Rapid Prototyping of Intrusion Detection Systems

Fabrice Kordon and Jean-Baptiste Voron
Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
Fabrice.Kordon@lip6.fr, Jean-Baptiste.Voron@lip6.fr

Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019, USA
iftode@cs.rutgers.edu

Abstract

Designing security softwares that evolve as quickly as threats is a truthful challenge. In addition, current software becomes increasingly more complex and difficult to handle even for security experts. Intrusion Detection Softwares (IDS) represent a solution that can alleviate these concerns.

This paper proposes a framework to automatically build an effective online IDS which can check if the program's expected behavior is respected during the execution. The proposed framework extracts relevant information from the program's source code to build a dedicated IDS. We use the GCC compiler to produce the structure of our behavior's model and ensure the IDS is correct. Thanks to Petri nets, our framework allows program offline monitoring and simplifies the online monitoring development.

1 Introduction

Intrusion Detection is defined as the monitoring and analysis of events occurring in a computer system or network, in order to detect signs of intrusions. Consequently, Intrusion Detection Systems (IDS) are software or hardware products that automate this monitoring and analysis process [2].

An IDS could be considered either offline or online. In the first case, IDS is used on logged data to detect if an intrusion has happened. The second option, online detection, allows to detect intrusions with very short delays or before they happen.

To be efficient, an IDS must be adapted to a given application to take into consideration its specificities. Thus, a developer must design the right IDS for a given system. A typical example is the monitoring of system calls. The associated IDS must consider the application profile and reject unusual system calls sequences.

So far, such a process is performed "manually" by experts. This is a problem because such a process takes time when IDS must adapt rapidly to detect new intrusion techniques exploiting new vulnerabilities.

The objective of this paper is to propose a technique that automatically generates IDS dedicated to a program. This work sketches the process that extracts relevant information from the source code and uses it for both online monitoring and offline monitoring. Our technique uses Petri-Nets [11] and is plugged on the GCC front-end ; it is thus relevant for several programming languages like C, Java, C++.

The paper is organized as follow. Section 2 briefly presents the domain and usual problems concerning it. Section 3 describes our method to extract and build a model using Petri nets. Section 4 presents experimental results and compare our approach with others.

2 Motivation and Related Work

Beginning in 1980 with James Anderson's paper [1], the notion of intrusion detection was born. Since then, several proposals have been made and IDS research has clearly progressed. This part describes more precisely IDS' functions and then raises some major issues in their construction.

2.1 What is an IDS

Considering the detection methods that are used, IDS could be broadly divided into two families :

- *misuse detection* uses signatures to represent attacks scenarios then looks for patterns that matches an attack signature during the execution. This method is very efficient for known attacks but often needs a human expert to express new signatures or refine old ones.
- *anomaly detection* uses a comparison between expected behavior and observed behavior during the program's execution.

There are two ways to detect intrusions in anomaly-based systems. In the first one, the IDS learns the monitored program's behavior by looking at many executions of it. Thus, after a *learning period*, the IDS is able to distinguish normal behaviors (those observed during the training phase) from others. IDS using this technique are either rule-based [14], statistic-based [6] or time-based [13, 18]. Finally, Forrest *et al.* proposes a process-based approach from a description of computer immunology [8] and uses sequences of system calls to handle the program's behavior.

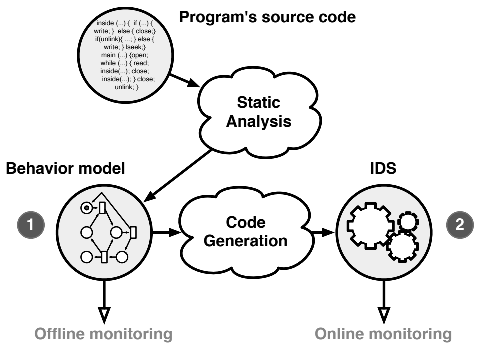
In the second approach, an external actor extracts the *expected behavior* from the system to feed the IDS. This operation takes place before any execution of the monitored program. The actor may use security specifications either expressed by means of a dedicated language [16] or directly issued from the program's specifications [5]. More recently, *static analysis* has been introduced to automatically derive a behavior's model from source [17] or binary [10] code.

[9] proposes an IDS classification based on how the *expected behavior* is computed. So-called "black box" and "gray box" families learn the behavior from repeated executions of the program. On the contrary, the "white box" family computes expected behavior from the program's structure and code only.

Consequently, the "white box" approach allows developers to evaluate the program before its execution. This first step, called *offline-monitoring*, leads to software verification techniques, also used in the context of program verification, to perform model checking directly on programs [12].

On the contrary, *online monitoring* takes place during the execution. The IDS runs as an independent entity and ensures that the program's execution remains in the expected behavior. In others terms, at any moment, the state of the program must correspond to an existing and planned behavior representation's state. To do so, the monitor has to be able to capture the state of a program, compare it to the expected one, and raise a response according to the validity of the state.

This paper focuses on online monitoring based on the "white box" approach. The first step consists to build a model describing the expected behavior thanks to static analysis (step 1 in figure 1). Then, the dedicated IDS is automatically generated (step 2 figure 1).



```
inside (...) {  if (...) {
write; }  else { close;}
if(unlink){ ...; } else {
write; } lseek;}
main (...) {open;
while (...) { read;
inside(...); close;
inside(...); } close;
unlink; }
```

Figure 1. Two steps in IDS Prototyping

2.2 Problems in IDS construction

There are four main challenges in IDS construction: the runtime overhead, the way non-determinism is handled, the size of the model representing expected behavior, and the extraction of the expected behavior.

Runtime overhead This is the most critical point [7]. If too many verifications are done at runtime, then the program's execution is too slow. In this case, the problem is to find a trade-off between the amount of information to be checked from the model and the time required by the IDS to process this information. The accessibility of information inside the model determines the time needed to process each state of the program.

Handling non-determinism There are two types of non-determinism, either states or stacks.

State non-determinism implies that multiple paths could start from the current state. So, for each step, the system has to cover all current states's successors. Consequently, the overhead is proportional to the number of states.

Stack non-determinism deals with the evolution of the program's execution stack. For each state, every possible paths in the model have to be explored to determine the possible next states and stack contents. Handling stack non-determinism may induce an exponential overhead.

Representation of expected behavior When monitored programs' complexity grows, representation of expected behavior must fit into memory [15]. Current approaches solve this problem by using a "higher-level" view on the execution. As a result, less precise configurations are handled but they still fit in memory.

Extraction of the expected behavior It is important to automate the extraction of the monitored program's behavior. Such an operation must be automatic in order to allow a quick and costless evolution of an IDS, either when the monitored program evolves or when attacks exploit new vulnerabilities.

Finally, a developer must consider the following points issued from the main challenges in IDS construction:

Efficiency: During execution, an important point is the runtime monitoring. For some models, this runtime could exceed 40 minutes per system call [7]. For this issue, the objective is to design models with the minimum of non-determinism.

Accuracy: Models have to be as precise as possible to handle a large number of attacks and to reduce the number of false negatives. The objective is to design a model representing exactly the program's behavior. According to previous researches, parameters like *system calls*, *call stack* and *program counter information* are the minimum to satisfy the accuracy.

Scalability: The space needed to store models is targeted by this concern. Models have to store the minimum information to keep the ability to handle large programs.

3 Principle of IDS Prototyping

Our objective is to show how an IDS can be automatically produced from a program's source code. This work should serve as a basis to implement an IDS generator.

We believe that formal methods could help us to answer some problems outlined in the previous section. In the meantime, the *formal method study* has to be merged with *computer security domain*. Thus, a common behavior's representation has to be build to blend these domains. This representation must be able to describe a program's behavior and serve as inputs to formal methods. Following choices are made in this paper:

- Petri nets [11] are used to model the expected behavior.
- Only system calls are monitored.
- The program to be monitored is written in C.

3.1 Overview of the Methodology

Our framework is composed of four distinct modules (see figure 2), each of them is responsible for a specific task.

Parser module : GCC is used to parse and extract relevant information from the program. GCC 4 provides us a way to extract a rich control flow graph (CFG) containing all information require to build a complete model.

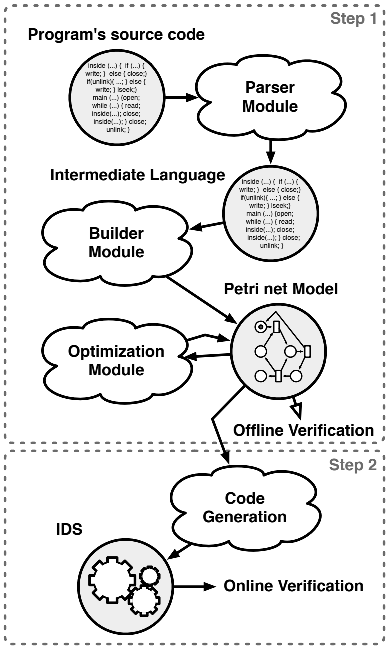


Figure 2. Several modules are necessary to transform the source code to an IDS.

Builder module : The control flow graph is then analyzed and transformed into a Petri net following basic rules (presented in section 3.3). This current work only focuses on gathering system calls' information but other data could be retrieved for other types of IDS.

Optimization module : Models provided by the builder module can be optimized. For example, a function that does not contain any system call can be discarded from the model. Then, based on known reductions that preserve properties [3], the first Petri net is reduced to an optimized one.

Code generation module : It generates the effective IDS from the model using a *token game* strategy [4]. To do so, the monitor must catch system calls and check them thanks to the model. When execution does not conform with the behavior's model, appropriate decisions are made (counter-measures, program abortion, warning sent to a security engineer).

3.2 Petri Nets

The following reasons motivate the use of Petri Nets:

- Petri nets can handle quite large models. This is particularly true for parallel or multi-threaded program. In this case, the *Colored Petri Nets* lets us factorize the behavior of all program's components into one model.

- Petri nets are used by numerous research teams all over the world and profit from large collection of dedicated tools. Offline monitoring may profits from these research works (but this is not the goal of this paper).

Intuitive definition Let us introduce Petri nets by means of a small example. The Petri net in figure 3 represents a control flow graph of a multi-threaded programs. After a calculus, threads (identified by a identity in type T) perform a write on a ressource. The program does not create more than **TR** threads.

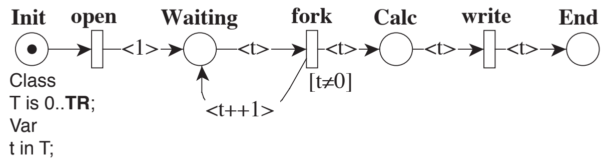


Figure 3. Example of Petri Net.

Places represent states of the system. Basically, a place contains one or more tokens. In our example, places **Calc** and **End** hold threads' identifiers. Place **Waiting** contains the next thread's identifier.

Transitions represent the evolution of the system. A transition is fired when all precondition places hold a sufficient marking and when the condition is verified. For example, transition **write** can be fired if there is one token in **Calc**. Transition **fork** is fired when the token's value is not 0. Types are circular, thus the successor of **TR** is 0.

When a transition is fired, input tokens (they may contains a value) are dropped from all precondition places to post condition places according to arcs' indications. For example, when **fork** is fired, the thread's identifier is transmitted to the place **Calc** and a new identifier is generated in place **Waiting**.

The Reachability Graph Petri nets allow to elaborate the state space of the specified system for model checking, thanks to the firing rule. The state space is usually called *reachability graph* and represents all concrete states of the system. Figure 4 presents the reachability graph for the Petri net of figure 3 with **TR** $= 2$. This state space has eight states (the initial state is represented by a double circle); it will grow following the rule: 2^{TR+1}.

Efficient storage of the expected behavior A huge state space is generated when the number of threads grows. While a normal automaton stores one information in each state, a Petri net can store many information in each of its state. Thus, Petri net is used as a generator of a normal automaton (or a state space), which is far more compact for real programs.

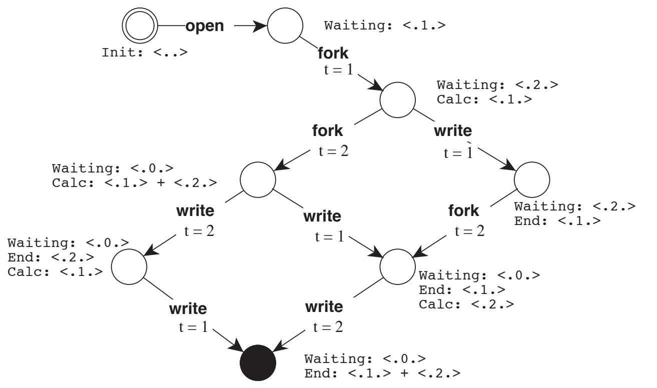


Figure 4. Reachability graph for the example

3.3 Extraction of the expected behavior

Construction of the Petri Net from the program's sources is performed in two steps.

Building the Petri Net Relevant information are extracted from the simple CFG generated by GCC. This information is used to build a skeleton that represents the program's structure. During this operation, each function is processed and its structure is represented by an assembling of blocks. Each block is related to its predecessors and successors blocks. Moreover, important instructions (like system calls or function calls) are represented in the CFG.

The parser module gather this information to build the skeleton that prefigure the expected behavior's model. In this paper, all the information about system calls is picked up. Once a block is processed, its description is replaced by the new description in terms of system calls.

The builder module uses the enriched skeleton to set up a Petri net. The construction is performed with respect to the following rules :

R1 Each function has a *Entry* state and a *Exit* state.

R2 CFG's blocks are represented by places.

R3 For each successor, in a bloc, a transition is created.

R4 A system call is represented by a transition.

R5 Each system call has a *pre* and a *post* place.

R6 A transition is included between two functions.

R7 Functions must carry their call site's identifier.

There are two types of tokens. The first one is a enumeration representing all possible system calls used in the program and the value "no-system-call" (no system call but structure information to keep). The second one serves to track functions. It is also an enumeration type with a value per function call in the CFG. Transitions between functions generates a colored token (see figure 6(c)). Its value depends on the function call site.

Finally, the *Entry* place's value of the `main` function is set with a simple non colored token. The model resulting is ready to be embedded into the IDS or to be used for offline monitoring.

Optimizing the Petri Net The use of Petri nets gives the possibility to apply reductions on the net. These reductions do not modify the meaning of the model but only its representation. In our proposal, three kinds of reductions are used to remove useless places and transitions. In this way, the size of our model may be significantly reduced (see section 4.2).

3.4 Automatic generation of the IDS

Our IDS is divided into three layers (see figure 5). The first level is dedicated to the capture of events. Thus, this part is built as a listener waiting for messages coming from the system. These messages, related to the monitored program, carry a type of event and a value. This part is invariant for all generated IDS and will be detailed in the future work.

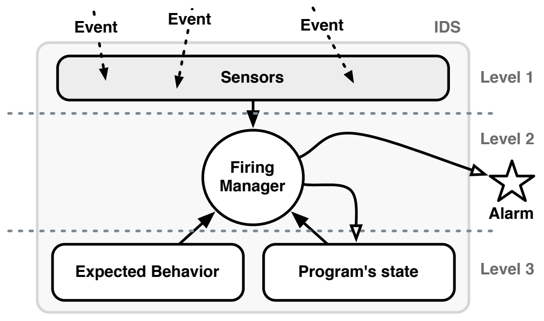


Figure 5. IDS architecture

The second floor is considered the engine of the IDS: *the firing manager*. It determines if a received event (in our work: a system call) is valid or not, according to the current program's state and the expected behavior. Like the previous one, this part is the same for all produced IDS.

The last layer is the only variable part of an IDS. It contains the expected behavior and the current representation of the program's state.

Expected behavior's representation The behavior is represented by an *incidence matrix*, which statically specifies the rules that govern the evolution of the net. This matrix indicates the conditions needed to fire each transition (*pre-conditions*) and the rules to update the net after the firing (*post-conditions*). The *code generation module* does this transformation by browsing all places and transitions and filling out the matrix. This step is very quick and easy since there is no expensive computation needed (see section 4.2). The size of the incidence matrix follows the rule : $p * t$ (where p and t are respectively the number of states and transitions in the model). In fact, only $3 * t$ values will be stored in this matrix.

Current program's state The initial state of the generated IDS is given by the model's initial marking. This state is store by the IDS as a vector containing the current values for each places of our net.

4 Application to a Simple Example

This section presents our proposal applied to a simple example. This way, we can illustrate the two phases described before. Some complete examples and results are also proposed.

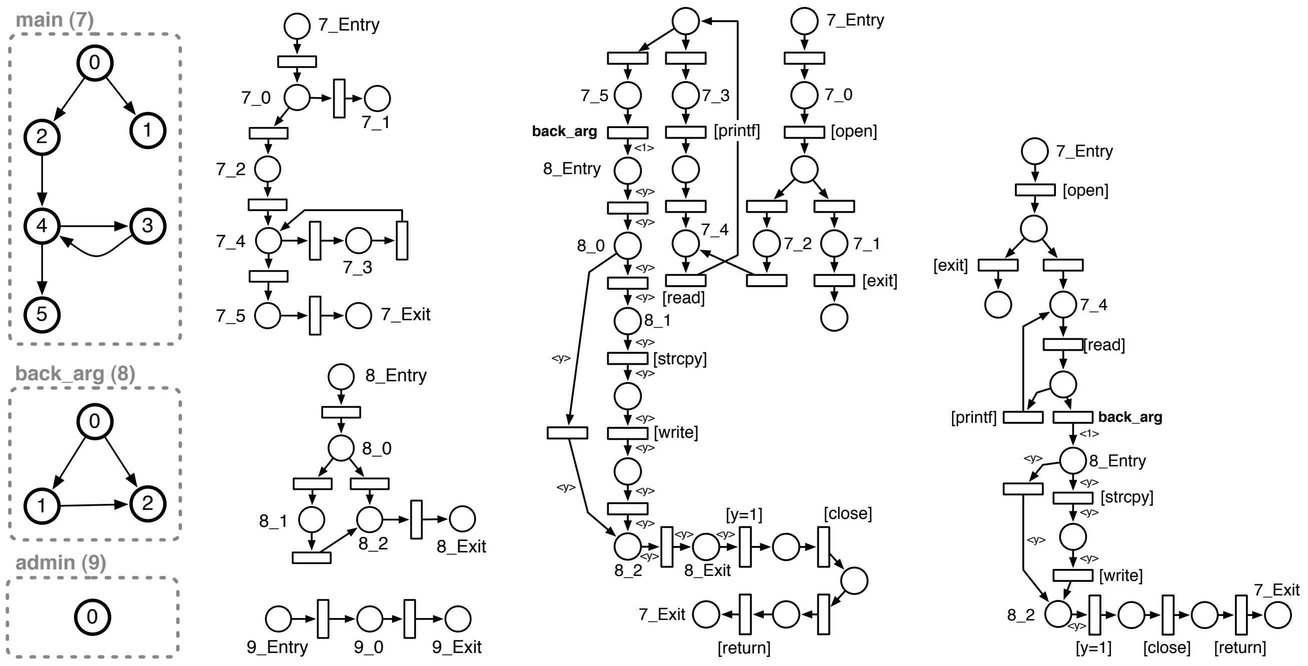
4.1 The program to be monitored

Listing 1 presents a simple program that reads a file, prints its content and writes the first argument to the same file. One of the functions used in this program is `strcpy()`. This function may be used to override the stack's content and consequently to change the behavior of the program by *buffer-overflow* approach.

In this example, given a special value to the first argument `ARGV[1]`, it is possible for an attacker to execute the `admin` function, which is called nowhere in the program.

```
int main (int argc, char** argv) {
  int back = open("exec.log",O_RDWR);
  if (back == -1) exit(-1);
  char buf[40];
  while ((read(back,buf,40))>0) printf("%s",buf);
  back_arg(argv[1],back);
  close(back);
  return 0;
}
static void back_arg (const char *s, int back) {
  if (s) {
    char text[20];
    strcpy(text,s);
    strcat(text,"ok\n");
    write(back,text,strlen(text));
  }
}
void admin() {
  printf("Unauthorized␣access␣!!\n");
}
```

Listing 1. A simple example



(a) GCC's output (b) Skeleton produced by the parser (c) Model includes system calls (d) Model after optimizations

Figure 6. Four steps are necessary to build the model of program's behavior

4.2 Extraction of the expected behavior

Presented in figure 6(a), GCC provides a CFG for the three functions: `main` (identified by 7), `back_arg` (identified by 8) and `admin` (identified by 9). Additionally to this graph, GCC includes some information into the blocks. For example, GCC indicates that an `open` function is called into block 0 of function `main`.

Following rules **R1**, **R2** and **R3**, the parser builds the basic skeleton (see figure 6(b)). During the link phase, the `admin` function is removed from the model because it is never called from any other function.

Thanks to GCC's indications, system calls are added to this skeleton following rules **R4**, **R5**, **R6** and **R7** (see figure 6(c)). Every blocks are still visible. The resulting net has 21 states, 23 transitions and 46 arcs.

Too much information about structure remain. After optimizations, 47% of nodes (places and transitions) have been removed. Numerous blocks have been deleted, and some structures' information have been merged too. Thus, the final net has only 11 places, 12 transitions and 24 arcs. It is presented on figure 6(d).

The overall time needed to build this model is less than 2 second. Moreover, no overhead has been measured during the program's compilation. The production of the CFG by GCC is thus negligible.

4.3 Building the IDS

The behavior's model is transformed into an incidence matrix. This transformation is done by applying the following algorithm on the Petri net:

```
**for all** transition $t$ **do**
    $matrix[t][\text{PreState}(t)] := -\text{Arc}(\text{PreState}(t), t)$
    $matrix[t][\text{PostState}(t)] := \text{Arc}(\text{PostState}(t), t)$
    $matrix[t][\text{``}Event\text{''}] := \text{Guard}(t)$
**end for**
```

$PreState(t)$ and $PostState(t)$ functions return respectively, the place's identifier preceding and following the transition t. The $Guard(t)$ function returns the value of the transition's guard. The $Arc(a, b)$ function returns the value of the arc linking two nodes: a and b.

The initial state (0) is stored by initializing the array $State$. The following algorithm describes this operation:

```
**for all** place $p$ **do**
    $state[0][p] := 0$
    $state[0][\text{``}main\_Entry\text{''}] := 1$
**end for**
```

The first level of the IDS is sensible to messages build upon the pattern : `<event,value>`. It receives these messages by way of signals and processes them as soon as they arrived. Thus, when a event is detected, IDS enqueues it; waiting to be processed by the firing manager.

The second level is designed to read the queue and try to apply the read event on the embedded representation. The following algorithms describe the way to proceed.

The first function indicates if a transition t is fireable considering the current state s and the incidence matrix.

```
function IS_FIREABLE($t, s$)
    $fire$ := true;
    for all place $p$ do
        if ($matrix[t][p] \geq 0$) then
            next;
        end if
        if ($state[s][p] > |matrix[t][p]|$) then
            $fire$ := false; break;
        end if
    end for
    return $fire$;
end function
```

The FIRE function creates a new state for each fireable transition.

```
function FIRE($t, s$)
    $newstate := s + 1$;
    for all place $p$ do
        $state[newstate][p] := state[s][p]+matrix[t][p]$
    end for
end function
```

The next function tries to fire all transitions for all current possible states given a specific event e. If a transition is fireable, new states are created and the old one is deleted (once all transitions of this state have been tested).

```
function TRY($e$)
    $modify$ := false;
    for all state $s$ do
        for all transition $t$ do
            if ($matrix[t][``Event''] \neq e$) then
                next;
            end if
            if (is_fireable($t, s$)) then
                fire($t, s$);
                mark_to_delete($s$); $modify$ := true;
            end if
        end for
        if mark_to_delete($s$) then
            delete($s$);
        end if
    end for
    return $modify$;
end function
```

The MAIN function first simulates a “no-event” event to fire all possible structural transitions. Once the model is blocked, it waits for an event. If the received event does not trigger any changes in the model, there is a problem. Thus, the ALARM function is raised. Otherwise, the process continues.

```
function MAIN
    repeat
        try(“$no\text{-}event$”);
        wait until(event $e$);
        if ($\neg$try($e$)) then
            alarm(); break;
        end if
    until true
end function
```

4.4 Results

We have chosen three other programs to try out our solution. Those are: `whois`, `gzip` and `ping`. Results are presented in the table below.

	whois	**ping**	**gzip**
Program’s size (lines)	874	2454	7323
Model’s size (nodes)	1499	2348	5692
Optimized model’s size (nodes)	627	1037	3301
Time to produce the model (s)	165	723	830

The program size is given by the number of source code lines. In fact, this number is obtained by browsing all files used during compilation and by removing all duplicated code.

These results illustrate the optimizations proposed in our simple example (see 4.2). For most of programs, the optimized model is 50% smaller than the basic one. This phase is also the longest one by taking near to 80% of total calculation time. In future works, we will focus on these algorithms to increase their speed.

Considering previous sequential programs test suites, our results concerning the representation size are similar to those proposed by other solutions. This can be explained by the fact that Petri Nets are a very compact representation for concurrent programs. Since our example is mostly sequential, the Petri Net and the corresponding state graph are very similar. Future work on concurrent multithreaded programs will show the real advantage of our solution on that particular aspect.

At last, the time needed to produce the model begins from the end of the compilation to the end of the optimizations. For the `gzip` benchmark, the production time must be compared to the 15000 system calls needed by some other solutions [15] to learn a consistent behavior.

5 Conclusion and Future Work

This paper presents a technique to automatically generate IDS dedicated to a program. This is a way to speed up the adaptation of such systems to face new intrusion techniques exploiting new vulnerabilities. Thus, it avoids long "training" phases of the IDS to adapt to specific programs.

A characteristics of our approach is to rely on Colored Petri Nets. Such an approach has several advantages. The first one is to allow on-line monitoring by "executing" the Petri net in parallel of the program. This allows to drive larger systems that the traditional "automaton" approach can not handle and will provide full benefits for the monitoring of multithreaded programs.

Another interesting feature (not explored in this paper) is to allow off-line monitoring of the program using "classical" verification techniques such as model checking or structural analysis available for Petri nets.

Our prototype of IDS generator is plugged on GCC. So far, it was only experimented with C but GCC also handles numerous languages. We foreseen a solution that could be language independent.

Currently, our approach only focus on system call supervision. We plan to apply similar techniques on other types of monitoring such as memory and stack profile.

We also plan to apply our approach to generate IDS for the monitoring of multithreaded programs. Our long-term objective is to achieve the monitoring of several parameters (system calls, stack profile, etc.) on concurrent programs.

Acknowledgments This work was supported in part by NSF through CAREER grant CCR-0133366.

References

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

[2] R. G. Bace. *Intrusion detection*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 2000.

[3] G. Berthelot. Checking properties of nets using transformation. In G. Rozenberg, editor, *Applications and Theory in Petri Nets*, volume 222 of *LNCS*, pages 19–40. Springer Verlag, 1985.

[4] J.-M. Colom, M. Sylva, and J.-L. Villarroel. On software implementation of Petri Nets and colored Petri Nets using high-level concurrent languages. In *7th International Workshop on Application and Theory of Petri Nets*, pages 207–222, 1986.

[5] A. Durante, R. D. Pietro, and L. V. Mancini. Formal specification for fast automatic ids training. In A. E. Abdallah, P. Ryan, and S. Schneider, editors, *FASec*, volume 2629 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2002.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[7] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. *sp*, 00:62, 2003.

[8] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Washington, DC, USA, 1996. IEEE Computer Society.

[9] D. Gao, M. K. Reiter, and D. X. Song. Gray-box extraction of execution graphs for anomaly detection. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 318–329. ACM, 2004.

[10] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*. The Internet Society, 2004.

[11] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag - ISBN: 3-540-41217-4, 2003.

[12] G. J. Holzmann. Trends in software verification. In *Springer*, 2003.

[13] A. K.Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. 1999. Online Publication.

[14] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[15] Z. Liu and S. M. Bridges. Dynamic learning of automata from the call stack log for anomaly detection. In *ITCC (1)*, pages 774–779. IEEE Computer Society, 2005.

[16] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[17] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[18] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.