Environment-Sensitive Intrusion Detection

Jonathon T. Giffin[1], David Dagon[2], Somesh Jha[1], Wenke Lee[2], and Barton P. Miller[1]

[1] Computer Sciences Department, University of Wisconsin
{giffin,jha,bart}@cs.wisc.edu
[2] College of Computing, Georgia Institute of Technology
{dagon,wenke}@cc.gatech.edu

Abstract. We perform host-based intrusion detection by constructing a model from a program's binary code and then restricting the program's execution by the model. We improve the effectiveness of such model-based intrusion detection systems by incorporating into the model knowledge of the environment in which the program runs, and by increasing the accuracy of our models with a new data-flow analysis algorithm for context-sensitive recovery of static data.

The environment—configuration files, command-line parameters, and environment variables—constrains acceptable process execution. Environment dependencies added to a program model update the model to the current environment at every program execution.

Our new static data-flow analysis associates a program's data flows with specific calling contexts that use the data. We use this analysis to differentiate system-call arguments flowing from distinct call sites in the program.

Using a new average reachability measure suitable for evaluation of call-stack-based program models, we demonstrate that our techniques improve the precision of several test programs' models from 76% to 100%.

Key words: model-based anomaly detection, Dyck model, static binary analysis, static data-flow analysis.

1 Introduction

A host-based intrusion detection system (HIDS) monitors a process' execution to identify potentially malicious behavior. In a model-based anomaly HIDS or behavior-based HIDS [3], deviations from a precomputed model of expected behavior indicate possible intrusion attempts. An execution monitor verifies a stream of events, often system calls, generated by the executing process. The monitor rejects event streams deviating from the model. The ability of the system to detect attacks with few or zero false alarms relies entirely upon the precision of the model.

Static analysis builds an execution model by analyzing the source or binary code of the program [5, 10, 14, 20]. Traditionally, static analysis algorithms are conservative and produce models that overapproximate correct execution. In particular, previous statically constructed models allowed execution behaviors possible in any execution environment. Processes often read the environment—configuration files, command-line parameters, and environment variables known at process load time and fixed for the entire execution of the process. The environment can significantly constrain a process' execution, disabling entire blocks of functionality and restricting the process' access.

If the process can generate the language of event sequences L_e given the current environment e, then previous program models constructed from static analysis accepted the language $L_s = \cup_{i \in E} L_i$ for E the set of all possible environments. L_s is a superset of L_e and may contain system call sequences that cannot be generated by correct execution in environment e.

These overly general models may fail to detect attacks. For example, versions of the OpenSSH secure-shell server prior to 3.0.2 had a design error that allowed users to alter the execution of the root-level login process [19]. If the configuration file setting "uselogin" was disabled, then the ssh server disabled the vulnerable code. However, an attacker who has subverted the process can bypass the "uselogin" checks by directly executing the vulnerable code. Previous statically constructed models allowed all paths in the program, including the disabled path. By executing the disabled code, the attacker can undetectably execute root-level commands.

In this paper, we make statically constructed program models sensitive to the execution environment. An *environment-sensitive* program model restricts process execution behavior to only the behavior correct in the current environment. The model accepts a limited language of event sequences L_v, where $L_e \subseteq L_v \subseteq L_s$. Event sequences that could not be correctly generated in the current environment are detected as intrusive, even if those sequences are correct in some other environment. In the OpenSSH example, if "uselogin" was disabled, then the model disallows system calls and system-call arguments reachable only via the vulnerable code paths. The model detects an entire class of evasion attacks that manipulate environment data, as described in Sect. 7.4.

Environment dependencies characterize how execution behavior depends upon environment values. Similar to def-use relations in static data-flow analysis [15], an environment dependency relates values in the environment, such as "uselogin", to values of internal program variables. When an environment-sensitive HIDS loads a program model for execution enforcement, it customizes the model to the current environment based upon these dependencies. In this paper, we manually identify dependencies. Our long-term goal is to automate this procedure, and in Sect. 5.3 we postulate that automated identification will not be an onerous task.

Environment sensitivity works best with system-call argument analysis. Our static analyzer includes powerful data-flow analysis to recover statically known system-call arguments. Different execution paths in a program may set a system-call argument differently. Our previous data-flow analysis recovered argument values without calling context, in that the analysis algorithm ignored the association between an argument value and the call site that set that value [9, 10]. In this work, we encode calling context with argument values to better model the correct execution behavior of a program. A system-call argument value observed at runtime must match the calling context leading up to the system call. Additionally, the data-flow analysis now crosses shared object boundaries, enabling static analysis of dynamically-linked executables.

Although environment-sensitive program modeling is the primary focus of our work, we make an additional contribution: a new evaluation metric. The existing standard metric measuring model precision, average branching factor, poorly evaluates models that monitor a program's call stack in addition to the system-call stream [5, 8]. We instead use context-free language reachability to move forward through stack events to discover the next set of actual system calls reachable from the current program location. Our new *average reachability measure* fairly evaluates the precision of program models that include function call and return events. Using the average reachability measure, we demonstrate the value of whole-program data-flow analysis and environment-sensitive models. On four test programs, we improved the precision of context-sensitive models from 76% to 100%.

In summary, we believe that this paper makes the following contributions:

- Static model construction of dynamically-linked executables. In particular, the static analyzer continues data-flow analysis across shared-object boundaries by learning the API by which programs call library code, as described in Sect. 4.1.
- Context-sensitive encoding of recovered system-call arguments, detailed in Sect. 4.2. Combined with whole-program analysis, this technique improved argument recovery by 61% to 100% in our experiments.
- A formal definition of environment-sensitive program models and methods to encode environment dependencies into statically constructed program models. Environment sensitivity and static system-call argument recovery improved the precision of program models by 76% to 100%. Section 5 presents this work.
- An extension to the commonly-used average branching factor metric suitable for program models that require update events for function calls and returns (Sect. 6). The average reachability measure provides a fairer comparison of call-stack-based models and other models that do not monitor the call stack.

2 Related Work

In 1994, Fix and Schneider added execution environment information to a programming logic to make program specifications more precise [7]. The logic better specified how a program would execute, allowing for more precise analysis of the program in a proof system. Their notion of environment was general, including properties such as scheduler behavior. We are proposing a similar idea: use environment information to more precisely characterize expected program behavior in a program model. As our models describe safety properties that must not be violated, we focus on environment aspects that can constrain the safety properties.

Chinchani *et al*. instrumented C source-code with security checks based upon environment information [1]. Their definition of environment primarily encompassed low-level properties of the physical machine on which a process executes. For example, knowing the number of bits per integer allowed the authors to insert code into a program to prevent integer overflows. This approach is specific to known exploit vectors and requires source-code editing, making it poorly suited for our environment-sensitive intrusion detection.

One aspect of our current work uses environment dependencies and static analysis to limit allowed values to system-call arguments. This specific problem has received prior attention.

Static analysis can identify constant, statically known arguments. While extracting execution models from C source code, Wagner and Dean identified arguments known

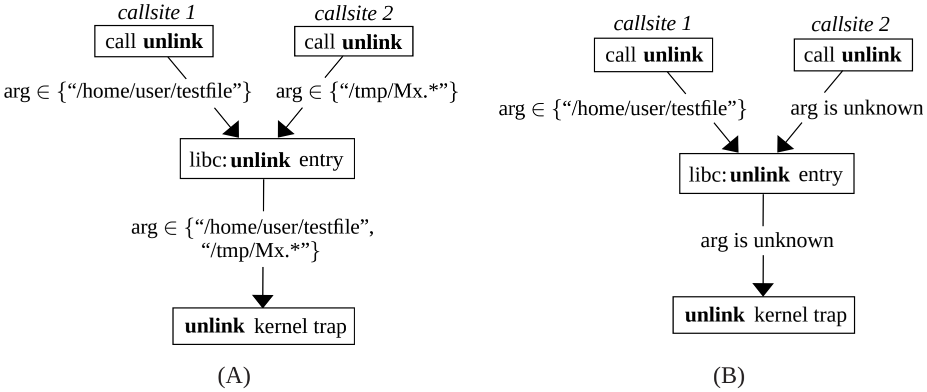

Fig. 1. Prior static argument recovery. Argument values recovered along different execution paths join together when the execution paths converge. (A) The association between a specific argument value and an execution path is lost. (B) If an argument value cannot be statically recovered on any execution path leading to a system call, all other recovered values must be discarded. The argument is completely unconstrained

statically [20]. In earlier work, we used binary code analysis to recover arguments in SPARC executables [9, 10]. These efforts suffered from several problems:

- Earlier binary data-flow analysis required statically-linked executables. In this paper, we use data-flow analysis to learn the API for a shared object. When analyzing an executable, we continue data-flow analysis anywhere the library API is used.
- Values recovered were not sensitive to calling context. This forces two inaccuracies. First, the association between a system-call argument value and the execution path using that value is lost (Fig. 1A). An attacker could undetectably use a value recovered on one execution path on any other execution path to the same system call. Second, if any execution path set an argument in a way not recoverable statically, all values recovered along all other execution paths must be discarded for the analysis to be safe (Fig. 1B). Our current work avoids these two inaccuracies by encoding calling context with recovered values.
- Static analysis cannot recover values set dynamically. In this paper, we make a distinction between dynamic values set at load time and values set by arbitrary user input. Environment dependencies augment static analysis and describe how values set when the operating system loads a process flow to system-call arguments.

Dynamic analysis learns a program model by generalizing behavior observed during a training phase. Kruegel *et al.* [13] and Sekar *et al.* [16] used dynamic analysis to learn constraints for system-call arguments. These constraints will include values from the environment that are used as part of a system-call argument, which forces a tradeoff. The training phase could modify environment values to learn a general model, but such a model fails to constrain later execution to the specific environment. Conversely, training could use only the current environment. If the environment ever changes, however, then the model no longer characterizes correct execution and retraining becomes necessary. By including environment dependencies described in this paper, learning could be done

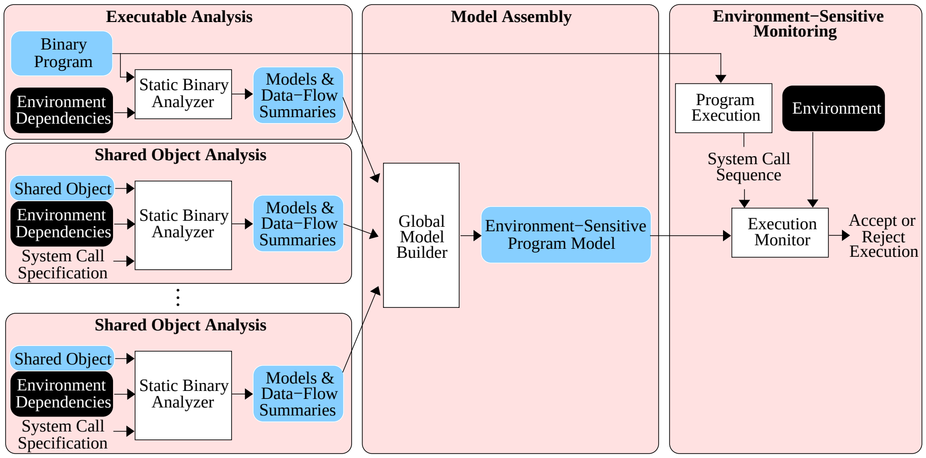

Fig. 2. Architecture

only for arguments not dependent upon the environment. Environment dependencies would resolve the remaining arguments to the current environment every time the model was subsequently loaded.

Environment-sensitive models are well suited to the model-carrying code execution design. Sekar *et al.* proposed that unknown, untrusted executables can include models of their execution [16]. A consumer of the executable can use a model checker to verify that the model does not violate their security policy and an execution monitor to limit the program's execution to that allowed by the model. The code producer must build the program model, but they cannot know any consumer's specific execution environment. To avoid false alarms, the model must be general to suit all possible environments. Such a general model may not satisfy a consumer's security policy. If the code producer adds environment dependencies to the model shipped with the code, the model will automatically adapt to every consumer's unique environment. With the environment constraints, the model is increasingly likely to satisfy a consumer's security policy.

3 Overview

Model-based anomaly detection has two phases: construction of the program model and execution enforcement using the model. Environment sensitivity affects both phases. Figure 2 shows the overall architecture of our system, including how environment information is used in each phase. Analysis, at the left, occurs once per program or shared object. The global model builder assembles all execution models into the single, whole-program model. The panel on the right, execution monitoring, occurs every time the program is loaded for execution.

The static analyzer builds a model of expected execution by reconstructing and analyzing control flows in a binary executable. The control flow model that we construct is the Dyck model, a context-sensitive model that uses a finite-state machine to enforce ordering upon system-call events as well as correct function call and return behavior [10]. The static analyzer encodes environment dependencies into the Dyck model.

```
 1 void parse_args(int argc, char **argv) {
 2   char *tn = tempnam(getenv("TMP"), "Mx");
 3   int execmode = 1;
 4   char c;
 5
 6   unlink("/home/user/tmpfile");
 7   while ((c = getopt(argc, argv, "L:")) != -1)
 8     switch (c) {
 9       case ’L’:
10         execmode = 0;
11         unlink(tn);
12         link(optarg, tn);
13         break;
14     }
15
16   if (execmode)
17     exec("/sbin/mail");
18 }
```

Fig. 3. Example code, with calls to C library system-call wrapper functions in boldface. Although we analyze SPARC binary code, we show C source code for readability. For conciseness, we omit error-handling code commonly required when calling C library functions

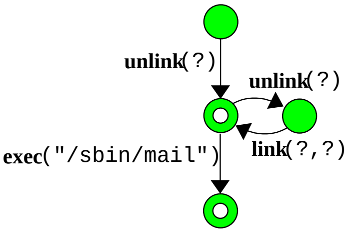

Fig. 4. A finite-state machine model of the code. System calls include argument restrictions identified by static data-flow analysis

Environment dependencies describe the relationship between a value in the execution environment and a variable in the program, as detailed further in Sect. 5.

A separate process, the runtime monitor, only allows process execution that matches the program model. The monitor resolves environment dependencies in the Dyck model given the actual environment in which the process is about to execute. By parsing the program’s command line, its configuration files, and the system’s environment variables, the monitor knows the execution environment when the operating system loads the program. It prunes portions of the model corresponding to code unreachable in the current environment by determining the directions that branches dependent upon the environment will take. It similarly propagates environment values along dependencies to update system-call argument constraints before the monitored process begins execution. The model used for execution verification thus enforces restrictions arising from environment dependencies.

Consider the example function in Fig. 3. Although the figure shows C source code for readability, we analyze SPARC binary code in our experiments. This code uses environment information in ways similar to many real programs. The `getenv` call in line 2 returns the value of the environment variable `TMP`, which typically specifies the system’s directory for temporary files. The returned directory name is used by the `tempnam` call to construct a filename in the temporary directory. The filename is used by the **link** and **unlink** system calls in lines 11 and 12. The `getopt` function call in line 7 parses options passed to the program via the command line and sets the value of the C library global variable `optarg`. The option “–L” requires one argument, `optarg`, that is passed as an argument to **link** at line 12. If the command line contains the “–L” option, the case statement at line 9 will execute and the **exec** at line 17 will not execute. If “–L” is not present, then the opposite holds: the **exec** will execute but the code inside the case statement will be skipped.

Figure 4 shows the finite-state machine model constructed for *parse_args* using earlier static analysis methods [9, 10]. This model overapproximates the correct execution of the function:

- The argument to both **unlink** calls is unconstrained, so an attacker could undetectably delete any file in a directory to which the process holds write access. The arguments are not statically recovered because the **unlink** at line 11 depends upon a dynamic value, the environment variable `TMP`. Both **unlink** calls target the same C library system-call wrapper function. Data-flow analysis of the system-call argument will join the values propagating from both call sites, as in Fig. 1B. Joining the statically recovered value from line 6 with the unknown value from line 11 forces the analyzer to discard the known value.
- Both arguments to **link** are unconstrained because they are computed dynamically from the execution environment.
- The two system calls inside the case statement and the **exec** system call are always accepted. In particular, all three calls would be accepted together. The branch correlation that forces *either* the case statement or the **exec** to execute has been lost.

At first glance, the **exec** call appears safe because static analysis can constrain its argument value. However, due to the overapproximations in the model described above, the model accepts a sequence of system calls that will execute a shell process. The attack first issues a nop call [21] and then relinks the statically recovered filename to a shell before the **exec** call occurs:

```
unlink(NULL); // Nop call
unlink("/sbin/mail");
link("/bin/sh", "/sbin/mail");
exec("/sbin/mail");
```

Note that the attack requires the initial nop call because the **link** transition in the model is preceded by two **unlink** transitions.

Environment sensitivity and the static argument analysis presented in this paper repair these imprecisions and produce a program model that better represents correct execution. Context-sensitive encoding of system-call arguments will differentiate the values passed from the two unique call sites to the **unlink** system-call wrapper, enabling recovery of the static argument at the line 6 call site even without recovering the argument at line 11. Adding environment dependencies then produces the environment-sensitive model shown in Fig. 5. The model is a template, containing dependencies that must be resolved by the execution monitor.

The monitor instantiates the template model in the current environment. Suppose the environment variable `TMP` is set to `/tmp`. For a command line without “–L”, the unreachable case statement code is removed (Fig. 6A). For the command line “`-L /home/user/log`”, the monitor will prune the unreachable **exec** call and constrain possible values to the remaining system-call arguments (Fig. 6B). The model better reflects correct execution in the specific environment. In both cases, the model prevents the relinking attack previously described.

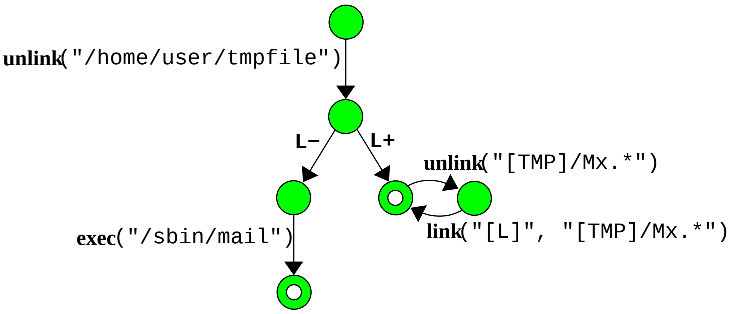

Fig. 5. The environment-sensitive model produced by the static analyzer. The model is a template, containing environment dependencies that are resolved when the model is loaded. The symbols `L-` and `L+` are branch predicates that allow subsequent system calls when the command-line parameter "–L" is omitted or present, respectively. The value `[L]` is the parameter value following "–L" on the command line. The value `[TMP]` is the value of the `TMP` environment variable

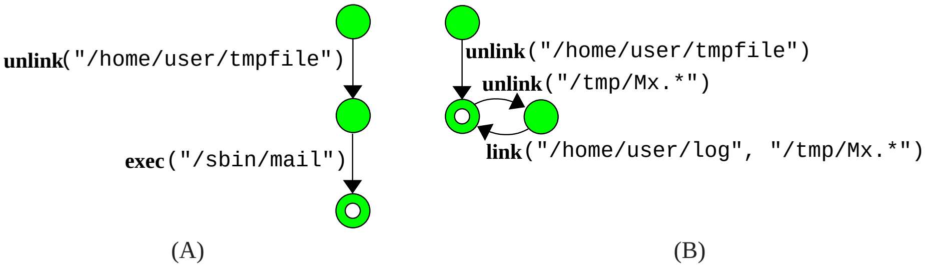
unlink("/home/user/tmpfile")
exec("/sbin/mail")
unlink("/home/user/tmpfile")
unlink("/tmp/Mx.*")
link("/home/user/log", "/tmp/Mx.*")
(A)
(B)

Fig. 6. The environment-sensitive model, after the execution monitor has resolved environment dependencies. System-call arguments are encoded with calling context, so different calls to **unlink** enforce different arguments. String arguments are regular expressions. (A) When the command line does not contain "–L", the code processing the option is pruned from the model. (B) When "–L" is present, the **exec** call is unreachable and pruned

4 System-Call Argument Analysis

Our analyzer attempts to recover system-call arguments that are statically known. It analyzes data flows within program code and into shared object code to determine how arguments may be constrained. The execution monitor enforces restrictions on any recovered system-call arguments and rejects any system call that attempts to use incorrect argument values.

4.1 Learning a Library API

The object code of a program is linked at two distinct times. *Static linking* occurs as part of a compilation process and combines object code to form a single program or shared object file. *Runtime linking* happens every time a program is loaded for execution and links code in separate shared objects with the main executable. Static analyzers inspect object code after static linking but before the final runtime link. Our analyzer simulates the effects of the runtime link to build models for programs whose code is distributed among shared object files. This model construction has two primary steps.

First, we analyze all shared objects used by a program. We build models for the program code in each shared object and cache the models on disk for future reuse. Our program models include virtual memory addresses of kernel traps and function call sites; however, the addresses used by shared object code are not known until runtime linking occurs. The analyzer performs *symbolic relocation* for shared object code. Each shared object is given its own virtual address space indexed at 0 that is strictly symbolic, and all addresses used in models reside in the symbolic address space. When later enforcing a program model, our execution monitor detects the actual address at which the runtime linker maps shared object code and resolves all symbolic addresses to their actual virtual addresses.

Second, we analyze the binary executable of interest. The executable may call functions that exist in shared object code. Our analyzer simulates the runtime linker's *symbol resolution* to identify the code body targeted by the dynamic function call. It reads the cached model of the shared object's code from disk and incorporates it into the program's execution model.

The separate code analysis performed for each shared object and for the main executable complicates data-flow analysis for system-call argument recovery. System calls generally appear only within C library functions. Frequently, however, the argument values used at those system calls are set by the main executable and passed to the C library through some function in the library's API. Separate analysis of the library code and the main executable code precludes our previous static data-flow analysis from recovering these arguments. The data flow is broken at the library interface.

To remedy this problem, we now perform *whole-program data-flow analysis* to track data flowing between separate statically linked object files. The analyzer first learns the API of a shared object. It initiates data-flow analysis at system-call sites with type information for the call's arguments (e.g. integer argument or string argument). Data-flow analysis follows program control flows in reverse to find the instructions that affect argument values. If any value depends upon a formal argument of a globally visible function, then that function is a part of the API that affects system-call arguments. We cache a *data-flow summary function* [17] that characterizes how data flows from the API function's entry point to the system-call site in the shared object. For example, one summary function for the C library stipulates that the first argument of the function call `unlink` flows through to the first argument of the subsequent **unlink** system call.

When later analyzing an object file that utilizes a learned API, we continue data-flow analysis at all calls to the API. The analyzer attempts to statically recover the value passed to the API call. By composing the cached data-flow summary function with data dependencies to the API call site discovered via object code analysis, we can recover the argument value used at the system call inside the library.

4.2 Context-Sensitive Argument Recovery

Static argument recovery uses data-flow analysis to identify system-call values that are statically known. The analysis recovers arguments using a finite-height lattice of values and an algebra that defines how to combine values in the lattice. The lattice has a bottom element ($\bot$) that indicates nothing is known about an argument because the argument has not been analyzed. The top element ($\top$) is the most general value and means that an argument could not be determined statically.

Argument values may reach a system call via multiple, different execution paths, as shown in Fig. 1. The algebra of the lattice defines how to compute the value that will flow down the converged execution path. The join operator ($\sqcup$) combines values. Our previous static argument analysis [10] recovered arguments using a standard powerset lattice P. For S the finite set of statically known strings and integers used by the program, lattice values were elements of $D_P = \mathcal{P}(S)$ with $\bot_P = \emptyset$ and $\top_P = S$. The algebra joined arguments with set union: $A \sqcup_P B = A \cup B$ for A and B any lattice values. The value reaching the system-call site is the recovered argument.

Joins in lattice P diminish the precision of the analysis. The set union does not maintain the association between an argument value and the execution path using that value. As a result, an attacker can undetectably use a value recovered on one path on any other execution path reaching the system call. Suppose a program opens both a temporary file with write privileges and a critical file with read-only access. Even if argument recovery can identify all arguments, the calling context is lost. The attacker can use the write privilege from the temporary-file open to open the critical file with write privilege as well.

Worse yet is the effect of values not recovered statically. If an argument cannot be identified on one execution path, it takes the value $\top_P$. At a point of converging execution, such as the entry point of a C library function, the join of $\top_P$ with any recovered value A discards the recovered value because $A \sqcup_P \top_P = \top_P$. This makes intuitive sense: when monitoring execution, the monitor cannot determine when a recovered value should be enforced without knowing the calling context of the value.

We solve this imprecision by extending the lattice domain to include calling context. Our new data-flow analysis annotates the recovered string and integer values with the address of the call site that passes the strings or integers as an argument. Stated differently: we recover values using a *separate* powerset lattice for each calling context. As a data value propagates through a call instruction, the analyzer annotates the value with the return address of the call. We have found that a single call site provides enough context to sufficiently distinguish argument values, although this analysis could be extended to include additional calling context as necessary. Note that the call site annotation is not the call site nearest to the system call, but rather the originating call site where the argument is first set. The originating call site may target any function in the program, including C library calls or arbitrary wrapper functions around library functions.

Data values recovered by our data-flow analysis are pairs (A, c), where $A \in D_P$ is a set of integers or strings as above, and c is the calling context information.

Definition 1. *Let P be the powerset lattice over the set S of all statically-known strings and integers used in the program, as defined above. Let $C = \{c_0, \ldots, c_n\}$ be call site identifiers, with $c_0 = \emptyset$ the special identifier indicating that no context information is known. Let Q be the context-sensitive data-flow lattice defined with domain $D_Q = \mathcal{P}(D_P \times C)$, $\bot_Q = \{(\bot_P, \emptyset)\}$, and $\top_Q = \bigcup_{i=0}^{n} \{(\top_P, c_i)\}$.*

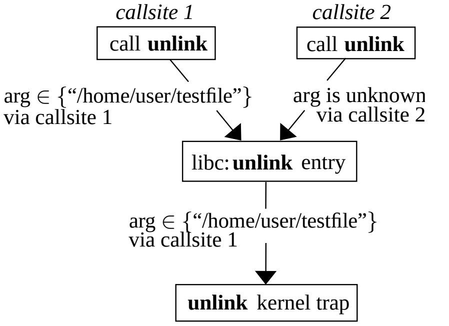
callsite 1
call **unlink**
callsite 2
call **unlink**
arg ∈ {"/home/user/testfile"} via callsite 1
arg is unknown via callsite 2
libc: **unlink** entry
arg ∈ {"/home/user/testfile"} via callsite 1
unlink kernel trap

Fig. 7. Static argument recovery with context-sensitive argument values

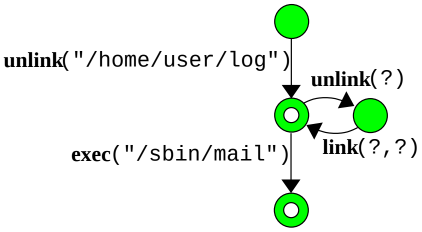

Fig. 8. The model for the program code of Fig. 3 with context-sensitive argument values. Note that the argument is constrained on the top-most **unlink** transition even though the argument at another **unlink** call site could not be statically determined

Let $A, B \in D_Q$ *be* $A = \{(A_i, x_i)\}_i$ *and* $B = \{(B_j, y_j)\}_j$ *with* $\forall i : A_i \in D_P, x_i \in C$*;* $\forall j : B_j \in D_P, y_j \in C$*; and* $x_0 = \emptyset = y_0$*. Define the join operator* $\sqcup_Q$ *as:*

$$A \sqcup_Q B = \{(A_i \sqcup_P B_j, x_i) \,|\, x_i = y_j\} \cup \tag{1}$$
$$\cup \{(A_i \sqcup_P B_0, x_i) \,|\, \nexists j \; x_i = y_j\} \cup \{(B_j \sqcup_P A_0, y_j) \,|\, \nexists i \; x_i = y_j\} \,. \tag{2}$$

The join operation of Q maintains calling context information at points of execution path convergence. Part (1) joins values in the powerset lattice P only when those values have identical calling context. Part (2) maintains correctness when joining against a value that does not yet have context: the value may occur in any previously-identified context. The lattice Q improves prior data-flow analysis in two important ways:

1. The convergence of a context-sensitive value with an unrecovered value is non-destructive. The analyzer can continue to propagate the known value with execution context (Fig. 7). Figure 8 shows the model for the example code with context-sensitive arguments. The statically known filename passed to the first call to **unlink** (call site 1 in Fig. 7) constrains that call. Intuitively, we need not discard the recovered context-sensitive value because the monitor, at runtime, can compare the value's context information with the executing process' call stack to determine if the argument restriction should be enforced.
2. When multiple context-sensitive values converge, no information is lost. Distinct calling contexts remain distinct. By preserving context, we can enforce the association between multiple arguments passed to a system call at the same call site. Recall the previous example of opening both a temporary file and a critical file with different access privileges. Since our analysis will annotate both the filename and the access mode at each call site with that site's calling context, an attacker cannot open the critical file with anything other than read-only access.

The monitor enforces an argument restriction only when the execution path followed to the system call contains the call-site address annotating the argument value. The monitor walks the call stack of the process at every system call to identify the calling context of the system call. If the call-site address that annotates a value exists in the calling context, the monitor enforces the corresponding argument restriction. If no argument was recovered for a particular context, the monitor will not constrain allowed values at runtime.

5 Environment-Sensitive Models

Environment-sensitive intrusion detection further restricts allowed process execution based upon the known, fixed data in the execution environment. Environment-sensitive program models do not include the data directly, but rather encode dependencies to environment data that will be evaluated immediately before the process begins execution.

We first formalize the notions of environment properties and dependencies between the environment and a program.

Definition 2. *The* environment *is program input known at process load time and fixed for the entire execution of the process.*

This includes environment variables, command-line parameters, and configuration file contents. The definition excludes environment variables altered or overwritten during execution. In our measurements, only about 3% of the programs installed with Solaris 8 modify at least one environment variable.

Definition 3. *A* property *of the environment is a single variable, parameter, or configuration setting in a file.*

A property may be present or omitted in the environment, and, if present, may have an associated value. An environment dependency captures the relation between environment properties and the program's execution behavior.

Definition 4. *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*

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

In words: over all possible executions, a program data value at p depends only upon the value of x. The function f characterizes how the data value depends upon the environment property.

The definition is intuitively similar to the definition of a def-use relation in programming language analysis [15]. The environment defines a data value that is later used by the executing process. Where existing program analyses examine only relations between instructions in the program, we extend the notion of value definition to the environment.

Dependencies are of interest only if they affect program behavior visible to the execution monitor. We focus on two classes of dependencies, both of which are present in the example code of Fig. 3. *Control-flow dependencies* exist at program branches where the branch direction followed depends upon an environment property. *Data-flow dependencies* occur when a visible data value, such as a system-call argument, is dependent upon the environment. The value of the environment property flows to the system-call argument.

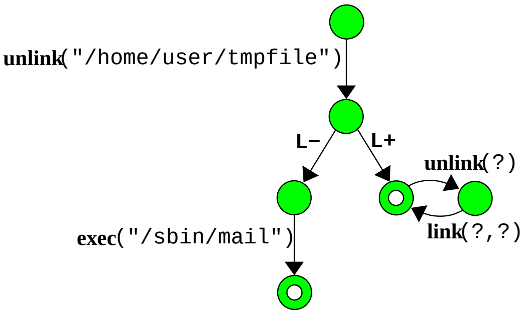

Fig. 9. Dyck model with environment branch dependencies. The symbols `L-` and `L+` are branch predicates that allow subsequent system calls when the command-line parameter "–L" is omitted or present, respectively

5.1 Control-Flow Dependencies

Control-flow dependencies restrict allowed execution paths based upon the values of the environment. The variable tested at a program branch may be dependent upon an environment property. For example, the `if` statement of line 16 guards the **exec** call so that it executes only when "–L" is omitted from the command line. The program's data variable used in the branch test is dependent upon "–L", as in Definition 4. As an immediate consequence, the branch direction followed depends upon "–L". Similarly, the `switch` statement at line 8 has an environment control-flow dependency upon "–L" and will execute the `case` at line 9 only when "–L" is present.

The static analyzer can encode control-flow dependencies into the Dyck model with predicate transitions. Figure 9 shows the model of Fig. 8 with predicate transitions characterizing the environment dependency. The predicate `L-` is satisfied only when the command line does not contain "–L". Likewise, `L+` is satisfied when "–L" is present.

The execution monitor evaluates predicate transitions when loading the model for a program about to execute. Predicates satisfied by the environment become ϵ-transitions. An ϵ-transition is transparent and allows all events following the transition. Conversely, the monitor deletes edges with predicates that are not satisfied by the environment, as legitimate process execution cannot follow that path. If the command line passed to the example code of Fig. 3 does not contain "–L", then the `L-` transition in Fig. 9 will allow the subsequent **exec** and the `L+` transition will be removed to prevent the model from accepting the following **unlink** and **link** calls.

5.2 Data-Flow Dependencies

System-call argument values may also depend upon environment properties. In particular, programs frequently use environment values when computing strings passed to system calls as filenames. These values can significantly restrict the allowed access of the process, and hence an attacker that has subverted the process. In the example code (Fig. 3), the environment variable `TMP` gives the system temporary directory used as the prefix to the filename argument of lines 11 and 12. The property constrains the **unlink** at line 11 so that the only files it could remove are temporary files. The parameter to the command-line property "–L" fully defines the filename passed as the first argument to **link**. Many real-world programs exhibit similar behavior. The Apache web server, for example, uses the command-line property "–d" to specify the server's root directory [11].

Environment data-flow dependencies augment existing system-call arguments recovered using techniques from Sect. 4. Figure 5 adds argument dependencies to the previous model of Fig. 9. A bracketed environment property indicates that the argument is simply a template value and must be instantiated with the actual value of the property at program load time.

Figure 5 is the completed environment-sensitive Dyck model with context-sensitive argument encoding. When the program of Fig. 3 is loaded for execution, the monitor reads the current environment and instantiates the model in that environment. Template argument values are replaced with the actual values of the environment properties upon with the argument depends. The final, instantiated models appear in Fig. 6, as described in Sect. 3.

5.3 Dependency Identification

This paper aims to demonstrate the value of environment-sensitive intrusion detection and does not yet consider the problem of automated dependency identification. We assume that environment dependencies have been precomputed or manually specified.

In our later experiments, we manually identified environment dependencies via *iterative model refinement*. At a high-level, this process parallels counterexample-guided abstraction refinement used in software model checking: the Dyck model is an abstraction defining correct execution, and we iteratively refine the model with environment dependencies to improve the abstraction [2]. We monitored a process' execution and collected a trace of reachable and potentially malicious system calls as described in Sect. 6. The trace included the calling context in which each potentially malicious call occurred. We inspected the program's code to determine if either:

- The argument passed to a call-site in the calling context depended upon environment information and reached the system call; or
- A branch guarded one of the call-sites and the branch predicate depended upon the environment.

Function-call arguments and branch predicates depend upon the environment if a backward slice of the value reaches a function known to read the environment, such as `getenv` or `getopt`. We added the dependency to the Dyck model and repeated the iteration. In practice, the number of dependencies added via iterative refinement was small: each program in our experiments contained between 10 and 24 dependencies.

Manual specification clearly has drawbacks. It requires the user to understand low-level process execution behavior and Dyck model characteristics. Manual work is error-prose and can miss dependencies obscured by control-flows that are difficult to comprehend. However, we believe that dependency identification is not limited to manual specification.

We postulate that automated techniques to identify environment dependencies with little or no direction by an analyst are certainly feasible. Summary functions for C library calls that read the environment would enable our existing static data-flow analysis to automatically construct environment-dependent execution constraints. Complex dependencies could be learned via dynamic analysis. A dynamic trace analyzer could correlate environment properties with features of an execution trace to produce dependencies.

This paper makes clear the benefits of model specialization based upon environment dependencies. The improvements noted in Sect. 7 motivate the need for implementation of the techniques to automatically identify dependencies. We expect future work will address these implementation issues.

6 Average Reachability Measure

Measurements of a model's precision and its ability to prevent attacks indicate the benefits of various analyses and model construction techniques. Previous papers have measured model precision using the average branching factor metric [5, 9, 10, 20, 22]. This metric computes the average opportunity for an attacker who has subverted a process' execution to undetectably execute a malicious system call. After processing a system call, the monitor inspects the program model to determine the set of calls that it would accept next. All potentially malicious system calls in the set, such as **unlink** with an unconstrained argument, contribute to the branching factor of the current monitor configuration. The average of these counts over the entire execution of the monitor is the average branching factor of the model. Lower numbers indicate better precision, as there is less opportunity to undetectably insert a malicious call. The set of potentially malicious system calls was originally defined by Wagner [22] and has remained constant for all subsequent work using average branching factor.

Average branching factor poorly evaluates context-sensitive program models with stack update events, such as the Dyck model used in this paper. Typical programs have two characteristics that limit the suitability of average branching factor:

- Programs often have many more function calls and returns than system calls. The number of stack update events processed by the monitor will be greater than the number of actual system-call events.
- Programs rarely execute a system-call trap directly. Rather, programs indirectly invoke system calls by calling C library functions.

These characteristics have important implications for both the stream of events observed by the monitor and the structure of the Dyck model. The first characteristic implies that stack updates dominate the event stream. The second characteristic implies that at any given configuration of the monitor, the set of events accepted next are predominantly safe stack update events that do not contribute to the configuration's branching factor. In fact, a potentially malicious system call is not visible as the next possible event until the process' execution path has entered the C library function and the monitor has processed the corresponding stack event for that function call. The number of potentially malicious system calls visible to the monitor decreases, artificially skewing the computed average

Table 1. Test programs, workloads, and instruction counts. Instruction counts include instructions from any shared objects used by the program

Program	*Workload*	*Instruction Count*
procmail	Filter a 1 MB message to a local mailbox.	374,103
mailx	Send mode: send one ASCII message. Receive mode: check local mailbox for new email.	207,977
gzip	Compress 13 MB of ASCII text.	196,242
cat	Write 13 MB of ASCII text to a file.	185,844

branching factor downward. The call-stack-based model is not as precise as its average branching factor makes it appear.

We have extended average branching factor so that it correctly evaluates context-sensitive models with stack update events and does not skew results. Our *average reachability measure* uses context-free language reachability [23] to identify the set of actual system calls reachable from the current configuration of the monitor. Rather than simply inspecting the next events that the monitor may accept, the average reachability measure walks forward through all stack events until reaching actual system calls. The forward inspection respects call-and-return semantics of stack events to limit the reachable set of system calls to only those that monitor operation could eventually reach. After each actual system-call event, we recalculate the set of reachable system calls and count the number that are potentially malicious. The sum of these counts divided by the number of system calls generated by the process is the average reachability measure.

The average reachability measure subsumes average branching factor. Both metrics have the identical meaning for context-insensitive models and for context-sensitive models without stack events, such as Wagner and Dean's *abstract stack* model [20], and will compute the same value for these model types. Average reachability measures for call-stack-based models may be directly compared against measures for other models, allowing better understanding of the differences among the various model types.

We implemented the average reachability measure using the `post*` algorithm from push-down systems (PDS) research [4]. We converted the Dyck model into a PDS rule-set and generated `post*` queries following each system call. The `post*` algorithm is the same as that used by Wagner and Dean to operate their abstract stack model. Note that we use the expensive `post*` algorithm for evaluation purposes only; the monitor still verifies event streams via the efficient Dyck model.

7 Experimental Results

We evaluated the precision of environment-sensitive program models using average reachability. A precise model closely represents the program for which it was constructed and offers an adversary little ability to execute attacks undetected. To be useful, models utilizing environment sensitivity and our argument analysis should show improvement over our previous best techniques [5, 10]. On test programs, our static argument recovery improved precision by 61%–100%. Adding environment sensitivity to the models increased the gains to 76%–100%. We end by arguing that model-based

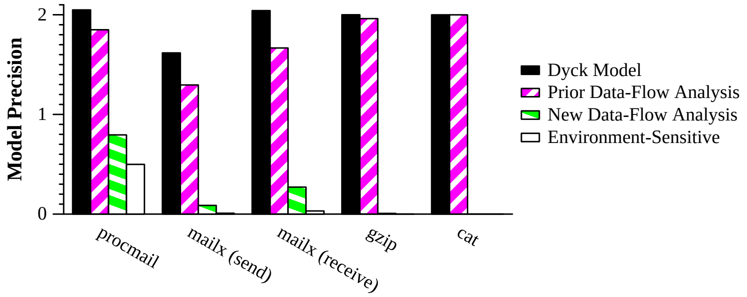

Fig. 10. Precision of program models with increasing sensitivity to data-flows and the environment. The y-axis indicates precision using the *average reachability measure*: the average number of reachable and potentially malicious system calls. Lower numbers indicate greater precision and less opportunity for attack. All programs have 4 bars; bars that do not show on the graph have value less than 0.01

intrusion detection systems that ignore environment information leave themselves susceptible to evasion attacks.

7.1 Test Programs

We measured model precision for four example UNIX programs. Table 1 shows workloads and instruction counts for the programs tested. Note that instruction counts include instructions from all shared objects on which the program depends. `Procmail` additionally uses code in shared objects loaded explicitly by the program via `dlopen`. As our static analyzer does not currently identify libraries loaded with `dlopen`, we manually added the dependencies to this program.

These programs, our static analyzer, and our runtime monitor run on Solaris 8 on SPARC. The monitor executes as a separate process that traces a process' execution via the Solaris `/proc` file system. To generate stack events for the Dyck model, the monitor walks the call stack of the process before every system call, as done by Feng *et al.* [6]. By design, the full execution environment of the traced process is visible to the monitor. The environment is actually passed to the monitor, and the monitor then forks and executes the traced process in that environment with an environment-sensitive model.

7.2 Effects of Static Argument Analysis

We used average reachability to evaluate models constructed for these four test programs. We compared three different versions of the Dyck model using varying degrees of static data-flow analysis (Fig. 10). We report two sets of results for `mailx` because it has two major modes of execution, sending and receiving mail, that produce significantly different execution behavior. Other programs with modes, such as compressing or decompressing data in `gzip`, did not exhibit notable changes in precision measurements.

Table 2. Environment dependencies in our test programs. We manually identified the dependencies via inspection of source code and object code

Program	*Environment dependencies*
procmail	• Program branching depends upon “–d” command-line argument. • Program branching depends upon “–r” command-line argument. • Filename opened depends upon user’s home directory.
mailx	• Program branching depends upon “–T” command-line argument. • Program branching depends upon “–u” command-line argument. • Program branching depends upon “–n” command-line argument. • Filename created depends upon the parameter to the “–T” command-line argument. • Filename opened depends upon the `TMP` environment variable. • Filename opened depends upon the user’s home directory. • Filename unlinked depends upon the `TMP` environment variable.
gzip	• Argument to **chown** depends upon the filename on the command line. • Argument to **chmod** depends upon the filename on the command line. • Filename unlinked depends upon the filename on the command line.
cat	• Filename opened depends upon the filename on the command-line.

First, we used a Dyck model without any data-flow analysis for system-call argument recovery. Although there is some overlap between our current test programs and test programs previously used with a Dyck model [10], we reiterate that the results computed here by the average reachability measure are *not* comparable to average branching factor numbers previously reported for the Dyck model. Our current results may be compared with previous average branching factor numbers for non-stack-based models [9, 20].

Second, we added system-call argument constraints to the Dyck model when the constraints could have been recovered by a previously reported analysis technique [9, 10, 20]. Arguments values are recovered only when a value is recovered along all execution paths reaching a system call. If the value from one execution path cannot be identified statically, then the entire argument value is unknown. Furthermore, any data-flows that cross between a shared object and the program are considered unknown. This limited data-flow analysis improved model precision from 0% to 20%.

Last, we enabled all static data-flow analyses described in Sect. 4. Our new argument analysis improved precision from 61% to 100%.

7.3 Effects of Environment Sensitivity

We then made the models environment sensitive. For each program, we manually identified execution characteristics that depended upon environment properties. Stated more formally, we defined the functions f of Definition 4 that describe data-flows from an environment property to a program variable used as a system-call argument or as a branch condition. Table 2 lists the dependencies added to the Dyck model for each program. The system-call argument dependencies augmented values recovered using the static data-flow analyses presented in Sect. 4. Immediately before execution, the monitor instantiates the model in the current environment by resolving the dependencies.

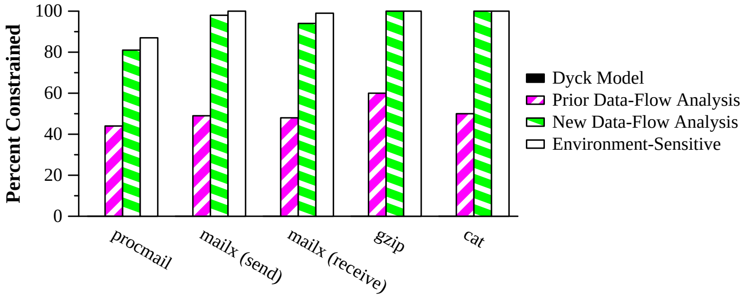

Fig. 11. Percentage of potentially malicious system calls identified by the average reachability measure made safe by constraints upon their arguments. The Dyck model with no data-flow analysis constrained no arguments

Figure 10 reports the average reachability measure for each program's execution when monitored using these environment-sensitive models. Model precision has improved from 76% (`procmail`) to 100% (`gzip` and `cat`). Both `gzip` and `cat` had average reachability measures of zero, indicating that an adversary had no opportunity to undetectably insert a malicious system call at any point in either process' execution.

Successful argument recovery constrains system calls so that an attacker can no longer use the calls in a malicious manner. We evaluated the ability of our techniques to constrain system calls. Figure 11 shows the percentage of potentially malicious system calls discovered during computation of the average reachability measure that were restricted because of system call argument analysis and environment-sensitivity. In this figure, higher bars represent the improved constraints upon system calls that produced the correspondingly lower bars previously shown in Fig. 10. For three programs, `mailx`, `gzip`, and `cat`, environment-sensitive models constrained 99–100% of the potentially dangerous calls.

We expect environment-sensitive program models to affect the performance of runtime execution monitoring. The monitor must both update the program model at load time to remove paths unreachable in the current environment and enforce context-sensitive argument restrictions at every system call. Table 3 shows the execution time overhead arising from the model update and the more precise enforcement. These overheads are modest: about one-half second for the short-lived processes `procmail` and `mailx` and two seconds for the longer-running `cat`. Although the overheads for `procmail` and `mailx` are high when viewed as a percentage of the original runtime, this occurs due to the short lifetime of these processes and the monitor's upfront fixed cost of pruning unreachable paths. Longer-lived processes such as `cat` give a better indication of relative cost: here, 2.8%.

Further, improved argument recovery may increase the size of program models as the model must contain the additional constraints. For all programs, environment-sensitive models required 16 KB (2 pages) more memory than a Dyck model with no argument recovery or environment-sensitivity.

Table 3. Performance overheads due to execution enforcement using environment-sensitive models. *Model update* is the one-time cost of pruning from the model execution paths not allowed in the current environment. The *enforcement* times include both program execution and verification of each system call executed against the program's model

Program	*No model update*	*Environment-sensitive*			*Overhead*
	No enforcement	*Model update*	*Enforcement*	*Total*	
procmail	0.55 s	0.41 s	0.67 s	1.08 s	0.53 s
mailx (send)	0.08 s	0.38 s	0.16 s	0.54 s	0.46 s
mailx (receive)	0.07 s	0.38 s	0.14 s	0.52 s	0.45 s
gzip	6.26 s	0.00 s	6.11 s	6.11 s	0.00 s
cat	56.47 s	0.00 s	58.06 s	58.06 s	1.59 s

We believe that these results strongly endorse our proposed environment-sensitive intrusion detection. The precision measurements demonstrate that with the right analysis tools, program execution can be safely constrained to the point that attackers have little ability to undetectably execute attacks against the operating system via a vulnerable program. We certainly do not constrain all execution: for example, our models do not enforce iteration counts on loops or verify data read or written to files. However, we strongly limit process execution that can adversely affect the underlying operating system or other processes executing simultaneously.

7.4 Evasion Attacks

Intrusion detection systems that are not environment-sensitive are susceptible to evasion attacks. These attacks mimic correct process execution for *some* environment [18, 21], just not the current environment. To demonstrate the effectiveness of environment sensitivity in defense against such attacks, we designed an attack against `mailx` that overwrites command-line arguments stored in the process' address space to change the process' execution. Although the original command line passed to the program directed it to check for new mail and exit, our attack changes the environment data so that `mailx` instead reads sensitive information and sends unwanted email.

Our attack makes use of a buffer overrun vulnerability when `mailx` unsafely copies the string value of the `HOME` environment variable. We assume that the attacker can alter the `HOME` variable, possibly before the monitor resolves environment dependencies. The attacker changes the variable `HOME` to contain the code they wish to inject into `mailx`. The exploit follows the typical "nop sled + payload + address" pattern [12].

1. The first part consists of a sequence of nops (a "sled") that exceeds the static buffer size, followed by an instruction sequence to obtain the current address on the stack.
2. The payload then rewrites the command-line arguments in memory. The change to the command-line arguments alters execution so that the process will perform a different operation, here sending spam and leaking information.
3. The return address at the end of the payload is selected to reenter `getopt` so that the new command-line arguments update appropriate state variables. If necessary, an evasive exploit can alter its reentry point so that no additional system calls or

stack frames occur between the overflow and the resumed flow. In our attack, reentering at `getopt` was sufficient.

We implemented the `mailx` exploit, loaded it via `HOME`, and caused the program to read arbitrary files and send unwanted email. Since the exploit did not introduce additional system calls and reentered the original execution path, the attack perfectly mimicked normal execution for some environment, with one exception caused by the register windows used by the SPARC architecture. To effectively manipulate the return address, exploit code must return from a *callee* function after corrupting the stack [12]. This "double return" makes exploit detection slightly easier on SPARC machines, because an exploit that attempts to reenter a function alters return addresses in a detectable way. This attack limitation is not present on the more common x86 architecture.

Environment-sensitive models can detect these evasion attacks. The monitor resolves environment dependencies before process execution begins, and hence before the attack alters the environment data. In this example, the execution paths that `mailx` followed subsequent to the attack, reading sensitive files and sending email, do not match the expected paths given the command-line input.

8 Conclusions

Program models used for model-based intrusion detection can benefit from our new analyses. Our static argument recovery reduces attack opportunities significantly further than prior argument analysis approaches. Adding environment sensitivity continues to strengthen program models by adding environment features to the models. The usefulness of these model-construction techniques is shown in the results, where the models could severely constrain several test programs' execution.

Acknowledgments

We thank the anonymous reviewers and the members of the WiSA project at Wisconsin for their helpful comments that improved the quality of the paper.

Jonathon T. Giffin was partially supported by a Cisco Systems Distinguished Graduate Fellowship. Somesh Jha was partially supported by NSF Career grant CNS-0448476. This work was supported in part by Office of Naval Research grant N00014-01-1-0708 and NSF grant CCR-0133629. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notices affixed hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

References

1. R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. In *9th European Symposium on Research in Computer Security*, Sophia Antipolis, France, Sept. 2004.

2. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, Chicago, IL, July 2000.
3. H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805–822, 1999.
4. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, Chicago, IL, July 2000.
5. 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, CA, May 2004.
6. 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, CA, May 2003.
7. L. Fix and F. B. Schneider. Reasoning about programs by exploiting the environment. In *21st International Colloquium on Automata, Languages, and Programming*, Jerusalem, Israel, July 1994.
8. D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.
9. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
10. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2004.
11. httpd. Solaris manual pages, chapter 8, Feb. 1997.
12. J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2003.
13. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security*, pages 326–343, Gjøvik, Norway, Oct. 2003.
14. L.-c. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sept. 2004.
15. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
16. 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*, Bolton Landing, NY, Oct. 2003.
17. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
18. K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *5th International Workshop on Information Hiding*, Noordwijkerhout, Netherlands, October 2002.
19. U.S. Department of Energy Computer Incident Advisory Capability. M-026: OpenSSH uselogin privilege elevation vulnerability, Dec. 2001.
20. D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
21. D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.
22. D. A. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD dissertation, University of California at Berkeley, Fall 2000.
23. M. Yannakakis. Graph-theoretic methods in database theory. In *ACM Symposium on Principles of Database Systems*, Nashville, TN, Apr. 1990.