# Guarded Models For Intrusion Detection

Hassen Saïdi

Computer Science Laboratory
SRI International
saidi@csl.sri.com

## Abstract

Host-based intrusion detection systems that monitor an application execution and report any deviation from its statically built model have seen tremendous progress in recent years. However, the weakness of these systems is that they often rely on overly abstracted models that reflect only the control flow structure of programs, and therefore are subject to so-called "mimicry attacks". Authors of these models have argued that capturing more of the data flow characteristics of a program is necessary to prevent a large class of attacks, in particular, non-control-data attacks. In this paper, we present the guarded model, a novel model that addresses the various deficiencies of the state-of-the-art intrusion detection systems. Our model is a generalization of previous models that offers no false alarms, a very low monitoring overhead, and is automatically generated. Our model detects mimicry attacks by combining control flow and data flow analysis, but can also tackle the ever increasingly threatening non-control-data flow attacks. Our model is the first model built automatically by combining control flow and data flow analysis using state-of-the-art tools for automatic generation and propagation of invariants. Our model not only prevents intrusions, but allows in some cases the detection of application logic bugs. Such bugs are beyond the reach of current intrusion detection systems.

***Categories and Subject Descriptors*** D [*4*]: 6 Security and Protection

***General Terms*** Security, Verification

***Keywords*** Static analysis, dynamic analysis, invariant generation, intrusion detection

## 1. Introduction

It has been shown [15] that finding anomalies in the stream of system calls issued by user applications is an effective host-based intrusion detection capability. Wagner and Dean [32] proposed an approach by which static analysis is used to derive a model of application behavior resulting in a host-based intrusion detection system with three advantages: a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms. Automation is guaranteed by extracting the model from the control structure of the source code of the program. Detection of attacks rely on the fact that code injection often results in the execution of a system call that is not invoked by the application at all, or a sequence of system calls that are executed in an order not allowed by the application code. Finally, false alarms are eliminated because the model is an overapproximation of the behavior of the application. Therefore, while the model might be porous and not detect attacks, it will never raise a false alarm. Starting with Wagner and Dean's paper, various models capturing the sequences of legitimate system calls have been proposed [32, 18, 13, 16, 8].

Current state-of-the-art host-based intrusion detection systems approaches struggle with several issues. The most important one is the precision of the model. The more precise the model is, the less an attack is possible. Current models are limited when it comes to handling complex control structures such as loops, recursive function calls, and circuits in function calls. These limitations produce today's models in forms of very crude overapproximations of the actual behavior of the application and are all subject to so-called *mimicry attacks* [33] that produce a sequence of system calls that conforms to the model but is not a sequence in the original code. The imprecision of state-of-the-art models comes from the weakness of the static analysis methods employed to construct them. These weaknesses can be summarized by the absence of any significant data flow analysis. Mimicry attacks exploit the nondeterminism that arises from the imprecision of the models. A new wave of attacks called, *non-control-data attacks* [10] exploit the nondeterminism in the application code itself and therefore allow the subversion of the intended data flow. We observe that the attacks mentioned above are all preventable by making sure that the application models used for intrusion detection capture as precisely as possible the semantics of the program in a simple and intuitive way. We observe that mimicry and non-control-data attacks always inevitably violate an *invariant* of the program, and that any approach to static-analysis-based intrusion detection will be weakened by the absence of such invariants from application models.

In this paper, we propose a novel model that can reduce the risk of mimicry attacks, but can also detect the ever increasingly threatening non-control-data attacks. Our model is a generalization of previous models and offers no false alarms, and a very low monitoring overhead, and is automatically generated. In some cases, our model is precise enough that it is possible to automatically check the application for the presence of application logic bugs. Our model, called the *guarded model*, is a control flow graph where system calls are guarded by invariants. We ensure both control flow and data flow integrity by making sure that system calls not only occur in the order specified by the model, but that each system call must be preceded by a check that some program invariant generated by static analysis holds. Such invariants capture crucial properties about system call arguments, system call return values, input variables, and the values of branch predicates at all control locations of the program. Weaker versions of these properties are gen-



*PLAS'07* June 14, 2007, San Diego, California, USA.

erated manually in [16], or learned from runs of the application in an attack-free environment in [8] to form an underapproximation of the application behavior, which leads to an intrusion detection approach that generates many false alarms.

The contribution of our work can be summarized as follows:

- A novel model for intrusion detection, generated by static and dynamic analysis. The model is built using a combination of predicate abstraction, invariant generation techniques, and interprocedural invariant propagation.
- Our model is from the theoretical point of view more precise than the models used by the state-of-the-art host-based intrusion detection systems [16, 8] and therefore reduces the space of possible attacks.
- Our model is built fully automatically from source code and combines control flow and data flow properties of the application. In particular, our model allows the detection of mimicry attacks generated by various automated mimicry attack generation tools [17, 24].
- Our model treats in a uniform way both mimicry and non-control-data attacks thanks to the use of invariants.

The rest of the paper is organized as follows: In Section 2, we describe a toolbus architecture that allows the plugin of several cooperating invariant generation tools to build guarded models. In Section 3 we illustrate in a series of simple but illustrative examples the deficiencies of the state-of-the-art models used in intrusion detection systems and how our approach, based on the construction of a guarded model, addresses those deficiencies. In Section 4 we formally define our model, and in Section 5 we describe the automatic process by which guarded models are built. In Section 6 we illustrate the properties of our model by showing that it is more precise than the state-of-the-art models, provides a way of detecting a much broader class of attacks, detects all attacks generated automatically by some recent methods described in the literature [24], and provides a way of detecting logic bugs, which is not a feature covered by any other intrusion detection system.

## 2. An Architecture for Building and Using Guarded Models

The ability of guarded models to be successful at detection and preventing actual attacks rely heavily on our ability to capture program invariants. Our approach is effective because our invariants are generated by using a combination of several complementary methods and tools implementing both static and dynamic code analysis. First, assertions are generated from observed runs during a training period in an attack-free environment using Daikon [12]. Daikon generates *likely invariants* from execution traces gathered using Valgrind [26]. Likely invariants are assertions or relationships between program variables that are true at some control location during the observed runs, but may or may not be true in all possible runs of the program. We use the BLAST assertion checker for C programs [21] to prove or disprove the fact that the likely invariants hold at the locations indicated by Daikon. Daikon generates assertions at the entry point and exit point of all functions appearing in the application's code. We extended Daikon to generate assertions at arbitrary program locations. In particular, we generate likely loop invariants that are often crucial in determining invariants at the exit point of complex functions. Daikon also ignores calls to libraries and system calls and will not generate invariants at the entry point of a library implementing a system call. We have extended Daikon in such a way that likely invariants are generated at program locations where system calls are invoked. We also automatically generate invariants expressing linear relationships among integer program variables that hold at all control locations of a program using the Inv-Gen tool described in [20]. Inv-Gen also uses the notion of logical lattices that allows the generation of linear invariants in the form of assertions expressed in a combination of theories including linear arithmetic, uninterpreted functions symbols, and arrays. Uninterpreted function symbols are useful in abstracting away function calls.

We add the linear invariants to BLAST as additional information about the program so that it maximizes BLAST's chances of proving or disproving likely invariants. BLAST proves assertions using predicate abstraction [19]. That is, it partitions the reachable states of the program using a set of predicates appearing in the assertion to prove, the conditional branches and assignments as well as internally generated predicates from failed proofs using Craig's interpolation theorem [23], in order to obtain a finite representation of the program. BLAST's assertion checking process amounts to unfolding the control graph of the program, and evaluating the set of predicates in every control location, effectively propagating the predicates throughout the control graph of the program. The propagation works as follows: if an assignment instruction does not modify the variables in a predicate, then the predicate holds after the assignment. If, however, the program variables in the predicate are modified by the instruction, it is necessary to either prove that the post condition of the assignment implies that the predicate still holds, or its negation holds. If neither implication is proved, then the successor control node is split into a node where the predicate holds and a node where the negation of the predicate holds, introducing therefore some nondeterminism in the control flow graph. Proofs are discharged automatically using efficient decision procedures that can process thousands of proofs per second.

The result of our invariant generation and assertion checking process is a control flow graph where each program control point is annotated with a set of predicates that are true at that location. That is, an *abstract state graph* that captures both the control and the data flow properties of a program. Consider a set of predicates $P_1, \cdots, P_k$ over program variables. The abstract state graph of a program $P$ is a graph where each node is a pair $\langle v, e\rangle$, where $v$ is a control location and $e$ is a valuation of the predicates $P_1, \cdots, P_k$ at control location $v$. For instance, for $k = 3$, a node $\langle pc1, (0, 1, 1)\rangle$ indicates that at location $pc1$, the predicate $P_1 \wedge \neg P_2 \wedge \neg P_3$ is true. Figure 1 shows a simple program and its abstract state graph.

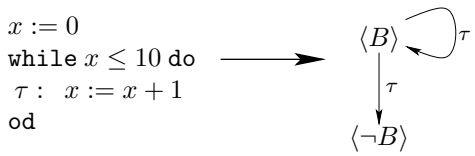


**Figure 1.** A Simple Program and its Nondeterministic Abstract State Graph Built Using the Predicate $\mathtt{B} \equiv x \leq 10$.

Unlike any other model construction approach for intrusion detection, we do not consider the control flow graph of an application but rather the much more precise abstract state graph. Early in their paper [19], Graf and Saïdi note that *an abstract state graph also represents a precise global control flow graph of the system*. Invariants that are true at control locations where system calls are invoked become guards that should evaluate at runtime to true before allowing a system call. This ensures both control flow and data flow integrity. Abstract state graphs have many interesting properties. For instance, if the abstract state graph is deterministic, then there is an equivalence between the application's code and its abstract state graph. That is, the model is not an overapproximation of the application behavior but is an exact abstraction. Therefore, not only a deviation from the model is an attack, but the model itself is

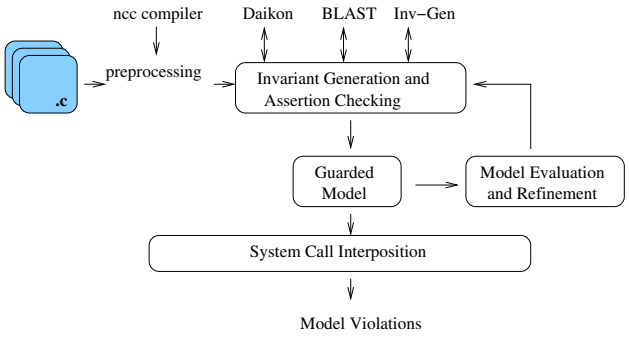


**Figure 2.** Guarded Models Construction and Monitoring Architecture

a finite faithful representation of the original program that can be easily evaluated against a specification of application logic bugs.

Our prototype implementation and monitoring framework is described in Figure 2. Invariant generation tools and BLAST are used to build the guarded model of the entire application. When the application deviates from the model, we detect an intrusion. If we include in the model the likely invariants or hypotheses that have not been established as invariants by BLAST, we can monitor when these likely invariants are violated and log the sequences of executions in which such violations occur. Such sequences can then be examined by complementary statistical approaches to conclude whether such rarely observed sequences are truly vulnerable or are legitimate. The use of Daikon, Blast, and Inv-Gen required some preprocessing capability that was necessary to deal with the limitation of each of these tools.

Guarded models as well as previously proposed models are useful only if they detect actual attacks. Evaluating if a model can be defeated by mimicry attacks addresses such a concern. The work described in [17] provides a simple and effective way of detecting if a given model is vulnerable to mimicry attacks by using model-checking techniques to automatically check the model against a specification of an attack expressed as the effect of the attack on the operating system state. Guarded models have the property of simplifying further such analysis. An undetected attack is any attacks that executes a sequence of system calls accepted in the model but contains a nondeterministic transition. A simple graph algorithm can be used to extract all sequences that contain such nondeterministic transitions. These sequences can then be checked using a model checking procedure similar to the one used in [17]. When a sequence representing an attack is discovered, the problem of refining the model is reduced to the problem of refinement of a predicate abstraction, and can be addressed by several techniques developed in the literature. In particular, the techniques developed in [28] and [29] that define the problem of refinement of a predicate abstraction as the problem of eliminating nondeterminism in an abstract state graph.

## 3. Examples

Following is a series of examples that illustrate the weaknesses of the various models proposed in the literature and how our model automatically captures dependencies between program variables and system calls in the form of program invariants that produce more precise control flow graphs and limit the attacker's ability to launch mimicry and non-control-data attacks.

### 3.1 Abstract State Graphs for Intrusion Detection

Figure 3 describes an example without any function calls, studied in [13]. The predicate $strncmp(user, \texttt{"admin"}, 5)$ determines which branch of the `if` statement is executed. Only two system call sequences are possible: $\langle sys\_1, sys\_3\rangle$ if the predicate evaluates to $true$ and $\langle sys\_2, sys\_4\rangle$ otherwise. Since recent models proposed in the literature (e.g., Dyck models [13, 16]) do not track automatically the values of branch predicates, they will allow the following four sequences: $\langle sys\_1, sys\_3\rangle$, $\langle sys\_1, sys\_4\rangle$, $\langle sys\_2, sys\_3\rangle$, and $\langle sys\_2, sys\_4\rangle$ (see Figure 3 (i)). An attack on these models uses a large `someinput` in `strcpy` to overflow `str` and change the value `"guest"` of `user` to `"admin"`. Then the illegal sequences $\langle sys\_1, sys\_4\rangle$ or $\langle sys\_2, sys\_3\rangle$ are executed without any detection. Figure 3 (ii) illustrates that the guarded model that we propose is simply the abstract state graph of the program built using the predicate $(user = admin)$ appearing in the conditional branches. Our model will first accept the system call $sys\_1$ because the guard $P \equiv (user = admin)$ is true, but will not accept system call $sys\_4$, because $sys\_4$ is allowed only if the negation of $P$ holds. The illegal sequences will violate the invariants $P$ and $\neg P$ that respectively guard system calls $sys\_3$ and $sys\_4$. The graph in Figure 3 (ii) is a simplification of the reachability graph produced by BLAST. BLAST does not always recognize that some expressions are equal. This is the case of both `if` conditions. Our preprocessing of the application's code allows us to help BLAST recognize such equalities. If we uncomment the program line `user` $=$ `"guest"`, our model is precise enough to allow only the sequences $\langle sys\_1, sys\_3\rangle$ and $\langle sys\_2, sys\_3\rangle$ (see Figure 3 (iii)).

```
char *str, *user;
...
if (strncmp (user, "admin", 5)){
  sys_1 ();}
else {
  sys_2 ();}
/* user = ''gest''; */
strcpy (str, someinput);
if (strncmp (user, "admin", 5)){
  sys_3 ();}
else {
  sys_4 ();}
}
```

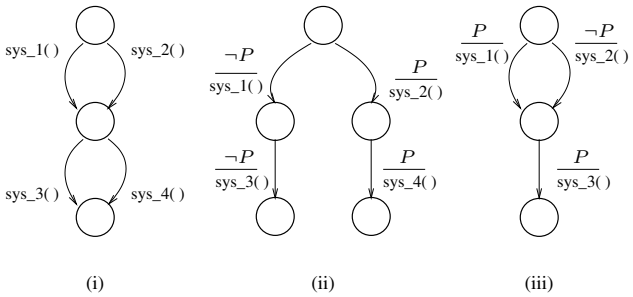


**Figure 3.** A Simple Program and its Abstract State Graph where $P \equiv (user == \texttt{"admin"})$ and $!P \equiv \neg P$.

### 3.2 Invariants and Model Precision

Figure 5 illustrates an example of a program with a call to a function `ctrl` that does not contain system calls. The behavior of this function affects the sequences of system calls that the program can generate. Existing models abstract away in their control-flow-based-models all function calls that do not contain system calls. The program prompts the user to enter two integers `i` and `j` and

a shell command. If the function ctrl returns 0, the command is executed, otherwise the command's script is printed.

```
int ctrl(int i, int j) {
  int x,y;
  x = i;
  y = j;
  while (x <> 0) {
    x = x - 1;
    y = y - 1;
    }
  if (i==j)
    return y;
  else
    return x - y;
}
```

**Figure 4.** A simple Example of a Function with no System Calls

Function ctrl, described in Figure 4, takes two integers as input and returns an integer value. Determining the return value of the function and its relationship to the arguments `i` and `j` is crucial in building a precise model of the program. The absence of any invariant on the return value of function ctrl leads to a nondeterministic model that will allow the nondeterministic choice between the execution of either the `if` or `else` for any inputs of the program.

```
void main (void) {
 char input[32];
 gets(input);
 gets(i);
 gets(j);
 if (ctrl(i,j) == 0) {
   setreuid(42,-1);
   syslog(1,"Execing file");
   execve(input,0,0);
 }  else {
   struct stat buf;
   syslog(1, "Echoing file");
   stat(input+2, &buf);
   int fd = open(input,O_RDONLY);
   void *filedata =
     mmap(0,buf.st_size,PROT_READ,MAP_PRIVATE,fd,0);
   write(1,filedata,buf.st_size);
 }
}
```

**Figure 5.** A Example of a Program with a Call to a Function with no System Calls

### 3.3 Invariant Generation and Assertion Checking

Figure 6 describes the result of running Daikon on the program with respect to the function `main`. That is, all of the likely invariants that Daikon discovers about the function `main` and all functions and system calls within the body of the function `main`, and by running the program and providing respectively the inputs "/csl/bin/library", 3, and 3, as the respective values for the variables `input`, `i`, and `j`.

Daikon captures the fact that arguments of functions and system calls are constants. An example of that is `__ruid == 42` and `__euid == -1` for the first and second arguments of `setruid`. Daikon can also capture the fact that the arguments to a function or a system call are equal. An example of that is the equality `__argv == __envp` between the second and third argument of `execve`. But most importantly, Daikon can establish equalities between return values and arguments of function calls. With our preprocessing capability, we are able to use Daikon to generate those relationships between return values and arguments of system calls as well. An example of such relationships is the quality between the return value

```
Daikon version 4.2.12, released December 1, 2006;
Processing trace data; reading 1 dtrace file:
[2:37:50 PM]: Finished reading example.dtrace
==============================================
..setreuid():::ENTER
__ruid == 42
__euid == -1
==============================================
..execve():::ENTER
__argv == __envp
__path == "/csl/bin/library"
__argv == null
==============================================
..gets():::EXIT
return == "/csl/bin/library"
Exiting Daikon.
```

**Figure 6.** Output of Running Daikon on the Program in Figure 5

```
Daikon version 4.2.12, released December 1, 2006;
Processing trace data; reading 1 dtrace file:
[8:19:21 PM]: Finished reading new.dtrace
==============================================
..ctrl():::ENTER
i == j
i == 3
==============================================
....
x == y
x one of { 0, 1, 2 }
==============================================
...
return == 0
Exiting Daikon.
```

**Figure 7.** Output of Daikon for Function `ctrl`

of `gets` and the first argument of `execve`. Such simple relationships are easily proved by BLAST as invariants that hold for all runs of the program.

Most importantly, Daikon generates a key likely invariant for function `ctrl`. Figure 7 describes the output of Daikon for function `ctrl` in the form of the assertion `i==j`. Daikon also generates the loop invariant `x==y` for this particular run as well. Since `i` and `j` are input variables to the function, and `x` and `y` are local variables, we combine the two assertions into `i==j implies x=y`. This likely invariant is trivially proved by BLAST.

### 3.4 Mimicry and Non-Control-Data Attacks

Figure 8 describes the Dyck model for the example in Figure 5. The first observation is that the model does not take into account system call arguments and therefor introduce a great deal of nondeterminism. The second observation is that the model ignores functions that do not perform any system calls but might influence the behavior of the application.

The Dyck model is a pushdown automaton where null system calls in the form of `push` and `pop` are introduced before and after each system call and used to ensure that system calls are executed in the order defined by the control flow graph of the program. In [17], the following mimicry attack against the Dyck model of Figure 5 is generated to illustrate the weakness of state-of-the-art models proposed in the literature:

```
read(0);
read(0);
read(0);
setreuid(0, 0);
write(0);
execve("/bin/sh");
```

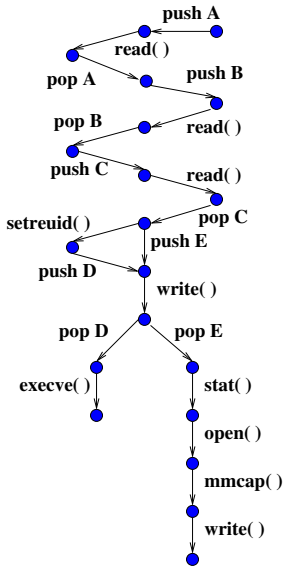


**Figure 8.** Dyck Model for the Example in Figure 5

A mimicry attack is a sequence of malicious system calls that is allowed by the model but not by the application. A mimicry attack exploits the nondeterminism introduced in the model. This nondeterminism is the result of the approximation process during the construction of the Dyck and similar models. In contrast, our guarded model of the same program described in Figure 9 includes key invariants generated by Daikon and validated by BLAST and represents a semantically richer representation of the program's behavior. The combination of likely invariant generation and assertion checking is the key to the success of our approach.

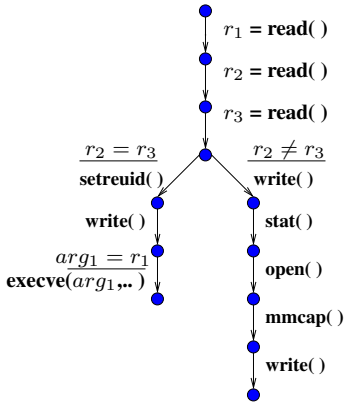


**Figure 9.** Guarded Model for the Example in Figure 5

While our extension of Daikon can generate a large number of invariants for a given program, it is possible to provide Daikon with a list of program control points and generate invariants only at those control locations. It is also possible to select a subset of local and global variables to reduce further the number of invariants. With these restrictions, we are guaranteed to capture automatically all system call arguments that are constants, and all equalities between system call return values and system calls arguments. This particular form of likely invariants is much simpler to prove with any assertion checker than arbitrary assertions about program variables. Furthermore, these likely invariants are exactly the kind of invariants that prevent non-control-data attacks. Non-control-data attacks exploit the nondeterminism that is inherent to the applications code it self. Figure 10 illustrate a non-control-data attack on a simple program where `ch` is an input character that determines what system call is executed in the `switch` statement. Between the time where the user inputs a value for `ch` and the time it is used to decide which branch of the `switch` statement to execute, the user inputs a string `input` that might override the value of `ch`.

```
int main(void) {
  char ch;
  char input[32];
   ...
  ch = getchar();
   ...
  gets(input);
   ...
  switch(ch) {
    case 'u': sys1 ;
    case 'e': sys2 ;
    case 'r': sys3 ;
    }
 }
```

**Figure 10.** Example of Code vulnerable to Non-control-data Attack

Daikon automatically generates for this kind of example, the relationship between the return value of `getchar` and the variable `ch` used in the `switch statement`.

## 4. Guarded Models

In the previous section, we illustrated how invariants generated from source code can produce guarded models that are precise enough to capture rich semantics information about the application and prevent a large number of attacks. In this section, we formally define guarded models. In particular, we show how guarded models relate to application code via predicate abstraction [19]. Our first model is the guarded model for an individual procedure. This model is the abstract state graph of the procedure. It is a much more precise version of the control flow graph of the procedure.

DEFINITION 1. ***(ASG: Abstract State Graph)*** *Let $P$ be a program, and $p_1, \cdots, p_k$ a set of arbitrary predicates over its variables. An ASG of a procedure $P$ using the predicates $p_1, \cdots, p_k$ is the* intraprocedural *control flow graph $G_P = \langle A, \top_P, \bot_P, \tau_a \rangle$ associated with the program source code of $P$ where*

- *$A$ is the finite set of abstract states such that each abstract state is a pair $(v, e)$ where $v$ is a program control location and $e$ a valuation of the predicates $p_1, \cdots, p_k$*
- *$\top_P \in V$ is the entry point of $P$.*
- *$\bot_P \subseteq V$ is the finite set of $P$'s exit points.*
- *$\tau_a \subseteq V \times V$ is a transition relation.*

The ASG of a procedure is turned into a Guarded Control Flow Graph by guarding each transition originating from an abstract state $(v, e)$ by the boolean combination of the predicates $p_1, \cdots, p_k$ given by the valuation $e$. This boolean combination is an invariant of the program at location $v$.

DEFINITION 2. ***(GCFG: Guarded Control Flow Graph)*** *A GCFG of a procedure $P$ is the intraprocedural control flow graph $G_P = \langle V, \top_P, \bot_P, \tau \rangle$ associated with the program source code of $P$ where*

- *$V$ is the finite set of program locations.*
- *$\top_P \in V$ is the entry point of $P$.*

- $\bot_P \subseteq V$ *is the finite set of* $P$*'s exit points.*
- $\tau \subseteq V \times \mathcal{G} \times V$ *is a transition relation where each transition between two control locations is guarded by an invariant in the set* $\mathcal{G}$ *of program invariants.*

Since we monitor system calls, our model abstracts away from the GCFG any instruction block that does not refer to a system call or function call to obtain a guarded call graph.

DEFINITION 3. ***(GCG: Guarded Call Graph)*** *Let* $P$ *be a procedure and let* $G_P = \langle V, \top_P, \bot_P, \tau \rangle$ *be its GCFG. A guarded call graph (GCG) of* $P$ *is a tuple* $G = \langle \mathcal{S}, \top_P, \bot_P, \tau_C, \mathcal{F}, \Sigma, \mathcal{A} \rangle$ *where*

- $\mathcal{S}$ *is a finite set of states.*
- $\mathcal{F}$ *is the finite set of function calls that appear in* $P$*.*
- $\Sigma$ *is the finite set of system calls that appear in* $P$*.*
- $\mathcal{A}$ *is a two-dimensional array associated with function call sites.*
- $\tau_C \subseteq (\mathcal{S} \times \mathcal{G} \times \mathcal{F} \times \mathcal{S}) \cup (\mathcal{S} \times \mathcal{G} \times \Sigma \times \mathcal{S})$ *is a transition relation.*

Intuitively, a state $s$ in $\mathcal{S}$ represents a set of consecutive program points than identify a basic block, that is, a straight-line piece of code without any jumps, or system calls, or function calls. Transitions are labeled by either a function call or a system call. Since we monitor only system calls, it is necessary to replace function calls by a jump to the model of the called function while retaining information about the control location to which the application should return after the call to the function is completed. By eliminating all function calls, we obtain a Guarded System Call Graph.

DEFINITION 4. ***(GSCG: Guarded System Call Graph)*** *Let* $G = \langle \mathcal{S}, \top_P, \bot_P, \tau_{SC}, \mathcal{F}, \Sigma, \mathcal{A} \rangle$ *be the GCG of a procedure* $P$*. Then* $G$ *is said to be a Guarded System Call Graph if* $\mathcal{F} = \emptyset$*. In other words,* $G$ *does not contain function calls. Furthermore, the transition relation* $\tau_{SC}$ *is defined as follows:*

- $\tau_{SC} \subseteq (\mathcal{S} \times \mathcal{G} \times \Sigma \times \mathcal{U} \times \mathcal{S})$ *is a transition relation where* $u \in \mathcal{U}$ *is an action of the form* $\mathcal{A}[F, s] := 1$ *or* $\mathcal{A}[F, s] := 0$ *that updates the array* $\mathcal{A}$*.*

Intuitively, by eliminating function calls, the transition relation refers only to system calls. However, for the purpose of maintaining control flow integrity, it is crucial to maintain information related to function call sites and return sites. This is achieved by augmenting the transition relation with assignments of the form $\mathcal{A}[F, s] := 1$ to indicate that a function $F$ has been called at control location $s$, and assignments of the form $\mathcal{A}[F, s] := 0$ to indicate a return from a call to the function $F$. Therefore, for each function $F$ and each program control point $s$ where the function is invoked, the boolean value $\mathcal{A}[F, s]$ indicates whether the program is executing $F$ or not.

Similarly to other approaches in the literature, we show how to extend the previous model with a stack, allowing us to build models of programs with recursive functions.

DEFINITION 5. ***E_GSCG: Extended GSCG***
*Let* $G = \langle \mathcal{S}, \top_P, \bot_P, \tau_C, \mathcal{F}, \Sigma, \mathcal{A} \rangle$ *be the guarded system call graph (GSCG) of a procedure* $P$*. The E_GSCG* $E_G$ *extends. We define the transition relation* $\tau_E$ *as follow:*

$$\tau_E \subseteq \cup(\mathcal{S} \times \mathcal{G} \times \Sigma \times U_E \times \mathcal{S})$$

*where* $U_E$ *is the finite set of actions where each action* $a_i$ *is of the form* $\mathsf{push}(\gamma)$*,* $\mathsf{pop}(\gamma)$*, an update action* $u \in \mathcal{U}$*, or the empty action* $\epsilon$*.*

The semantics of the transition relation $\tau_E$ identifies valid sequences of system calls within a program execution. We associate to each function $F$, the set $scope(F)$ of all functions that appear in the call graph of $F$.

The semantics of each transition of the form

$$(s_i, g, \sigma_i, a_i, s_{i+1})$$

is given bellow:

1. *invoking a system call:* a transition corresponding to a system call $\sigma_i$ is accepted if and only if the invariant $g_i$ guarding the system call is true.
2. *call to a function:* if $s_i$ is the entry point to a function, then $\sigma_i$ is the first system call executed in $F$. The call is allowed if the invariant $g_i$ is true, and the corresponding action $a_i$ is the assignment $\mathcal{A}[F, s_i] := 1$ which indicates that the program in executing $F$ from $s_i$.
3. *return from a function call:* if $s_{i+1}$ is the exit point of a function, then $\sigma_i$ is the last system call executed in $F$. The call is allowed if the invariant $g_i$ is true, and if the function $F$ has been invoked from $s_{i+1}$, that is, $\mathcal{A}[F, s_i] == s_{i+1}$. The corresponding action $a_i$ is the assignment $\mathcal{A}[F, s_i] := 0$ which indicates that the program is returning from executing $F$. In addition to checking that $g_i$ is true, we must check that:
$$\forall\, G \in scope(F).\, \forall s.\mathcal{A}[G, s] == 0$$
indicating that all function calls inside $F$ have resulted in a return to the intended return locations.
4. *call to and return from a recursive function call:* if $s_i$ is the entry point to a recursive function, then in addition to the action described above, the pair $(F, s_i)$ is pushed onto the stack. If $s_{i+1}$ is the exit point of a nonrecursive function, then it is necessary to check that the top of the stack contains the pair $(F, s_i)$.

Models that use a pushdown System to monitor sequences of system calls belong to the family of *visibly pushdown automata* [5]. Intuitively, the stack alphabet is a finite set of return addresses specifying where the program execution should resume after the call to a function. The visibly pushdown automaton pushes onto the stack an element when it reads a function call, and it pops the stack only when returning, and does not use the stack when it reads allowed system calls. Using a visibly pushdown automaton requires code instrumentation to trigger the push and pop operations based on the observed inputs to the automaton of the application. Our model is more efficient at monitoring time because we already encoded stack activity using the array $\mathcal{A}$. Our model achieves stack determinism and can be simulated by a Dyck model which generates more overhead during monitoring and which cannot deal with recursive functions nor with the presence of circuit in function calls. In our model, we show that recursive functions can be handled without generating extra system calls. We succeed in building a model which does not need to monitor the program counter, unlike [32], nor does it introduce extra null system calls, unlike the Dyck model [13, 16].

Our model shares many features of already proposed models. Steps 2 and 3 ensure context sensitivity and capture the call-return semantics of high-level programming languages like any reasonable model would. Step 4 adopts the traditional approach of adding a stack to model recursive programs. The novelty of our model resides in the use of program invariants generated by invariant generation techniques, dynamic program analysis, and assertion checking. Building our model using predicate abstraction and invariant generation allows us to derive key properties that make the model useful as a formal approach to intrusion detection.

## 5. Building Our Model

Our model construction process consists of the execution of the following steps:

1. run the Inv-Gen tool to generate a set of invariants in the form of linear relationships among program variables at exit function points for each function.
2. run the application using Daikon and generate a set of likely invariants. Likely invariants are generated at the entry and exit points of each function and at each location where a system call is invoked.
3. turn every likely invariant generated by Daikon into an assert statement in the line of code indicated by Daikon and augment the application code with these assert statements.
4. run the BLAST assertion checker and check all assert statements in the augmented application code and make use of the invariants generated by Inv-Gen as aids in the assertions checking process.
5. filter the set of likely invariants into three sets:
   - likely invariants for which BLAST can prove that the corresponding asserts hold
   - likely invariants for which BLAST generates execution traces that violate the corresponding asserts
   - likely invariants for which BLAST either runs out of memory, crashes, or returns with an inconclusive answer
6. for each procedure, generate, using BLAST, the control flow graph of each function where each node is decorated by the set of predicates that hold at that node location.

Step 6 effectively produces the abstract state graph of a procedure using the predicates collected by Daikon and Inv-Gen, and the predicates appearing in conditional branches and assignments in the source code of the function. We minimize the abstract state graph by applying a minimization algorithm based on the observational equivalence proposed by Milner [25] and implemented in [14], that is, by retaining only system calls and function calls while preserving the invariants that hold true before each system call and function call and at the return location of each function call as well. This minimization guarantees that the sequences of system calls accepted by the original model and the minimized model are the same. This allows us to obtain the guarded call graph for each function. The call graph of each function is turned into a guarded call graph by guarding each system call and function call by the invariant that holds at the control location where the call is invoked. The minimization step does not lose any information about the collected invariants at the control nodes that have been collapsed since the construction of the abstract state graph propagates every invariant to every node location where it holds, in particular at control locations where a function or a system call is invoked.

While obtaining the guarded call graph for each function is simple once an abstract state graph is built, combining the guarded call graphs of all functions to build a global call graph can be done in different ways. In [18] inlining is used whenever a function call is made. The problem in this approach is that the same call graph is duplicated at each call site of a function without further analysis. In our approach, invariants that are true at the function call site are propagated inside the function to obtain a more refined control structure. That is, for every function call site, we might inline a different, more precise, model of the function. In the case where the propagation of the predicates does not produce a more precise model of the function, we just transfer the control to the function's guarded call graph. Since we need to remember the call site to which the execution should return to after the function call, we update the array $\mathcal{A}$ when a particular call at a particular control location is invoked. We also update the array $\mathcal{A}$ at function return sites. Updating the array $\mathcal{A}$ amounts to combining the guarded call graph of all functions and producing a global guarded system call graph when there are no recursive functions, or an extended guarded system call otherwise. This is done by eliminating the function calls and deciding whether or not to apply inlining. The following algorithm describes our process for producing the final global model, starting from the guarded call graph (GCG) of the main function. Let $T = (s, g, F, s') \in \tau_C$, a transition where a function call to $F$ is invoked at site $s$. If $F$ does not invoke any function calls and $F$ contains at least one system call, we connect $s$ to the GCG of $F$ as follows: we merge the states $s$ and $\top_F$, and we add the transition

$$t_k = (s, g_k, \sigma_k, \mathcal{A}[F, s] := 1, s_k)$$

for every first system call $\sigma_k$ encountered in every possible control path from $\top_F$. Also, for every state $s_e$ from which $\bot_F$ is reached in one step $(s_e, g, \sigma, \bot_F)$, we replace such transition by

$$(s_e, g \wedge I, \sigma, \mathcal{A}[F, s'] := 0, s')$$

where $I$ is the predicate

$$\mathcal{A}[F, s'] == 1 \wedge \forall\, G\ \in scope(F).\, \forall s.\mathcal{A}[G, s] == 0$$

If $F$ does not contain system calls, we merge $s$ and $s'$, and we strengthen the guards of all outgoing transitions of $s'$ with the disjunction of invariants that hold at the exit points of $F$ and we remove $T$. If $F$ invokes function calls, then we apply the same algorithm to every function called by $F$. If $F$ is a recursive function, the assignments to $\mathcal{A}$ are replaced by the appropriate push or pop actions.

BLAST can be invoked with an option that can compute a global abstract state graph for the whole application code and with an option that computes the abstract state graph for each individual function. When the application code is larger than a few hundred lines of code, it is recommended to avoid building the global abstract state graph since it requires more space and time for the generation process to be completed. Due to the limitations of BLAST, we use the ncc compiler [3] to generate the call graph of the entire program or a particular function, and use the analysis information generate by ncc to indicate the location for which Daikon should generate invariants. Invariants proved by BLAST are then attached to the corresponding nodes in the call graph.

## 6. Properties of Guarded Models

Our model allows fewer attacks than, for instance, a Dyck model [16]. Recall that Dyck model instrumentation involves inserting distinct pre-calls and post-calls at each function call site. This ensures determinism in stack operations as each call site is identified by its pre-call. A Dyck stack records the pre-call and a path is deemed feasible if each return leads to a post-call, matching with the last pre-call at the top of the Dyck stack.

THEOREM 6.1. ***[Expressivity]***
*Let $P$ be a program, and Pred_GSCG and E_GSCG be respectively the guarded system call and the extended system call graph for $P$.*

1. *If $P$ contains no recursive functions or no circuit in the sequence of functions calls, then any Pred_GSCG of $P$ can be simulated by the Dyck model of $P$.*
2. *If $P$ contains, recursive functions then any E_GSCG of $P$ can be simulated by the Dyck model of $P$.*

PROOF 1. *In the absence of any data flow analysis, the Dyck model introduces for each system call a Pre- and Post-call. The Dyck model does not allow any path that is not allowed by the control*

| Reference | Program | Attacks |
|---|---|---|
| [10] | ghttpd | overwrite filename |
| [10] | wu-ftpd | overwrites userid |
| [2] | rm.c | race condition overwrites path name |
| [27, 31] | ssh | overwrites authenticated flag to non-zero |
| [10] | Netkit Telnetd | overwrites execve's arguments |

**Figure 11.** Attacks on Real-world Applications Detected by Invariant Violation.

*flow graph of the program, but does not eliminate any paths from the control flow graph either. Guarded models are more precises that the control flow graph of the program. Therefore, to every path in a guarded model, corresponds a path in the Dyck model.*

Theorem 6.1 shows that our Model is always more precise than the Dyck model. Also, the level of stack determinism is reached without the overhead of instrumenting the binary with null system calls. We also do not need to monitor the program counter, as it is statically taken into account by the abstract state graph. By using updates the the array $\mathcal{A}$, we eliminate the need to instrument the source or the binary code with additional system calls at each system call site. If $P$ does not contain recursive functions or circuits in a sequence of function calls, then a stack is not mandatory (even if a function is called at different sites).

In some cases, our model can detect application logic bugs efficiently.

THEOREM 6.2. ***[Exact Model]***
*Let $P$ be a program, and Pred_GSCG the guarded system call graph of $P$. If Pred_GSCG is built from deterministic abstract state graphs, then the sequences of system calls accepted by Pred_GSCG are exactly those accepted by the $P$.*

PROOF 2. *The proof is trivial in the sense that in the case where the abstract state graph is deterministic, the model is not an overapproximation of the application behavior, but a finite an exact faithful abstraction of the application. This means that every sequence in the model is a sequence of the application and vice versa.*

In addition to its robustness against attacks to which the Dyck model is vulnerable, our model can tackle non-control-data attacks that are beyond the reach of the Dyck model. In Figure 11 we refer to published attacks that are based on non-control-flow manipulation described briefly in the third column. All these attacks do not modify the sequences of system calls generated by the execution of the program (second column) but violate a trivial invariant that is generated and propagated along the control structure of the program. Notice that three attacks listed in [10] are detected by data flow integrity mechanisms implemented in [9] and that connect a variable value used in an instruction to the value it was set to in a previous instruction. These are exactly the kind of dependencies Daikon generates for these examples and those are exactly the invariants that BLAST manages to trivially prove true since no program instruction will affect those variables between the time they are set and the time they are used. This has been confirmed by our experiments with the three attacks.

To illustrate further the effectiveness of our model, we prove that our automatically generated model detects any attack generated automatically by the techniques described in [24].

THEOREM 6.3. ***[Detection]***
*Let Pred_GSCG be the guarded system call graph of a program $P$. Any attack on $P$ automatically generated by the framework described in [24] will be detected by Pred_GSCG.*

PROOF 3. *The technique described in [24] consists in finding a sequence $sc_1, sc_2, \cdots, sc_n$ of system calls that are accepted by an intrusion detection model such as a Dyck model. The attacker hijacks the application and executes the following sequence:*

$$mc_1, sc_1, r_1, mc_2, sc_2, r_2 \cdots, mc_n, sc_n, r_n$$

*That is, it executes a malicious code $mc_i$ before a system call $sc_i$ in which arguments to the system call have been overwritten with values defined by the attacker, executes the system call $sc_i$ with the attacker's arguments, and then finds a set of memory locations and corresponding values so that if these memory locations are overwritten in a sequence $r_i$ of instructions, the program's flow of execution will return to a legitimate program control point to conform to the control flow integrity enforced by the model. The sequence is repeated for each system call $sc_i$. The Pred_GSCG guards each system call $sc_i$ with an invariant $g_i$. Therefore, there exists some $i$ for which executing the sequence $mc_i, sc_i$ that overrides system call arguments with attacker's values instead of the sequence $lc_i, sc_i$ consisting of the legitimate block of instruction $lc_i$ preceding the system call $sc_i$ will not follow the program execution paths from which $g_i$ has been generated and propagated and will violate $g_i$ that summarizes the data flow in $lc_i$.*

For the attack to succeed, the attacker has to execute all of the sequence $mc_1, sc_1, r_1, mc_2, sc_2, r_2 \cdots, mc_n, sc_n, r_n$. For our model to prevent the attack, it is sufficient that one invariant $g_i$ be violated.

Current state-of-the-art models introduce nondeterminism that is not present in the applications code and therefore are subject to mimicry attacks. In [17], Giffin and al. presented a simple and efficient way of checking whether a model is subject to a mimicry attack and can generate a sequence of system calls and their arguments that defeats a given model if it is vulnerable. Such a technique is based on model checking the model it self against a specification of an OS policy expressed in a simple form. In the case of guarded models, we make such a process even simpler. Guarded models are built using predicate abstraction which has nice theoretical properties of great practical usefulness. The nondeterminism created by the abstraction process determines exactly where mimicry attacks are possible. A mimicry attack is every sequence of system calls that reaches a particular OS configuration and that has at least a nondeterministic transition.

THEOREM 6.4. ***[Evaluation]***
*Every sequence in a guarded model containing a nondeterministic transition created by predicate abstraction can potentially be exploited by a mimicry attack.*

Theorem 6.4 allows us to devise a simple procedure for evaluating guarded models against mimicry attacks and reduces the model checking process of evaluating a given model.

## 7. Experimentation

Using our prototype implementation, we ran some experiments to measure the model construction and monitoring overhead for small and medium size applications. For monitoring purposes, we used the etrace framework [22] that allows the interception of system calls and provides an API for extracting system calls arguments. The overhead of checking whether system calls arguments satisfy or not some invariant is negligible since most of the overhead in the monitoring process is caused by the interception of the system calls. Therefore, the number of invariants and their size does not induce additional performance penalties. The etrace framework induces

an overhead of between 2% and 30% for the applications that we studied such as gzip, httpd, ssh, and snort. The construction of the models was the most computationally intensive phase of the experiments. However, it is important to notice that model construction is a one-time cost to endure and once the model has been built, the practicality of our approach depends solely on the monitoring overhead.

We executed the steps described in Section 5 to build our models. We restricted the runs of each application to collect no more than 20 MB of execution traces to be processed by Daikon to generate likely invariants.

| Application | Daikon/ Inv-Gen | etrace overhead | model overhead |
|---|---|---|---|
| `ghttpd` | 16 | 5% | 1.1% |
| `gzip` | 106 | 2.2% | 0.9% |
| `ssh` | 374 | 13% | 1.9% |
| `snort` | 2130 | 28.5% | 3.2% |

**Figure 12.** Number of Invariants for Some Sample Applications and the Corresponding Overheads Associated with System Calls Interposition and Invariant Checking

Figure 12 shows our experimental results. For each application, Daikon takes between 30 and 60 seconds to processes each MB of run traces. Once those traces are processed, likely invariants are generated. For each application, the second column gives the total number of likely invariants generated by Daikon and the invariants generated by Inv-Gen. The percentage of the likely invariants that have been generated by Daikon that are proven true by BLAST is between 5% and 10%. This means that most likely invariants are proved to be violated. This means that if those invariants have been used to build an underapproximation model of the application, then that model will potentially generate a huge number of false alarms. The speed by which BLAST discharges each likely invariant depends largely on the size of the application's code. While it takes about 2 seconds in the example of small applications such as ghttpd, it may take up to 20 minutes for some likely invariants in the case of snort. However, it took BLAST far less than that to disprove more than 90% of the invariants. That is, BLAST takes more time to prove that a likely invariant holds than to generate a counter-example for a likely invariant that does not hold. This performance indicates that the tools that we use can handle large number of invariants and therefore large models, but the construction time might takes several days for a significant application. However, it is useful to remember that this is a one-time cost in the process of securing such application. More experiments are needed to devise a strategy by which invariants are generated only for some system calls and not all system calls. This is possible since attacks usually proceed in sequences of system calls and therefore, as illustrated in the previous section, one needs to enforce application invariants only at some system calls. This will speed up the model construction process and will reduce the monitoring overhead. In the case of snort, it was not possible to generate the global abstract state graph, but we were able to generate a global state graph for one particular important part of snort's code, that is, the functions that process network packets. Those are the most vulnerable functions since an intrusion is the result of processing a malicious packet. The model has 49 states and 657 transitions. Only 23 invariants are used in the model. Those 23 invariants are enough to monitor the application when it processes any packet on the network. In any given application, vulnerabilities are typically localized to a particular library, or a set of functions. While in general, it might not be possible to know what part of the application's code is vulnerable, it is often the case possible to determine which part of the code is critical and should be monitored. Our experiment with the snort vulnerability [1] illustrate this point. The code of snort could be divided into two basic blocs: an initialization block and a packet processing and logging bloc. A buffer overflow is likely to be triggered by a well crafted packet. Therefore, it is not necessary to build a model of the initialization phase that consists largely of configuring the network interface and reading snort rules.

## 8. Conclusion

We have presented guarded models, a novel model of application behavior for intrusion detection systems. Our model is built in a fully automatic way and provides an improvement over state-of-the art models [16, 8] due to its tight coupling of control flow and data flow analysis. Our model presents strong properties such as the automatic detection of mimicry attacks [33] to which state-of-the-art models are all subject. Furthermore, it can handle non-control-data attacks [10] that are often detected because of a violation of the many invariants that we generate. Because guarded models are built using predicate abstraction, their evaluation against mimicry attacks is largely simplified compared to other models, because the nondeterminism introduced by predicate abstraction characterizes precisely where in the code the model loses precision in modeling that application's behavior.

Our approach is complementary to many interesting and efficient approaches to securing software. Our invariants could easily be added as guards to a Dyck model to make it a more precise model. In approaches [8] where an application model is learned from a training period, a large number of facts are generated about the program from the various runs. These approaches have no way of evaluating which facts are useful. In particular, if a learned fact is violated later during monitoring, it is not possible to know if the violation corresponds to a legitimate execution of the program or to an attack. Our combination of Daikon and BLAST allows us to determine precisely which likely invariants are always true and which likely invariants are not. This means that we have a proof that some assertions observed during the training period will eventually be violated in a legitimate execution of the program. Therefore, we can provide to any learning approach the likely invariants for which BLAST did not succeed in proving or disproving their validity at particular locations of the program. Our model can benefit from recent efficient implementation of control flow integrity [4]. The combination of control flow integrity and memory protection [11] will provide guarantees that all our invariant checks are added to the application's binary and that they are protected from being overwritten. Recent data flow integrity mechanisms [9] detect non-control-data attacks by enforcing simple safety properties that reflect memory safety policies. We believe that by adding to such policies our invariants will provide more security. It is noted in [10] that mounting non-control-data attacks requires knowledge of the semantics of the application. Mechanisms such as those employed in [9] do not include in their memory policy semantics information about the application. We believe a combination with our invariants will produce a much more powerful detection mechanism. Invariant generation techniques have been investigated for over three decades. Our framework is extensible in the sense that any invariant generation technique can be exploited, in particular, any techniques that deal with multithreaded programs [7, 28], programs that manipulate dynamic objects, such as shape analysis [34], and programs involving complex relationships among program variables, such as nonlinear invariants [30]. Our framework is an open architecture to which different invariant generation tools can be plugged in. The maturity of assertion checking tools such as BLAST and SLAM [6] for C programs shows that our approach is scalable.

More information on this work is available at
`http://www.csl.sri.com/users/saidi/program_analysis`

## Acknowledgments

We thank the anonymous referees for their useful comments. The design and implementation of guarded models benefited from the help of Rachid Rebiha and his many useful comments and suggestions while being a summer visitor at SRI. Dr. John Rushby and Mr. Rance Delong were a source of many stimulating discussions about this work. This research was sponsored by NSF under contract number SA4102-10097PG/CCR-0325274 and by DARPA under contract number FA8750-06-C-0182.

## References

[1] 2007-02-19 sourcefire advisory: Vulnerability in snort dce/rpc preprocessor, versions 2.6.1, 2.6.1.1, 2.6.1.2, and 2.7.0 beta 1. http://www.snort.org/docs/advisory-2007-02-19.html.

[2] The common vulnerabilities and exposures (cve) web site. http://cve.mitre.org/.

[3] Ncc: a compiler that produces program analysis information. http://students.ceid.upatras.gr/ sxanth/ncc/.

[4] Abadi, Budiu, Erlingsson, and Ligatti. Control-flow integrity: Principles, implementations, and applications. In *SIGSAC: 12th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2005.

[5] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC: ACM Symposium on Theory of Computing (STOC)*, Aug 2004.

[6] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. pages 1–3, Jan. 2002.

[7] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335. Springer Verlag, 1996.

[8] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2006.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[11] O. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Usenix Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[13] 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*, page 194. IEEE Computer Society, 2004.

[14] J.-C. Fernandez, H. Garavel, AlainKerbrat, R. Mateescu, L. Mounier, and MihaelaSighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, Aug. 1996.

[15] S. Forrest and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[16] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In A. Valdes and D. Zamboni, editors, *Recent Advances in Intrusion Detection (RAID)*, volume 3858 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2005.

[17] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In D. Zamboni and C. Krügel, editors, *Recent Advances in Intrusion Detection, 9th International Symposium, RAID 2006, Hamburg, Germany, September 20-22, 2006, Proceedings*, volume 4219 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2006.

[18] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *IEEE Symposium on Security and Privacy*, pages 18–31. IEEE Computer Society, 2005.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer Verlag.

[20] S. Gulwani and A. Tiwari. Combining abstract interpreters. In T. Ball, editor, *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2006*, pages 376–386, 2006.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[22] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*. The Internet Society, 2000.

[23] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2005.

[24] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[25] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[26] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[27] K. Pekka and L. Kalle. Ssh1 remote root exploit., 2002. http://www.hut.fi/ kalyytik/hacker/ssh-crc32-exploit_Korpinen_Lyytikainen.

[28] H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101, Cocoa Beach, FL, Oct. 1999. IEEE Computer Society Press.

[29] H. Saïdi. Model-checking guided abstraction and analysis. In *7th International Static Analysis Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 377–396. Springer Verlag, June 2000.

[30] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using groebner bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 318–329, New York, NY, USA, 2004. ACM Press.

[31] P. Starzetz. Crc32 sshd vulnerability analysis. . http://packetstormsecurity.org/0102-exploits/ssh1.crc32.txt.

[32] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[33] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In R. Sandhu, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, Nov. 2002. ACM Press.

[34] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In D. A. Watt, editor, *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings*, volume 1781 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.