Adaptable Intrusion Detection Systems Dedicated to Concurrent Programs: a Petri Net-Based Approach

Jean-Baptiste Voron, Clément Démoulins, and Fabrice Kordon
Université Pierre & Marie Curie,
LIP6-CNRS UMR 7606,
4 place Jussieu, 75252 Paris Cedex 05, France
jean-baptiste.voron@lip6.fr, clement.demoulins@lip6.fr, fabrice.kordon@lip6.fr

***Abstract*—Intrusion detection systems (IDS) are one way to tackle the increasing number of attacks that exploit software vulnerabilities. However, the construction of such a security system is a delicate process involving: *(i)* the acquisition of the monitored program behavior and its storage in a compact way, *(ii)* the generation of a monitor detecting deviances in the program behavior. These problems are emphasized when dealing with complex or parallel programs.**

This paper presents a new approach to automatically generate a dedicated and customized IDS from C sources targeting multi-threaded programs. We use Petri Nets to benefit from a formal description able to compactly describe parallel behaviors. Obtained models can then be enhanced with extra requirements such as resources usage limits or temporal execution bounds by means of *observers*. We illustrate the benefits of our approach on a recent class of attacks targeting web servers.

I. Introduction

Context: As the size and complexity of software constantly grow, testing and verifying the correctness of a system become more and more difficult. Monitoring the system execution can be considered as a complementary approach to traditional ones. It also increases confidence in the correctness of the system at runtime [1].

Some security software use monitoring to protect systems from intruders. More precisely, a family of intrusion detection systems (*anomaly-based*) uses a program *reference behavior model* and, at runtime, compares it to the observed system execution traces. If a deviation from the reference behavior is observed (e.g. an unexpected trace or event), an alarm is triggered. A countermeasure may then be taken. Obviously, the precision of monitoring strongly relies on the quality and the precision of the reference behavior model.

However it appears that statically-computed reference behavior models are no more sufficient to handle new kinds of attacks [2]. Current attacks use either a combination of legal actions in order to reach an abnormal state [3] or mimic legal behavior and use a malicious data-flow to achieve an intrusion or to cause damages [4]. Static models are not relevant anymore in such cases because all actions are legal. Static analysis has to be enriched by a dynamic analysis that focuses on data, execution times or context variations and that can trigger appropriate alarms.

Building the reference behavior model of a given program is a challenging problem and several approaches have been considered during the past years:

- some approaches rely on *learning techniques*. In this case the reference behavior model is built according to the observation of execution traces [5],
- other approaches build the reference behavior model *directly from the program* [6] either manually (by a human expert) or thanks to static analysis of source [7] or binary [3] code.

Problems: Once the model is built, both approaches store the reference behavior model into a monitor which is dedicated to the analysis of the program execution. Current solutions suffer two main drawbacks presented below.

First, the reference behavior model size for complex systems is a *huge* problem since a lot of information has to be stored and processed at runtime. This is particularly true for parallel or multi-threaded programs, when all possible executions must be recorded and modeled. Moreover, considering the previous remark about new kinds of attacks, reducing the precision of the model may not be a good solution.

Second, the complexity of the reference behavior model is problematic for the monitoring efficiency. In fact, the more precise the model is, the more parameters have to be checked at runtime. Precise models generally lead to a high overhead at runtime. Thus, a compromise between the ability to discover fine attacks, the size of the model and the monitoring efficiency must be found.

Contributions: Considering these critical points, we propose an automatic way to generate host-based IDS [8] from program sources. The main characteristics of our approach are:

1) we use *Petri nets*, a formal notation, to describe the reference behavior model associated with a program. We show that multi-threaded programs with very large state-space can be monitored with a limited overhead, by a small and compact monitor, without any loss of precision or accuracy.
2) we propose to enrich the reference behavior model by *additional security constraints* in order to deal with complex and new attacks. We also show that Petri nets are well adapted to this kind of modular construction.

Since the entire process is *automatic*, a new monitor can be generated each time the program is patched or updated. The specification of new security constraints is also costless thanks to the definition of *observers*. Consequently, our method is much more resilient to *0-day* attacks (*i.e.* exploitation of undiscovered or unpatched application vulnerabilities [9]).

Finally, our approach includes a *flexible* and *customizable* process that allows designers of security solutions to tune the construction process in order to build a dedicated set of monitors.

Contents: To get a consistent vision of the IDS generation process, section II recalls how we extract relevant informations from program sources to automatically build a behavioral reference model as already presented in [10]. Then, section III presents the automatic code generation of a dedicated IDS from the behavioral reference model. Section IV introduces how extra requirements that can be added to the reference models. Finally, section V illustrates the benefits of our approach thanks to several experiments including a recent (June 2009) *denial of service (DoS)* attack on a HTTP server.

II. Building the Reference Behavior Model

Our approach consists in building a reference behavior model based on a formal notation that describes all acceptable program behaviors. This model is used, at runtime by a monitor, to verify the correctness of the program execution. In fact, all monitored events (*e.g.* function calls, i/o, memory allocations) are intercepted, compared and matched to this model. Events that are not expected by the model will trigger an alarm.

A. IDS Factory: the Overall Process

This section recalls our construction process of an IDS. This automatic construction tackles the drawbacks identified in the introduction by: *(i)* a modular static analysis of the program source code, *(ii)* a Petri net-based representation of the reference behavior and *(iii)* the possibility to enrich the produced model with *observers* introducing a more accurate control on the monitored program execution.

Figure 1 sketches these three steps in our building process. First, we process C source code (Section II-B) to produce the corresponding Petri net model (Section II-C). Then, the Petri net model is embedded into the IDS to be used as a reference behavior during the execution of the monitored program (Section III).

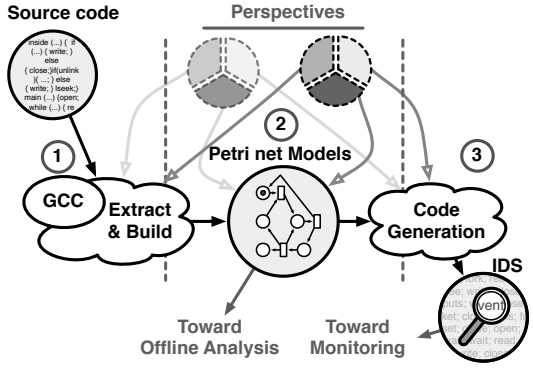


Fig. 1. A three-steps, perspective-driven construction flow

Perspectives are used to select and process the relevant information for monitoring and thus to build appropriate representations of program behavior. Perspectives are sets of parameters used to control and to tune information extraction, Petri nets production as well as IDS generation. This notion is not detailed in this paper since it was presented in [10].

In this paper, we use the *structural perspective* (in charge of control-flow aspects of the program), the *network perspective* (dedicated to network primitives), the *pthread perspective* (dedicated to thread management by the POSIX library) and finally the *system call perspective* for handling all other system calls.

B. Extracting relevant information

We use the *extended Control Flow Graph* (eCFG) produced by the C front-end of GCC as a basis for our static analysis. GCC is of interest since it is commonly used when compiling applications. Moreover it offers interfaces to be extended and used in other applications. Thus, our IDS generator is directly connected to GCC and the overall construction process is performed after the compilation of the application to be monitored.

Let us remind that a control flow graph is a directed graph representing the behavior of an individual process in a discrete event model. Each node in the eCFG represents some possible states of the process. Each action that a process may take from a given state is represented by an edge relating to potential successor states.

The eCFG dumped by GCC defines instructions blocks and their relationships. All control structures such as `for`, `while`, `continue`, `break` are processed in terms of block sequences in the eCFG, thus reducing the number of structures to be considered by our factory.

Listing 1 shows a snippet of the `nullHTTPd` server [11] source code. This portion of code (i) checks if the server can deal with one more client (line 3 to 9), (ii) handles new incoming client (line 14) and (iii) delegates a new thread for each request (line 18).

```
1  for (;;) {
2   /* -- check for a free thread -- */
3   for (i=0;;i++) {
4    /* -- if no thread is available... wait and retry -- */
5    if (i>=config.server_maxconn) {
6     sleep(1); i=0; continue;
7    }
8    if (conn[i].socket==0) break;
9   }
10  if (conn[i].PostData!=NULL) free(conn[i].PostData);
11  if (conn[i].dat!=NULL) free(conn[i].dat);
12  memset((char *)&conn[i], 0, sizeof(conn[i]));
13  fromlen=sizeof(conn[i].ClientAddr);
14  conn[i].socket=accept(ListenSocket, [...]);
15  if (conn[i].socket<0) continue;
16  /* -- delegate a new thread for this request -- */
17  conn[i].id=1;
18  if (pthread_create(&conn[i].handle, [...])==-1) {
19   logerror("htloop()␣failed...");
20   exit(0);
21  }
22 }
```

Listing 1. `NullHTTPD` server source code snippet

Listing 2, shows the eCFG extracted by GCC from the previous snippet. Several blocks are declared. Inside of these blocks, some part of initial code can be recognized. For example, block 6 (line 1) refers to the initialization of the

first loop (line 3 in Listing 1), block 16 (line 23) corresponds to the lines 14 and 15 while block 17 (line 36) refers to the body of the last `if` structure at line 18 in Listing 1.

```
1 # BLOCK 6, starting at line 571
2 # PRED: 4 (false) 18 (fallthru)
3 [server.c : 571] iD.5737 = 0;
4 # SUCC: 7 (fallthru)
5
6 # BLOCK 7, starting at line 572
7 # PRED: 6 (fallthru) 10 (fallthru)
8 [server.c : 572] D.5745 = config.server_maxconn;
9 [server.c : 572] D.5746 = (int) D.5745;
10 [server.c : 572] if (D.5746 <= iD.5737)
11   goto <bb 8>;
12 else
13   goto <bb 9>;
14 # SUCC: 8 (true) 9 (false)
15
16 # BLOCK 8, starting at line 573
17 # PRED: 7 (true)
18 [server.c : 573] sleep (1);
19 [server.c : 574] iD.5737 = 0;
20 [server.c : 575] goto <bb 10>;
21 # SUCC: 10 (fallthru)
22 [...]
23 # BLOCK 16, starting at line 595
24 # PRED: 15 (false)
25 [server.c : 595] D.5748 = (long unsigned int) iD.5737;
26 [server.c : 596] D.5761 = (long int) iD.5737;
27 [server.c : 596] D.5762 = (void *) D.5761;
28 [...]
29 [server.c : 596] D.5765 = pthread_create (D.5764, [...]);
30 [server.c : 596] if ([server.c : 596] D.5765 == -1)
31   goto <bb 17>;
32 else
33   goto <bb 18>;
34 # SUCC: 17 (true) 18 (false)
35
36 # BLOCK 17, starting at line 597
37 # PRED: 16 (true)
38 [server.c : 597] logerror (&"htloop()␣failed..."[0]);
39 [server.c : 598] exit (0);
40 # SUCC:
41
42 # BLOCK 18
43 # PRED: 15 (true) 16 (false)
44 [server.c : 601] goto <bb 6>;
45 # SUCC: 6 (fallthru)
```

Listing 2. Part of the eCFG produced by GCC for the listing 1

In order to build the reference behavior model, our factory extracts, from this eCFG, information which is relevant for the monitoring. The *control* information is processed before all *system* and *library calls*. Each extracted element is associated with a small piece of Petri nets according to building rules that are defined by perspectives.

C. Building the Reference Behavior Model

Introduction to Petri nets: Petri nets [12] are a graphical formal modeling language where *places* represent resources or states, and *transitions* describe relations between places. Places hold *tokens* potentially carrying data that are consumed or produced by transitions according to a *firing rule*. Tokens are carried out from places to transitions and then from transitions to places by means of *arcs*. The *firing rule* is the following: when *input places* of a transition hold a sufficient number of tokens having the appropriate values, the transition can be *fired*. If it happens, such tokens are removed from input places and new tokens are produced in *output places*. Due to lack of space, we do not provide a formal definition of Petri nets which can be found, for example, in [12] or [13].

We however introduce the following notations that are used later in the paper. P is the finite *set of places* and T is the finite *set of transitions* ($P \cap T = \emptyset$). C is a *color function* which is a mapping from $P \cup T$ to Σ, a set of finite and non empty sets; W^- and W^+ are the *forward* and *backward incidence matrices* that associate with each pair (p,t) of $P \times T$ a mapping from $C(t)$ to $Bag(C(p))$. ϕ is a *guard function* that associates with each $t \in T$ a mapping from $C(t)$ to $\mathbf{B}$ (the set of booleans). Finally, we associate a *label* λ with each transition. $\lambda(t) = \varepsilon$ means that the transition t can be fired at anytime (once its preconditions are satisfied). Other labels $\lambda(t) = e$ indicate that t can be fired only if the event e has been caught by the IDS (see Algorithm 1).

The set of *inputs* (resp. *outputs*) of a place $p \in P$ is the set ${}^\bullet p$ (resp. $p^\bullet$) defined by: ${}^\bullet p = \{t \in T \mid W^-(p,t) \neq 0\}$ (resp. $p^\bullet = \{t \in T \mid W^+(p,t) \neq 0\}$). The same sets can be defined for transitions. We note $m(p) \in Bag(C(p))$ the *marking* of a place p that indicates the number and the quality of the tokens inside this place.

Petri nets are suitable for the description of concurrent behaviors [13] and can be seen as a compact way to represent all the possible executions of the corresponding model (*i.e.* the *state space*). This characteristic is crucial since safety-critical programs are often complex, multi-processed or multi-threaded. Their state spaces are generally huge due to the interleaving of elementary actions. This interleaving is not represented in the Petri net itself since it models the structure of the program (its dynamics is represented by tokens). Petri nets can be seen as a *state space generator* and thus, as a compact representation of state spaces. Section V-B shows some convincing data for a small multi-threaded program.

So, since state spaces are hard to handle with the classical representation techniques of IDS such as automata or regular expressions, we use Petri nets to describe legal behaviors of programs we want to monitor.

For composition purposes we use *colored* and *hierarchical* Petri nets [12] where *submodels* can be attached to places of a standard Petri net (see Figure 3). Hence, a model is composed of one or more submodels connected by means of *input/ouput* places. A *virtual* place can also refer to another one to make links between submodels.

In our approach, Petri nets are used to represent legal program behaviors. Places stand for legal behavior states, reachable at runtime. Transitions are used as event barriers, fireable only if the expected events are received. And tokens carry thread, process or resource identifiers.

Building the structural model: Information about the program structure is always processed first. Thanks to construction rules defined by the *structural perspective*, each control element is extracted from the eCFG and is transformed into a piece of Petri net. More precisely, blocks (`BLOCK`) are transformed into places and links between blocks (`SUCC`, `PRED`) are transformed into transitions. Static function calls are also resolved and dedicated transitions are used to link them together. Special patterns are used [10] to deal with recursive

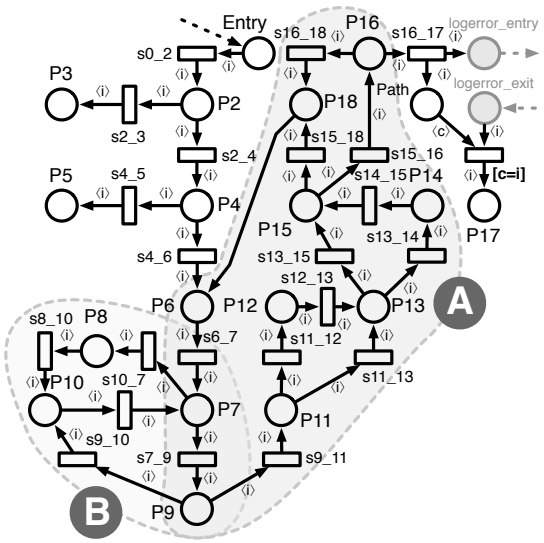


Fig. 2. Structural model built from Listing 2 processing

calls and to avoid the "*impossible path problem*" [7].

Figure 2 shows the structural model associated with the eCFG presented in Listing 2. Dotted arcs represent links to the rest of the structural model. The main loop (lines 1) is modeled in area **A**, while the small one (line 3) is in the area **B**. An external function call (`logerror`) is made in block `16`. Two virtual places (**logerror_entry** and **logerror_exit**) are used to connect this function to the `logerror` Petri net. Name of places (resp. transitions) is built according to this rule **P**{**block_number**} (resp. **s**{**block_from**}_{**block_to**}).

Enriching the model: Information to be checked during the program execution (*i.e.* declarations, system calls, library calls, etc.) is processed according to the remaining selected perspectives. Thanks to the same pattern mechanism, these perspectives lead to the construction of sub-Petri nets connected to the structural model.

As shown by Figure 3, sub-models are carrying more details about the block contents. In this example, we describe the blocks 15 and 16. The first one contains an `accept` system call modeled by a *labeled transition*. At runtime this transition is fireable only if an *accept* event is caught from the observed program. The second detailed block (16) contains the instruction creating a new thread: `pthread_create`. This behavior is modeled by a transition that produces two tokens: one for the parent's execution state and one for the child's one. A virtual place (**htloop_entry**) is also used to indicate that the life of the child begins in `htloop` function.

Once all selected perspectives have been processed, the model for `nullHTTPd` is composed of 597 sub-models, 2835 places, 1737 transitions and 4809 arcs.

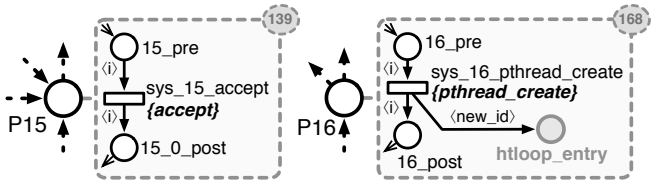


Fig. 3. Sub-models produced by the *syscall* perspective for blocks 15 & 16

Finalizing the model: This model can be flattened and optimized in order to reduce its size and its complexity. First, sub-models are merged with their place and targets of virtual places are resolved. The complete algorithm is also presented in [10]. Second, formal structural reductions are applied. They preserve the Petri net behavioral properties [14] but reduce its size. Our experiments show that those reductions can reduce the size of the model from 40% up to 70% (see Table I). For `nullHTTPd`, the finalized reference behavior model contains 485 places, 583 transitions and 1761 arcs.

III. Automatic Generation of the Dedicated IDS

Once the behavior reference model is built. The associated IDS must be produced. Our implementation choices are considered in this section.

A. Overall Architecture

The overall architecture of a generated IDS is described in Figure 4. It consists in two parts. The dynamic one (the *model*) is generated by our factory (*Evinrude*) according to both the program and the set of symbols to be monitored. It is linked to the static part (the *Firing Algorithm*) that is independent from the monitored program.

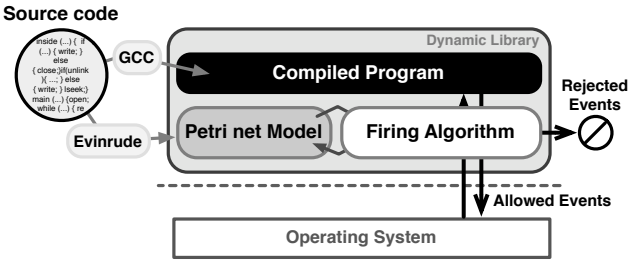


Fig. 4. Architecture of the generated IDS

Catching incoming events: The IDS is compiled as a dynamic library (`.so`) loaded thanks to the `LD_PRELOAD` environment variable prior to monitored program execution. This library overloads desired symbol definitions and thus allows to define actions to be performed before executing the real calls. Listing 3 shows the kind of actions we perform before allowing an `accept` system call to be processed.

```
1  /* backup the real system call */
2  _BIANCAaccept =
3    (int (*)(int, struct sockaddr*, socklen_t*))
4    dlsym (RTLD_NEXT, "accept");
5
6  /* override the system call */
7  int accept(int fd, struct sockaddr *a, socklen_t *alen) {
8    int ret;
9    /* ask the model whether the event is allowed or not */
10   fire(&global_table, ACCEPT, tid);
11   /* call the real system call */
12   ret = _BIANCAaccept(fd, a, alen);
13   /* return the result to the monitored program */
14   return ret;
15 }
```

Listing 3. How to overload the `accept` system call

The main advantages of this interposition technique are flexibility and easy initialisation. Thus, we have been able to develop a working prototype and to focus on the benefits of the behavior reference model representation. But, from a security point of view, this mechanism can be easily circumvented by an attacker. Aware of that weak point, we have designed our IDS in such a way that this *catching layer* can be replaced without impacting the rest of the IDS. Future work will consider more advanced mechanisms such as virtual-based interposition or monitoring in a isolated memory space.

Validating events: Each time an event is caught (*i.e.* observed) by the IDS, it must be validated against the reference behavior model. This validation is performed thanks to the `fire` function (see line 10 in Listing 3).

The behavior of this function is presented in Algorithm 1. M is the set of current states of the model and e is the event that has been caught from a process or thread identified by id.

Algorithm 1 The monitor firing algorithm

```
1:  function FIRE(M, event, id)
2:      S ← false
3:      for all m ∈ M do
4:          P_id ← {p ∈ P | id ∈ m(p)}
5:          for all {t ∈ p• | p ∈ P_id} do
6:              if (λ(t) ≠ ε) ∧ (λ(t) ≠ event) then
7:                  M ← M \ {m}              ▷ discarded markings
8:                  continue;
9:              end if
10:             if m(p) ≤ W⁻(p,t) ∧ φ(t) then
11:                 m'(p) ← m(p) − W⁻(p,t) + W⁺(p,t)
12:                 if λ(t) = ε then
13:                     M ← M ∪ {m'}
14:                 else
15:                     m(p) = m'(p)         ▷ here, λ(t) = event
16:                     S ← true
17:                 end if
18:             else
19:                 M ← M \ {m}
20:             end if
21:         end for
22:     end for
23:     return S
24: end function
```

To check the validity of the event, the monitor tries to fire all transitions (line 5) that are outputs of places ($p^\bullet$) containing an *id* token. To be fireable, a transition must be labeled by ε or by the event e that is being processed (line 6). The marking of input places as well as the transition guard are checked (line 10). If all conditions are satisfied a new marking is computed (line 11). If the transition is labeled by ε, we add the new state to M. This new state will be processed during the next loop. If the transition is labeled by e, then the event is valid (line 16) and the reached marking (m') replace m (*i.e.* the one we fired the transition from) in M. If the considered transition labelled by e cannot be fired, the marking is removed from M. If no transition labeled by e is fired during this execution (*i.e.* $S = \mathsf{false}$), the `fire` function will return false and the execution will be stopped.

B. Implementing the algorithm

The proposed algorithm is optimized to reduce the monitoring overhead. Indeed, in case of parallel or multi-threaded program, a global interpreter lock (GIL) is necessary since the incidence matrix W is shared between all threads.

To release the GIL, we divide the Petri net into several asynchronous sub-Petri nets, each one being executed by a different process or thread. The partitioning is performed by reordering the incidence matrix columns in order to extract autonomous sub-Petri nets (*i.e.* that minimize interactions). This partition corresponds to independent synchronized state-machines. Thus, only the remaining synchronizations have to be protected by a *mutex* mechanism which reduces the overhead as shown in Figure 8. Parallel code is generated for each sub-Petri nets. It encodes the corresponding state machine and performs a relation to caught events.

So, the dynamic part of the IDS consists in the generated code corresponding to the Petri net. The static part implements the main algorithm checking for the validity of events according to the dynamic part and updates the markings.

IV. Adding Requirements to the Behavior Model

Since static analysis of the source code produces an *over-approximation* of the program behavior, produced models are subject to false-negatives. Such false-negatives are typically exploited by new kinds of attacks provoking illegal actions using combinations of legal behavior [15].

A way to cope with this lack of precision is to provide new constraints for the model. In this paper, we present two kinds of constraint patterns expressed by means of Petri nets components. This technique, originally introduced by [16] for verification purpose, consists in adding extra pieces of models that *observe* the existing model and *enforce* constraints or formulas at runtime.

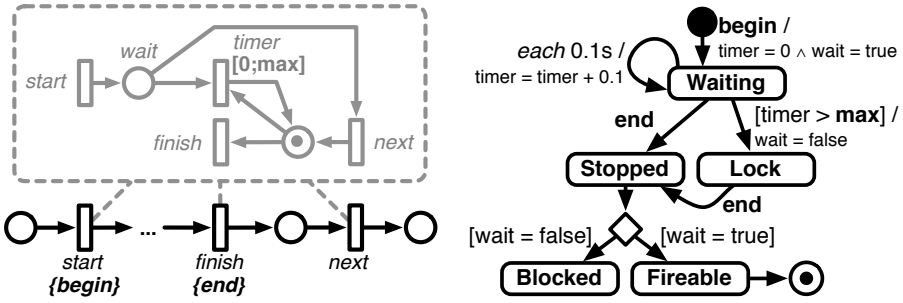
Let us introduce two kinds of *observers*: resource usage observers, and temporal bounds observers.

Each observer requires *observable points* (*i.e.* where it is connected to the model). These points are identified in the source code and then located in the Petri net model thanks to the links to the corresponding program source. *Bounds* of these observers are set up by an engineer. They could also be computed dynamically during the program execution.

A. Resource Bounds Observers

At runtime, programs usually deal with resources such as files, sockets, semaphores, threads, processes, etc. An inadequate (but legal) use of those resources can lead to a crash of the program or of the underlying system.

The *resource-bounds observer* observes and constrains the number of resources that can be *acquired* and *released* during the execution of the program. Based on a counter, it triggers an alarm if the counter is lower than the minimum or greater



(a) Sub-Petri net attached to the main model (b) Observer statechart

Fig. 5. A temporal-bounds observer that limits the execution time to *max* time units

than the maximum bound. It requires two observable points: (i) where the resource is acquired and (ii) where it is released. In both cases, there are potentially multiple locations.

The gray sub-Petri net, at the top of Figure 6, is an example of an observer. It is attached to the main net (the black part of the Figure 6) thanks to the observable points: **acquire** and **release** transitions. When a "*get*" event is caught by the monitor, it tries to fire the transition labelled by this event. The number of current allocated resources $\langle \mathbf{r} \rangle$ is fetched from place **counter**. If this number is lower than the specified bound (*max-bound*), the transition can be fired, a resource from the place **resources** is consumed and the number of allocated resource is increased ($\langle \mathbf{r{+}{+}1} \rangle$). Otherwise, the transition **acquire** cannot fire and the execution is stopped.

A similar check is performed when the "*set*" event is caught by the monitor. In this case, the number ($\langle \mathbf{r} \rangle$) of current allocated resources is compared to the *min-bound* before being decreased ($\langle \mathbf{r{-}1} \rangle$). If the check fails, **release** cannot fire and the execution is stopped.

More sophisticated resource-bounds observers can also be defined. Since the process or thread identity is carried by Petri net tokens, a maximum usage bound can be defined *per process* or *per thread*. Hence, if a thread or a process starts using resources in an incorrect way, it is possible to stop without breaking the execution of the entire program. The Section V presents a typical use of this kind of observer in order to avoid a denial of service attack in a web server.

B. Temporal Bounds Observers

We define a *temporal-bounds observer* as a way to observe and control the time needed by a program to go from one state to another during its execution. Too much variations of this

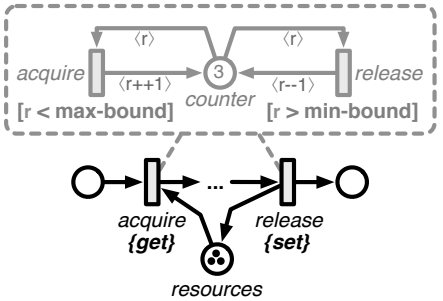


Fig. 6. Sub-Petri net that limits the use of a resource between *min* and *max*

value is the typical signature of a system or program anomaly. To avoid this situation, extra requirements such as timeout can be defined and added to the reference behavior model.

Like the resource bounds observer, the temporal-bounds observer is expressed by means of a Petri net submodels. Its representation uses a *time transition* coming from the time Petri nets [17]. This particular transition can be fired only before *max* time units elapsed from the moment it is enabled (*i.e.* all conditions are satisfied).

Consequently the time bounds observer uses one bound: *max*. This limit represents the maximum time allowed to the program to go from the first observable point (**start** transition) to the second one (**finish** transition). The last observation points (**next** transitions) are automatically computed since they follow the **finish** transition.

An example of such an observer is presented in Figure 5(a). Here, the time transition **timer** defines that *max* time units are allowed, at runtime, to go from **start** to **finish**.

The behavior of the observer is depicted in Figure 5(b): once **start** is fired (when a "*begin*" event is caught), the timer is started. If the timer value overpasses *max* time units, the observer is locked. Thus, when the "*end*" event is caught, the timer is stopped. At this stage, the observer is either unlocked (`wait=true`) or locked (`wait=false`). In the first case, next events caught by the monitoring are allowed. In the second case, the observer is blocked and thus stops the program.

V. Experimentation

This section illustrates the benefits of our approach and the efficiency of the produced IDS thanks to three benchmarks.

The first one concerns standard and sequential programs (Section V-A). It shows that on simple and usual programs the produced IDS do not suffer from an excessive runtime-overhead and can be compared to existing monitoring solutions.

The second benchmark is dedicated to the analysis of a highly multi-threaded simple server (Section V-B). This program has a huge state space and its monitoring is impossible with classical approaches based on automata. We show that, in addition to the monitoring of such an example, our IDS is able to perform this validation without a prohibitive overhead.

The third benchmark demonstrates the usefulness of extra-requirements. It deals with the detection of a recent denial of

Service on the `nullHTTPd` server (Section V-C).

A. IDS for Sequential Programs

We first assess our approach using a benchmark of well-known programs such as `gzip`, `cat`, `cp`, `openssh` and the `python` interpreter. These programs are either sequential or use very few threads or processes.

Table I provides the size of the generated Petri nets for these programs (*i.e.* the behavioral reference model embedded into the monitor) and the percentage of nodes that have been deleted during the finalization phase (see Section II). It also shows the time needed to generate those models and the size of the resulting IDS.

These results confirm that our IDS factory is able to process *real-life* programs without any human intervention. Any changes in the program can be propagated in a new version of the IDS in a very limited time.

Figure 7 illustrates the monitoring overhead for `cp`, `cat` and `gzip` induced by our IDS. It also reveals the *composition* of this overhead. Tests are performed using a set of large inputs and execution time is measured without any monitoring. Then, the same inputs are processed on the monitored version. We also measure the overhead with a slightly modified version of the monitor: a *pseudo-monitor* (*i.e.* a monitor *without* the firing rule implementation) to evaluate the part corresponding to the "event catching" mechanism.

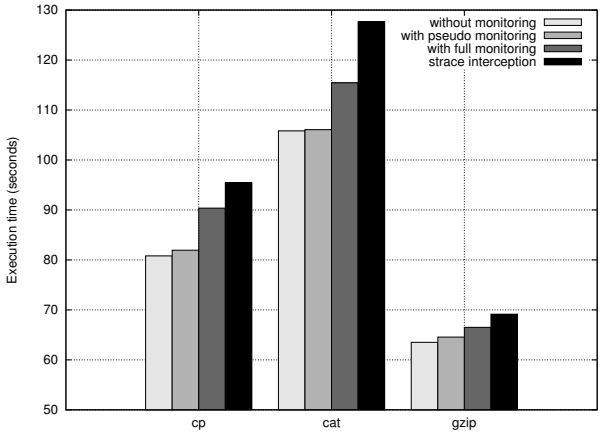


Fig. 7. Monitoring overhead (with and without the firing rule algorithm) for three sequential programs compared to the `strace` performances

These results are compared to the overhead observed when the program is monitored with `strace` (from the family of `ptrace`-based solutions) to catch events (in a situation similar to the *pseudo-monitoring*). Basically, `strace` is a debugging linux tool able to catch system calls and is commonly used in host-based IDS implementation.

The event catching mechanism induces an execution overhead below 2% and appears to be efficient compared to the `strace`-based solutions. In fact, all symbols are statically loaded during the library initialization and only desired system and library calls are caught at runtime. Moreover the `LD_PRELOAD` does not require any kernel access or trap mechanism in order to catch system calls.

The full monitor overhead remains around 10% depending on the number of calls. This is always lower than the `strace` based event catching mechanism alone. These performances are thus better than most of the ones produced in the current state of the art (let us cite a well accepted solution in [18] that is about 20% overhead).

B. IDS for Multithreaded Programs

Let us now evaluate the benefits of our approach on a massively multi-threaded program and, in particular, the evolution of the overhead when parallelism increases. For that purpose we use a small program to be monitored: the *echo server*. This program has been fetched and adapted (to be able to scale up) from a public database of C programs. It is implemented as a main thread waiting for queries. For each query, the main thread delegates a new thread to execute the service.

A characteristic of this example is its *state-space explosion*: the number of reachable states grows exponentially according to the number of clients asking for a service (See Table II). Such an application cannot be handled by current monitoring approaches, due to the high memory consumption required to store the reference behavior model as automata.

TABLE II
ECHO SERVER STATE-SPACE ACCORDING TO THE NUMBER OF CLIENTS

Number of simultaneous clients	Behavior size (# of states)
5	1.68×10^4
10	2.82×10^8
25	1.34×10^{21}
50	1.79×10^{42}
100	3.23×10^{84}

The source code of this application has been processed by our IDS factory in 5 seconds. The final embedded model is a Petri net composed of 13 places, 18 transitions and 39 arcs. Storage of the reference behavior model requires a few Kbytes for the structure and another Kbyte to store threads context (the second part increases linearly with the number of threads).

The test procedure considers 49 executions of our server (numbered by i from 2 to 50). Execution i involves i clients. Each client makes $\frac{10^9}{i}$ calls to the server and thus expects the same number of answers. We measure the time needed by the server to proceed all calls. Note that clients run on separate machines, thus generating up to 50 simultaneous threads in the server. Performances have been measured on a Intel Core2Duo at 2.66GHz, with 2GB of RAM connected with Gigabit Ethernet to a cluster where 50 nodes are playing the client role. Detailled results are presented in Figure 8.

Whatever the number of concurrent clients, our event catching mechanism remains costless (*pseudo-monitoring* in Figure 8). The third curve shows that most of the overhead (60%) comes from the remaining locks used for synchronizations between threads (*pseudo-monitoring (and locks)*).

The global observed overhead for this example is about 34%. It is almost constant from 2 to 50 simultaneous clients (*i.e.* threads). These results seems average but this example is one of the worst possible configuration since we observe that

TABLE I
PETRI NET ABSTRACTIONS AND RELATED STORAGE FOR SOME COMMON PROGRAMS

Program	Source Code (*loc*)	Petri Net Model (reduced) Places	Transitions	Arcs	Reduc. Factor (*% nodes*)	IDS size (*KBytes*)	Time (*seconds*)
`su-7.5`	522	68	70	175	64%	97	*11*
`cat-7.5`	782	225	262	713	56%	101	*25*
`cp-7.5`	1 142	715	916	2 835	54%	184	*110*
`stty-7.5`	1 905	239	276	809	67%	207	*35*
`nullhttpd-0.5.1`	2 023	485	583	1 761	56%	227	9
`netcat-0.7.1`	4 701	309	432	1 178	54%	201	11
`openntpd-3.9p1`	5 514	589	743	2 474	52%	313	13
`gzip-1.2.4`	8 049	408	598	1 757	68%	270	11
`wu-ftpd-2.6`	24 742	2 022	2 786	9 061	56%	988	328
`openssh-5.2p1`	90 069	37 653	47 502	180 884	61%	11 264	1 137
`squid-2.7`	138 624	20 622	25 813	107 590	50%	13 030	23 991
`httpd-2.2.13`	293 840	12 181	14 769	50 724	36%	14 844	15 652
`python-2.6.2`	1 142 172	54 258	66 460	366 199	70%	15 360	25 781

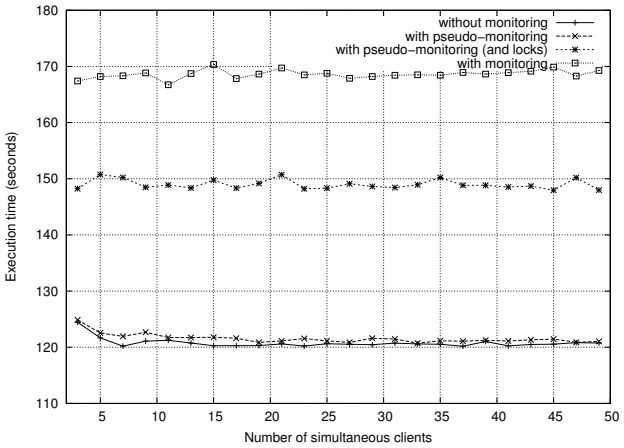


Fig. 8. Runtime overhead considering various monitor implementations for the `echo-server` example and according to the number of clients

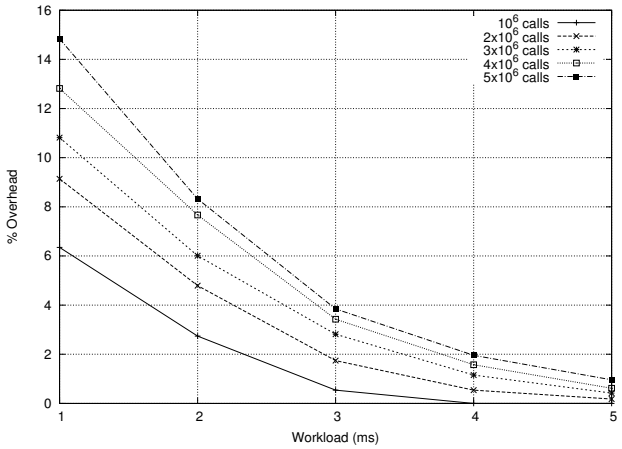


Fig. 9. Runtime overhead for 50 simultaneous clients making 5×10^6 requests and considering various server additional workload for the `echo-server` example

all threads are waiting for an access to the kernel space to execute system calls.

In a real application, the overhead due to locks would be hidden by the computation done in user space and by the time spent in i/o. This situation is shown in Figure 9 where the server executes some code before answering a request. Thus, the ratio between user instructions and system calls is more like that a standard application.

In this second experiment, we show that a workload of 3ms is sufficient to divide the overhead by almost two. For a server that has to answer 5×10^6 queries, the overhead is below 2% if each threads does a work (*i.e* an i/o or a user space computation) of 5ms before answering.

Thus, if we consider that most of parallel or multi-threaded applications are doing some computation between system calls, our solution appears to be quite efficient.

C. The Slow Denial of Service Attack

A denial of service (DoS) attack is an attempt to make a service unavailable. A classic DoS attack is the distributed denial of service (DDoS) that involves a large number of computers flooding the bandwidth of the targeted server, thus overloading it.

In June 2009, a new class of DoS attack was discovered. It only requires a single computer and is able to bring down most of web servers. A demonstration of this attack (called Slowloris) is available in [19]. It uses partial and *valid* HTTP requests to hold connections and to slow down the web server. Since the HTTP protocol is not violated, host-based IDS are unable to detect the problem. Moreover, the bandwidth remains available and HTTP packets are valid, thus network-based IDS (that monitor network data) cannot diagnose the problem.

Most of multi-threaded HTTP servers like Apache web server [20] delegate a thread for each request or group of requests and have a maximum number of working threads. Thus, if an attacker can hold as many connections as the maximum number of working threads, the service becomes unavailable. This maximum number of simultaneous connections is usually low (200 for Apache web server and 50 for `nullHTTPd`).

Unlike DDoS, this attack is stealthy: only the targeted service is affected and the bandwidth and CPU usage remain very low. Moreover, no log message is written by the server because no request is completed. Thus, this attack is tricky to detect even for system administrators.

For demonstration purposes, we use the version 0.5.1 of the `nullHTTPd` [11] server, which is vulnerable to the Slowloris attack. The reasonable size of the generated IDS (see Table I) makes it a perfect illustration. The server is configured to

handle 50 simultaneous client (this is the default configuration) and performances have been measured on a cluster of bi-processor Intel Xeon at 2.8GHz, with 2GB of RAM, and connected with a Gigabit Ethernet.

We run Slowloris on a single host. We also set 15 clients that request a page of 1Mbyte on the server. Figure 10 shows the evolution of the average response time according to the number of connections hold by Slowloris. Since all requests are valid, no anomaly is detected by our IDS but performances are going down. Thus, protected or not, the server quickly becomes unavailable.

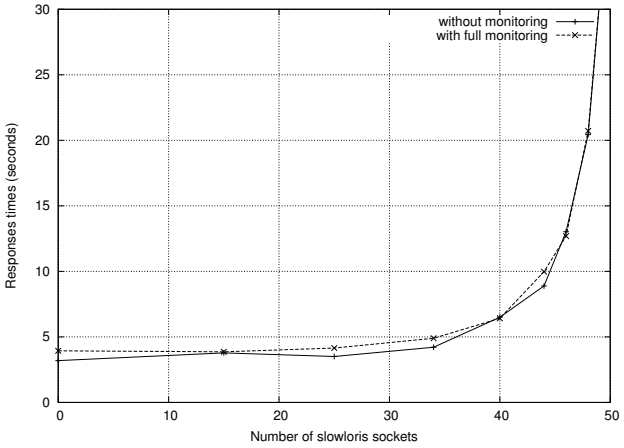


Fig. 10. Response time for 15 simultaneous clients according to the number of connections handled by the client running Slowloris

To handle this attack, we specify a constraint to the model: *"A client must not use more than 10 working threads"*. This constraint is checked at runtime thanks to a *resource bounds observer* (see Section IV-A) directly plugged into the reference behavior model associated with `nullHTTPd`. This observer checks for the number of working threads dedicated to each client.

Like many other servers, `nullHTTPd` uses a main thread that waits for requests on a `accept` instruction. For each request, it delegates a new thread (*via* the `pthread_create` system call). Once the request is finished, this new thread closes the communication socket with its client (using the `close` system call) and dies. Thus, the observer must *acquire* a new occurrence of the resource for each new connection (*i.e.* increment the counter associated with the observer), and *release* it (*i.e.* decrements the counter) when the thread closes its socket.

This information gives us the *observable points* in the source code. Thanks to links between the code and the model, we can attach the observer to the model. Figure 11 shows a part of the reference behavior model and its connection to the observer. The central place (**Connect_Obs**) is the core of the observer. It contains tokens ⟨**id,value**⟩ where **id** is the identifier of the client which makes the request and **value** the number of threads already dedicated to it. When the **accept** transition is fired, the token dedicated to the client is fetched from the observer place and its value is incremented. On the contrary, when the **close** transition is fired, the value of the token associated with the client is decremented.

Let us note that, so far, the observer place is initially marked with one token per color domain identifying clients of the server (with the associated **value** set with 0). This solution should be replaced by a more elegant one where tokens are created when new clients are treated. This is not the case yet due to current implementation constraints.

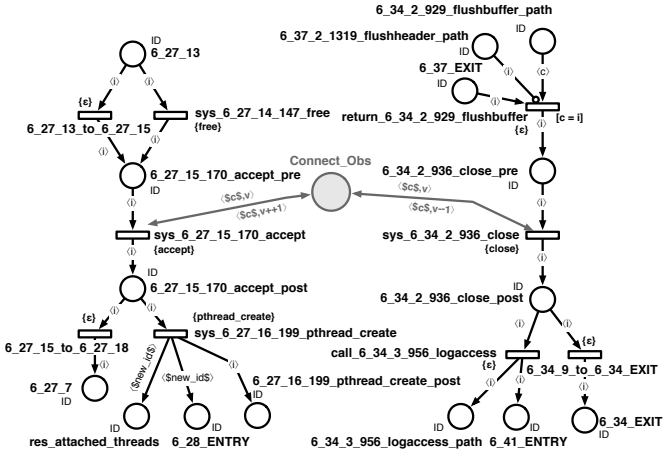


Fig. 11. Resource bounds observer that limit the number of working threads allowed for one client. Only the relevant part of the Petri net describing the reference behavior model is shown here

In the standard implementation of observers in our IDS, the program is stopped once the limit is exceeded. But in our case, if a client surpasses the limit of 10 working threads, we don't want to stop the server (it would be ease the attacker's work). So, we propose a dedicated implementation described below.

In this new implementation of our IDS, if the `fire` function indicates that bound is exceeded, the caught system call is not propagated to the underlying system, and the return value is modified. More specifically, in Listing 3 line 12 is not executed, and the variable `ret` is set to `-1`. Hence, the `accept` call returns an error code for this client. The observer returns a valid value for other clients to allow the `accept` call.

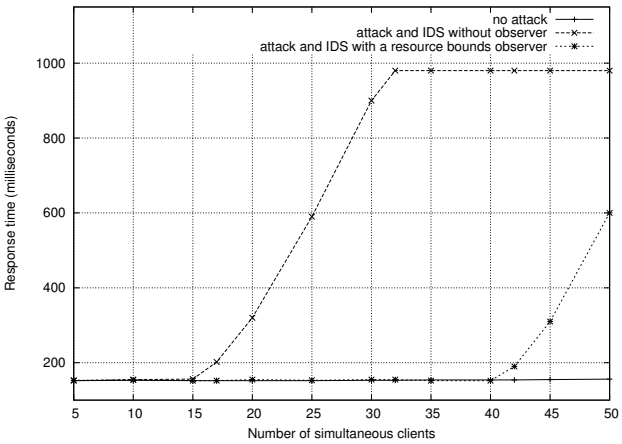


Fig. 12. Response time considering one malicious client handling 35 connections and according to the number of simultaneous clients.

The execution results of this modified IDS are shown in Figure 12. In this experiment, a client executes Slowloris and tries to handle 35 connections. The first set of results represents the server response time under normal conditions (no attack). The second one shows that the server protected by the IDS without an observer cannot handle requests coming from more than 32 clients (the DoS is thus effective). Finally, the last curve shows that an IDS with an observer correctly set up is able to limit the impact of a malicious client. Requests are slow down (from 40 simultaneous clients), but the server is still available.

VI. Conclusion

This paper presents an automatic way to generate dedicated IDS from program sources. To do so, we use a Petri net based representation of the program behavior obtained from static analysis of the eCFG produced by GCC.

Our solution brings several advantages:

1) the process is automatic and thus, IDS can be built more rapidly than in current solutions as soon as a correction is applied on a program to be monitored;
2) the notion of perspective allows to select a set of actions to be monitored in the program, based on a dictionary of symbols (*e.g.* system calls), it is easy to select the set of events to be monitored and thus collect relevant information only from source code;
3) thanks to the use of Petri nets, our representation of the program behavior is much more compact than current approaches and thus allow us to tackle larger programs with a good precision;
4) the generated behavior reference model can be enriched with observers to precisely check for resource usage or time constraints, thus adding precision to the monitoring;
5) the generated IDS are quite small compared to traditional ones and thus, they introduce a very low execution overhead.

Points 1 and 4 allow us to deal with so called 0-day attacks.

A prototype of our IDS generator is implemented. It has been used to assess the solution against several case studies, including the detection of a recent (June 2009) denial of service attack on a web sever. These experiments show the viability and the efficiency of our solution against real programs.

Another advantage (not yet exploited) is that the reference behavioral model could be exploited by verification tools (at least for reasonable size programs), thus performing so-called *off-line* analysis, in a similar way than the one performed with SPIN [21].

Another future work include the management of stack configuration during execution, to enable a finer detection of mimicry attacks [15].

References

[1] N. Delgado, A. Quiroz, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *Software Engineering, IEEE Transactions on*, vol. 30, no. 12, pp. 859–872, 2004.

[2] Z. Liu, S. M. Bridges, and R. B. Vaughn, "Combining static analysis and dynamic learning to build accurate intrusion detection models," in *IWIA '05: Proceedings of the 3rd IEEE International Workshop on Information Assurance*. IEEE Comp. Soc., March 2005, pp. 164–177.

[3] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection," in *RAID '05: Proceedings of the 8 th International Symposium on Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, A. Valdes and D. Zamboni, Eds., vol. 3858. Springer, September 2005, pp. 185–206.

[4] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, V. Atluri, Ed. ACM, November 2002, pp. 255–264.

[5] A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning program behavior profiles for intrusion detection." in *Workshop on Intrusion Detection and Network Monitoring*. USENIX, April 1999, pp. 51–62.

[6] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou, "Specification-based anomaly detection: a new approach for detecting network intrusions." in *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, V. Atluri, Ed. Washington, DC, USA: ACM, 2002, pp. 265–274.

[7] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2001, pp. 156–169.

[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*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–128.

[9] M. A. McQueen, T. A. McQueen, W. F. Boyer, and M. R. Chaffin, "Empirical estimates and observations of 0day vulnerabilities," in *HICSS '09: Proceedings of the 42nd Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, January 2009, pp. 1–12.

[10] J.-B. Voron and F. Kordon, "Transforming Sources to Petri Nets : A Way to Analyze Execution of Parallel Programs," in *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems*. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, March 2008, pp. 1–10.

[11] D. Cahill, "Null httpd," 2002. [Online]. Available: http://nullhttpd.sourceforge.net/httpd/

[12] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Verlag, June 2009.

[13] C. Girault and R. Valk, *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[14] S. Haddad and J.-F. Pradat-Peyre, "New efficient Petri nets reductions for parallel programs verification," *Parallel Processing Letters*, vol. 16, no. 1, pp. 101–116, Mar. 2006.

[15] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, March 2008, pp. 156–167.

[16] B. Alpern and F. B. Schneider, "Verifying temporal properties without temporal logic," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 147–167, 1989.

[17] P. M. Merlin, "A study of the recoverability of computing systems." Ph.D. dissertation, University of California, Irvine, 1974.

[18] J. T. Giffin, "Model-based intrusion detection system design and evaluation," Ph.D. dissertation, Madison, WI, USA, 2006.

[19] RSnake. (2009) Slowloris HTTP DoS. [Online]. Available: http://ha.ckers.org/slowloris/

[20] Apache Software Foundation. Apache HTTP Server Project. [Online]. Available: http://httpd.apache.org/

[21] A. Zaks and R. Joshi, "Verifying multi-threaded c programs with spin," in *Model Checking Software, 15th International SPIN Workshop*, ser. LNCS, vol. 5156. Springer Verlag, 2008, pp. 325–342.