEE Symposium on Research in Security and Privacy*, pages 278–284, 1990.

[154] C.-Y. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt. A specification-based intrusion detection system for AODV. In *1st ACM Workshop on Security of Ad Hoc and Sensor Networks*, pages 125–134, Fairfax, Virginia, 2003.

[155] A. Tsymbal. The problem of concept drift: Definitions and related work. Technical Report TCD-CS-2004-15, Computer Science Department, Trinity College Dublin, Apr. 2004.

[156] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *IEEE Symposium on Security and Privacy*, pages 280–289, Oakland, California, May 1989.

[157] G. Vigna, B. Cassell, and D. Fayram. An intrusion detection system for aglets. In N. Suri, editor, *International Conference on Mobile Agents*, LNCS, Barcelona, Spain, Oct. 2002.

[158] G. Vigna, F. Valeur, and R. A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering Joint Meeting (ESEC/FSE)*, Helsinki, Finland, Sept. 2003.

[159] W. von Dyck. Gruppentheoretische studien. *Mathematische Annalen*, 20:1–44, 1882. (In German).

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

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

[162] 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, Nov. 2002.

[163] 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.

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

[165] R. Wojtczuk. Defeating solar designer non-executable stack patch. *http://www.securityfocus.com/archive/1/8470/2004-05-05/2004-05-11/2*, Jan. 1998.

[166] N. Ye. A markov chain model of temporal behavior for anomaly detection. In *IEEE Workshop on Information Assurance and Security*, West Point, New York, June 2000.

[167] N. Ye and X. Li. A scalable clustering technique for intrusion signature recognition. In *IEEE Workshop on Information Assurance and Security*, West Point, New York, June 2001.

A Dyck Model Definitions

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

DEFINITION 4. *Let* $G = \langle V, E \rangle$, $E \subseteq V \times V$ be the control flow graph of program Pr. Let $a \triangleleft v$ indicate that vertex $v \in V$ contains call site a. The *local model* for each function $i \in Pr$ is $A_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, F_i)$, where:

- $Q_i = V$
- $\Sigma_i = S_i \cup C_i \cup \{\epsilon\}$ where $S_i \subseteq \mathcal{S}$ and $C_i \subseteq \mathcal{C}$
- $q_{0,i} \in V$ is the unique CFG entry state
- $F_i = \{v \in V \setminus q_{0,i} \,|\, v \text{ is a CFG exit } \}$
- $q \in \delta_i(p, a)$ if $a \triangleleft p$ and $(p, q) \in E$
- $q \in \delta_i(p, \epsilon)$ if $\forall a \in S_i \cup C_i : a \not\triangleleft p$ and $(p, q) \in E$ ■

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

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

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

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

DEFINITION 5. *Let* i range over all functions in Pr with τ the entry point function. Let f_γ denote an automaton alphabet symbol that pushes γ onto the Dyck stack. Let g_γ denote an alphabet symbol that pops γ from the stack. Let $\mathcal{F} = \{f_\gamma : \gamma \in \mathcal{C}_{12}\}$ and $\mathcal{G} = \{g_\gamma : \gamma \in \mathcal{C}_{12}\}$. Then D is a *Dyck model* if there exists $D_\epsilon = (Q, \Sigma \cup \{\epsilon\}, \Gamma, \delta_\epsilon, q_0, z_0, F)$ with:

1. $Q = \bigcup_i Q_i$
2. $\Gamma = \mathcal{C}_{12}$
3. $\Sigma = \mathcal{S} \cup \mathcal{F} \cup \mathcal{G}$
4. $q_0 = q_{0,\tau}$
5. $z_0 = \epsilon$
6. $F = F_\tau$

7. $(q, z) \in \delta_\epsilon(p, a, z)$ if $a \in \mathcal{S}$ and $\exists i : q \in \delta_i(p, a)$
8. $(q, z) \in \delta_\epsilon(p, \epsilon, z)$ if
   (a) $\exists a \in \mathcal{C}_2 \cup \mathcal{C}_3\, \exists i : q \in \delta_i(p, a)$; or
   (b) $\exists a \in \mathcal{C}_4\, \exists i\, \exists r \in Q_i : r \in \delta_i(p, a) \wedge q = q_{0,\theta(a)}$; or
   (c) $\exists a \in \mathcal{C}_4\, \exists i\, \exists r \in Q_i : q \in \delta_i(r, a) \wedge p \in F_{\theta(a)}$
9. $(q, za) \in \delta_\epsilon(p, f_a, z)$ if $a \in \mathcal{C}_{12} \wedge \exists i\, \exists r \in Q_i : r \in \delta_i(p, a) \wedge q = q_{0,\theta(a)}$
10. $(q, \epsilon) \in \delta_\epsilon(p, g_a, a)$ if $a \in \mathcal{C}_{12} \wedge \exists i\, \exists r \in Q_i : q \in \delta_i(r, a) \wedge p \in F_{\theta(a)}$

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

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

B Data Dependence Graph

The Data Dependence Graph (DDG) is a common program analysis structure representing interprocedural flows of data through a program [81, 119]. It is a subgraph of the program dependence graph [42] 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 6. *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 data location L. ■

Figure 7 shows the DDG constructed for the program code in Figure 2. Shaded node numbers correspond to line numbers in Figure 2. SPARC delay slots are unwound, so node 26 precedes node 25 in the graph.

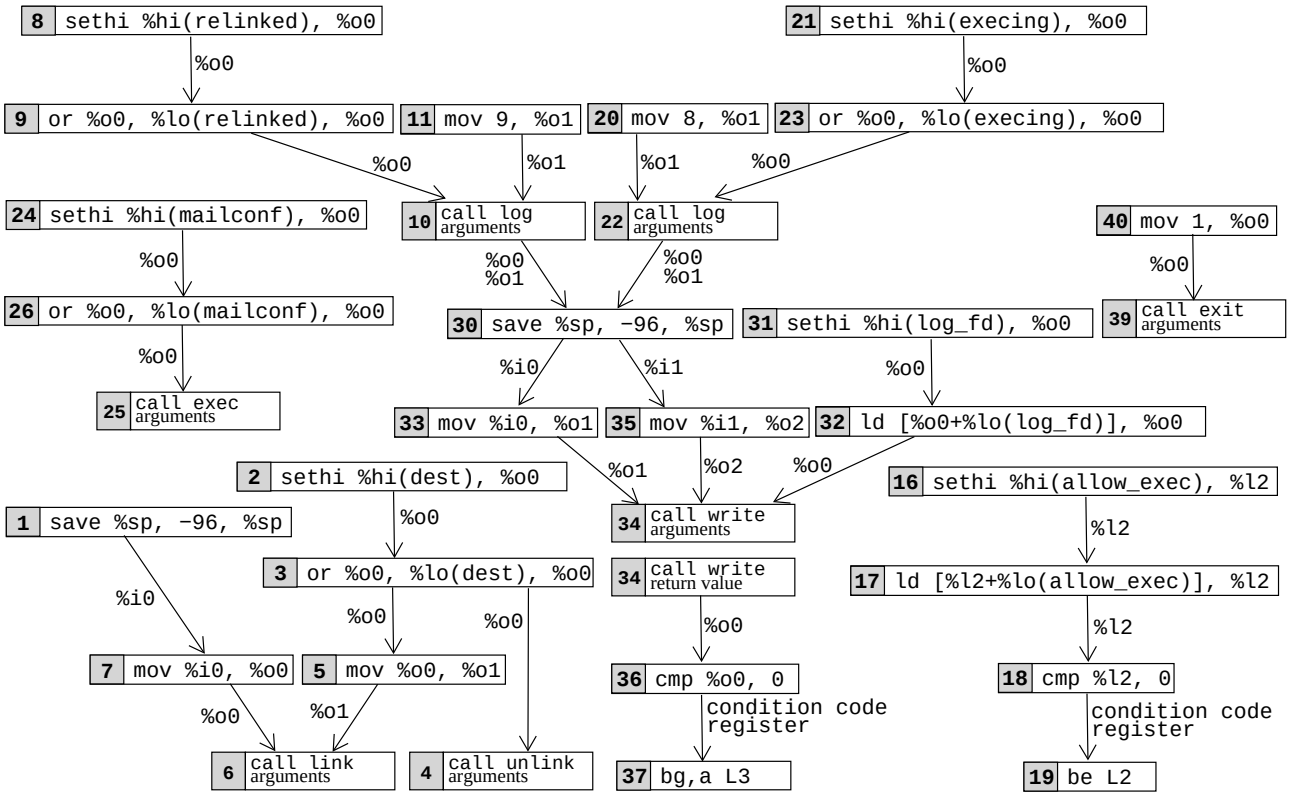

Figure 7: Data dependence graph for the code in Figure 2. The shaded numbers correspond to the line numbers in Figure 2. In SPARC code, the rightmost register in an instruction is the written register. Registers `%o0-%o5` contain call arguments. `%o0` contains the return value of a call. Call instructions are split into two nodes representing data flows through the arguments to the call and flows through the return value; this figure shows only the call nodes affecting data flow.