A Software Architecture to support Misuse Intrusion Detection.*

Technical Report CSD–TR–95–009

Sandeep Kumar Eugene H. Spafford

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{kumar,spaf}@cs.purdue.edu

March 17, 1995

Abstract

Misuse Intrusion Detection has traditionally been understood in the literature as the detection of specific, precisely representable techniques of computer system abuse. Pattern matching is well disposed to the representation and detection of such abuse. Each specific method of abuse can be represented as a pattern and many of these can be matched simultaneously against the audit logs generated by the OS kernel. Using relatively high level patterns to specify computer system abuse relieves the pattern writer from having to understand and encode the intricacies of pattern matching into a misuse detector. Patterns represent a declarative way of specifying what needs to be detected, instead of specifying how it should be detected. We have devised a model of matching based on Colored Petri Nets specifically targeted for misuse intrusion detection. In this paper we present a software architecture for structuring a pattern matching solution to misuse intrusion detection. In the context of an object oriented prototype implementation we describe the abstract classes encapsulating generic functionality and the inter-relationships between the classes.

1 Introduction

Intrusion Detection is an important monitoring technique in computer security aimed at the detection of security breaches that cannot be easily prevented by access and information flow control techniques. These breaches can be a result of software bugs, failure of the authentication module, improper computer system administration, etc. Intrusion detection has historically been studied as two sub-topics: *anomaly detection* and *misuse detection*. Anomaly detection is based on the premise that many intrusions appear as anomalies on ordinary or specially devised computer system performance metrics such as I/O activity, CPU usage, etc. By maintaining profiles of these metrics

*This work was funded by the Division of INFOSEC Computer Science, Department of Defense.

for different subject classes, for example individual users, groups of users or programs and monitoring for large variations on them, many intrusions can be detected. Misuse intrusion detection has traditionally been understood in the literature as the detection of specific, precisely representable techniques of computer system abuse. For example, the detection of the Internet worm attack by monitoring for its exploitation of the `fingerd` and `sendmail` bugs [Spa89] would fall under misuse detection.

Several approaches to misuse detection have been tried in the past. They include language based approaches to represent and detect intrusions [HCMM92], developing an API[1] for the same [Sma95], expert systems [SSHW88, Sma88, BK88] and high level state machines to encode and match signatures [Ilg92, PK92]. We proposed using a pattern matching approach to the representation and detection of intrusion signatures [KS94c]. This approach resulted from a study of a large number of common intrusions with the aim of representing them as signatures [KS94a]. The signatures were then classified into categories based on their theoretical tractability of detection. We consider the following to be unique advantages specific to our model of pattern represention and matching.

- Sequencing and partial order constraints on events can be represented in a direct declarative manner. Systems that use expert system rules to encode misuse activity only do so indirectly because it is hard or inefficient to specify temporal relationships between facts in rule antecedents. [Ilg92, PK92] permit the specification of state transition diagrams to represent misuse activity but their transition events are high level actions that do not directly correspond to system generated events. ASAX [HCMM92] is the closest to our approach but ASAX is less declarative. In specifying patterns in their rule based language RUSSELL one must explicitly encode the order of rules that are triggered at every step. While [HCMM92] tends to be a mechanism for general purpose audit trail analysis, our effort is a combination of mechanism and policy. The features provided in our work are closely tied to the intrusion characteristics we are trying to detect.
- Our model provides for a fine grained specification of a successful match. The use of pattern invariants (to be explained later) allows the pattern writer to encode patterns that do not need to rely on primitives built into the matching procedure to manage the matching, for example to clean up partial matches once it is determined that they will never match. This frees the matching subsystem from having to provide a complete set of such primitives and, in the process, tying the semantics of pattern matching with the semantics of the primitives.

Our method also has the following benefits but these are not necessarily a consequence of our approach.

Portability. Intrusion signatures can be moved across sites without rewriting them to accommodate fine differences in each vendor's implementation of the audit trail. Because pattern specifications are declarative, a standardized representation of patterns enables them to be exchanged between users running variants of the same flavor of operating system, with varying audit trail formats.

[1] Application Programming Interface, i.e. a set of library function calls employed for representing and detecting intrusions.

Declarative Specification. Patterns representing intrusion signatures can be specified by defining what needs to be matched, not how it is matched. That is, the pattern is not encoded by the signature writer as code that explicity performs the matching. This cleanly separates the matching from the specification of what needs to be matched.

In this paper we describe our implementation of the model in [KS94c]. Although we have used a popular object-oriented programming language (C++) for our effort, the technique does not require it. Our implementation is directed at providing a set of integrated classes that can be used in an application program to implement a generic misuse intrusion detector. Our implementation also suggests a structure of classes encapsulating generic functionality and the inter-relationships between the classes to design any misuse detector. The paper also describes that structure.

The implementation is ongoing and measurements of its speed and resource requirements will appear in a later paper. Our choice of the language was dictated by the free availability of high quality implementations of the language, our familiarity with it and the linguistic support provided in it to write modular programs. The set of integrated classes we have developed can be programmed in many other Object Oriented languages as well because no properties specific to C++ have been assumed or used. We only exploit the language's encapsulation and data abstraction properties. We use the word *class* in a generic sense and the corresponding notion from many other languages can be substituted here.

2 Our Approach

The model of pattern representation and detection on which the implementation is based, and its theoretical properties were first described in [KS94b] and later refined in [KS94c]. Briefly, each intrusion signature is represented as a specialized graph in this model. We have used the example of detecting TCP connections using IP datagrams to illustrate the various elements of our approach [see fig. 1]. The appendix outlines examples of system vulnerabilities and their representation in this model that would be more suitable for detection using a C2[2] audit trail. These graphs are an adaptation of Colored Petri Nets [Jen92] with guards defining the context in which signatures are considered matched. Vertices in the graph represent system states. The pattern represents the sequence of events and its context that forms the core of a successful intrusion or its attempt. Patterns may have pre-conditions and post-actions associated with them. A pattern pre-condition is a logical expression that is evaluated at the time the pattern springs into existence. It can also be used to set up state that may be used later by the pattern. Post-actions are performed whenever the pattern is matched successfully. For example, it might be desirable to raise the audit level of a user if he fails a certain number of login attempts within a specified time duration. This is easily expressed as a post-action. Patterns may also include invariants to specify that another specified pattern specification cannot appear in the input stream while the main pattern is being matched. If a pattern is regarded as a set of event sequences P that it matches, and an invariant is regarded as another set of event sequences I that the invariant matches, then a pattern with an invariant specification corresponds to the set $P \wedge \overline{I}$. A pattern can have more than one invariant. That

[2] A DoD security evaluation criteria class requiring auditing and unavailability of encrypted passwords [oDS85].

corresponds to $P \wedge \overline{I_1} \wedge \cdots \wedge \overline{I_n}$. For example, a pattern that matches process startups and records all file accesses by the process may require an invariant that ensures that the process has not exited while the principal pattern is being matched. From the practical viewpoint of specifying intrusion patterns, invariants usually result in more efficient matching rather than adding functionality to the pattern specification.

As a concrete example of a pattern, consider the monitoring of TCP connections on a fast gateway by examining each packet that passes through it. The example highlights how stateful matching involving a sequence of events can be represented in our model. The example is drawn from a familiar domain and illustrates that the model is independent of the nature of the underlying events.

A TCP connection setup between the initiator `S` and the recipient `D` involves a three-way handshake [Com91]. The first segment of the handshake involves sending an IP datagram from `S` to `D` with the SYN bit set in the code field. In response to this SYN packet `D` sends a datagram that acknowledges the SYN packet and sets the SYN bit to continue the handshake. The final message is the acknowledgement of the second SYN and is sent from `S` to `D`.

Thus, in order to detect simplified TCP connections not involving retransmissions we can monitor for the sequence:

1. A SYN packet, from a source `S` to a destination `D`.
2. A SYN+ACK, from `D` back to `S`.
3. An ACK, from `S` to `D`.

Pictorially this looks like:

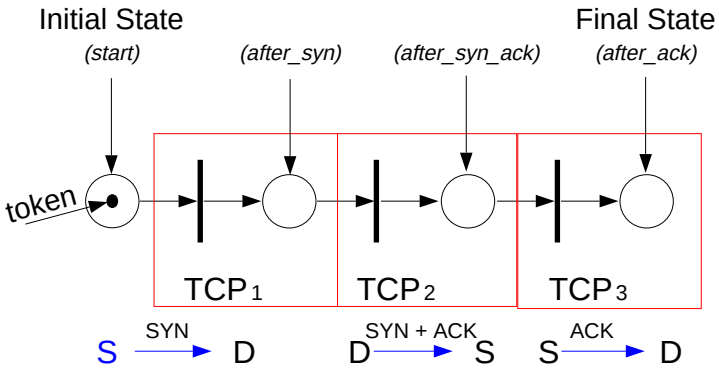


Figure 1: Matching a TCP connection

`Pre-condition`: none
`Guards`:

$\texttt{TCP}_1$: this[SYN] = 1 && (FROM_PORT = this[FROM_PORT]) && (this[TO_PORT] = RLOGIN_PORT) && (FROM_HOST = this[FROM_HOST]) && (TO_HOST = this[TO_HOST])

(If this packet is a SYN packet destined to the RLOGIN port, store its source and destination host and source port in the token).

`TCP`$_2$: this[SYN] = 1 && (this[FROM_PORT] = RLOGIN_PORT) && (this[TO_PORT] = FROM_PORT) && (this[FROM_HOST] = TO_HOST) && (this[TO_HOST] = FROM_HOST)

(If this packet is a SYN packet from the RLOGIN port of a host whose name matches that stored in the token, destined to the host and port corresponding to this token's variables FROM_HOST and FROM_PORT then fire the transition).

`TCP`$_3$: this[SYN] = 0 && (this[FROM_PORT] = FROM_PORT) && (this[TO_PORT] = RLOGIN_PORT) && (this[FROM_HOST] = FROM_HOST) && (this[TO_HOST] = TO_HOST)

(Any non SYN packet flows from (FROM_HOST, FROM_PORT) to (TO_HOST, TO_PORT)).

`Invariant:` No RESET flows from D $\rightarrow$ S.
`Post Action:` Print that a TCP connection has been established between S & D.

Transitions are labeled by event types, in fig. 1 all transitions are labeled with the type *TCP*. This means that the transition can potentially be triggered by a TCP packet. Tokens have state associated with them, often used to store data from events. In the example these are `FROM_PORT`, `TO_PORT`, `FROM_HOST` and `TO_HOST`. The pattern is considered matched when a token reaches the final state. In order for a transition to trigger there must be a token in its input state that satisfies the guard at the transition. The guards placed at each transition are shown above. Guard expressions permit the extraction of data from events for later use or allow the constraining of events that can match a transition. Expressions involving `this` use the array indexing operator [] found in many programming languages to refer to data from the current packet for the transition. Each packet is converted into an object and its *public* member functions can be called using the `this[pub_mem_fun]` syntax to extract data from it. In our example, the pattern guards make use of the member functions SYN() to test whether the TCP packet has the SYN bit set, FROM_PORT() to retrieve the source port from the packet etc. `RLOGIN_PORT` is a global variable defined outside the pattern definition. For a more detailed description of the syntax and use of expressions see [KS94b]. This paper assumes the suitability of the model to misuse intrusion detection. That justification was done in [KS95].

Given the premise that patterns conforming to the model need to be represented and matched by applications, the implementation of this model can be broken down into the following sub-problems:

1. The external representation of patterns. That is, how does the pattern writer encode patterns for use in matching.
2. The interface to the event source. In our example it would be the interface to IP datagrams.
3. Dispatching the events to the patterns and the matching algorithms used for matching.

These issues are discussed in the next section. In addition to solving these requirements, our implementation is designed to simplify the incorporation of the following:

- The ability to create patterns and to destroy them dynamically, as matching proceeds.
- The ability to partition and distribute patterns across different machines for improving per-

formance.

- The ability to prioritize matching of some patterns over others.
- The ability to handle multiple event streams within the same detector without the need to coalesce the event streams into a single event stream.

We describe our design in the next section and show how the library classes implement the design.

3 Overall Architecture

The library consists of several classes, each encapsulating a logically different functionality. An application program that uses the library includes appropriate header files and links in the library.

The external representation of patterns (sub-problem 1) is done using a straightforward representation syntax that directly reflects the structure of their graph. These specifications can be stored in a file or maintained as program strings. When a pattern is needed to be matched in an application, a library provided routine (a class member function) is called that compiles the pattern description to generate code that realizes the pattern. This code is then dynamically linked to the application program and the pattern matching for that pattern is initiated. The application also instantiates a server for each type of event stream used for matching. Events are totally encapsulated inside the server object (sub-problem 2) and are only used inside pattern descriptions. As pattern descriptions are compiled they are added to the relevant server queue. The server accesses and dispatches events to the patterns on its queue in some policy specifiable order (sub-problem 3).

The application structure is explained below which gives an overall view of the application. Section 3.2 looks at the structure of events. Section 3.3 explains the structure of the server itself in detail and its relationship to the patterns that are instantiated by the application.

3.1 Application Structure

As an example application structure consider matching the pattern described in fig. 1. This may look as shown below. The function `dotted_decimal_addr` takes an integer (32 bits) and prints it as four integers corresponding to each octet of the integer, separated by a dot. This is a common way of writing internet host addresses.

```
//file application.C
1    #include "IP_Server.h"
2
3    int RLOGIN_PORT = 513;
4
5    int print_tcp_conn(int from_HOST, int to_HOST) //callback function
6    {
7       cerr << "A TCP connection has been established between "
8            << dotted_decimal_addr(from_HOST) << " and "
9            << dotted_decimal_addr(to_HOST)   << endl;
10      return 1;
11   }
```

```
12
13   int main()
14   {
15      IP_Server S;
16      IP_Pattern *p1 = S.parse_file("patterns-ip"); //read pattern from "patterns-ip"
17
18      /* duplicate a thread of control if necessary because run() doesn't return */
19      S.run();
20
21      return(1);
22   }
```

The application program makes use of an `IP_Server` object. The server object understands the layout of events and the event types that can be legally used in a pattern definition. `IP_Server` also knows how to access events, in this case from the machine's network interface, and how to dispatch them to the patterns that are registered with it. The server is also responsible for parsing pattern descriptions and can type-check the pattern specification because it understands the data format of the events. The call to the server member function `parse_file` reads, compiles and registers a new pattern with the server object. When the server object is started with a call to `S.run()` it starts reading events and dispatching them. This consumes one thread of control as `S.run()` never returns. The server is responsible for implementing concurrency control methods to ensure that calls to its public member functions do not corrupt its internal state. Our implementation uses the idea of monitors [Hoa74] to ensure this. The pattern description contained in file *patterns-ip* looks like:

```
//file patterns-ip
1    extern int RLOGIN_PORT_CLIENT, RLOGIN_PORT_SERV, print_tcp_conn(int, int);
2
3    pattern TCP_Conn_Mon "Monitor rlogin connections" priority 10
4      int FROM_PORT, FROM_HOST;
5      int TO_PORT, TO_HOST;
```

The variable declarations define the color of the tokens in the pattern. Each token has four integers that can be accessed through the syntax `this[FROM_PORT]`, `this[FROM_HOST]` etc.

```
6      state start;
7      nodup state after_syn, after_syn_ack;
8      state after_ack;
```

These are the states of the pattern. `after_syn` signifies the state after the initial SYN is observed, `after_syn_ack` signifies the observation of the initial SYN followed by a response SYN. `nodup` indicates that tokens in this state will not be duplicated to other states, rather they will be moved to other states when the transition fires.

```
9      post_action { print_tcp_conn(FROM_HOST, TO_HOST); }
```

`print_tcp_conn` is called with token values corresponding to the token in the final state of the pattern.

```
10     neg invariant first_inv
11       state inv_start, inv_final;
12
```

```
13        trans rst(TCP)
14          <- inv_start;
15          -> inv_final;
16          |_ { this[RST] = 1 &&  TO_HOST = this[FROM_HOST] && this[TO_HOST] = FROM_HOST; }
17        end rst;
18      end first_inv
```

The invariant specifies that no reset should be received during connection formation. An invariant specification can itself be a graph. Whenever a token is moved from the start state of the pattern, its copy is placed in the start state of the invariant. This token can have part of its color defined because the firing of a transition may change a token color.

```
19      trans tcp_1(TCP) /* TCP is the event type of the transition */
20        <- start;
21        -> after_syn;
22        |_ { this[SYN] = 1 &&  this[ACK] = 0 &&
23             FROM_PORT = this[FROM_PORT] && this[TO_PORT] = RLOGIN_PORT_SERV &&
24             FROM_HOST = this[FROM_HOST] && TO_HOST = this[TO_HOST];
25           }
26      end tcp_1;
27
28      trans tcp_2(TCP)
29        <- after_syn;
30        -> after_syn_ack;
31        |_ { this[SYN] = 1 && this[ACK] = 1 &&
32             (this[FROM_PORT] = RLOGIN_PORT_SERV) && (this[TO_PORT] = FROM_PORT) &&
33             (this[FROM_HOST] = TO_HOST) && (this[TO_HOST] = FROM_HOST);
34           }
35      end tcp_2;
36
37      trans tcp_3(TCP)
38        <- after_syn_ack;
39        -> after_ack;
40        |_ { this[SYN] = 0 && this[ACK] = 1 &&
41             (this[FROM_PORT] = FROM_PORT) && (this[TO_PORT] = RLOGIN_PORT_SERV) &&
42             (this[FROM_HOST] = FROM_HOST) && (this[TO_HOST] = TO_HOST);
43           }
44      end tcp_3;
```

This defines the structure of the pattern graph.

```
45    end TCP_Conn_Mon;
```

Listing 1: A Sample Pattern Description

Similarly, if an application needed to match patterns against a C2 audit trail it might have used a `C2_Server` instead of `IP_Server` or concurrently with it within the same application program.

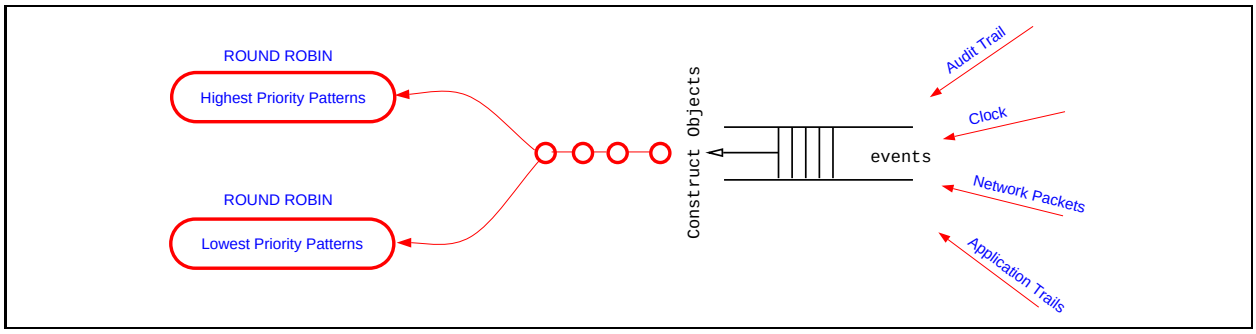
3.2 Event Structure

Each event in the event stream is converted to an instance of an event class. For handling IP datagrams this class might be named `IP_event`. This class encapsulates all the attributes common to IP datagrams. Derived classes of `IP_event` can be used for specifying more specialized types of IP datagrams. For example, `TCP_event` and `UDP_event` can be derived to represent TCP and UDP datagrams. Each event object can identify its type through its `type()` member function. This is used by the server to dispatch the event to the appropriate patterns. All the data belonging to the event is made available through its member functions. This mechanism encapsulates the organization of data in the event which may be system dependent in general. The description of all the event classes is what constitutes the backend of the system and is one of the few system dependent layers.

3.3 Server Structure

For each event, the server looks at its type and consults a dynamically maintained table of patterns that have requested events of that type. It then calls the `Patproc` procedure of each such pattern. `Patproc` is a procedure associated with every pattern (its member function) that handles events for it. This approach to handling events is similar to the approach taken in Microsoft Windows [Pet92]. Events of interest are requested by patterns when they are instantiated by the server.

Events are dispatched to patterns based on the priority of the pattern. Patterns are placed in queues at the appropriate priority level, and patterns are serviced in each queue in a round robin fashion. This ordering of patterns by priority assumes that on the average, an event can be dispatched to all the patterns requesting it in a time less than the average time of generation of an event. If this requirement is not met, patterns up to a certain level in priority may be perpetually starved. We have not provided for any aging of patterns in this design, such mechanisms can be added. Pictorially this looks like



3.4 Summary

The use of an event stream requires the creation of two classes. An event class that is the root class of all events provided in the event stream and a server class that parses pattern descriptions, instantiates them and manages them on its data structures. The server class interacts with the event class by converting raw events into objects of this class and dispatching them. The inter-

relationship between the various classes is shown in fig. 2. Class names bounded by dotted boxes are abstract classes. The functions identified within these boxes are the pure virtual functions of these classes.

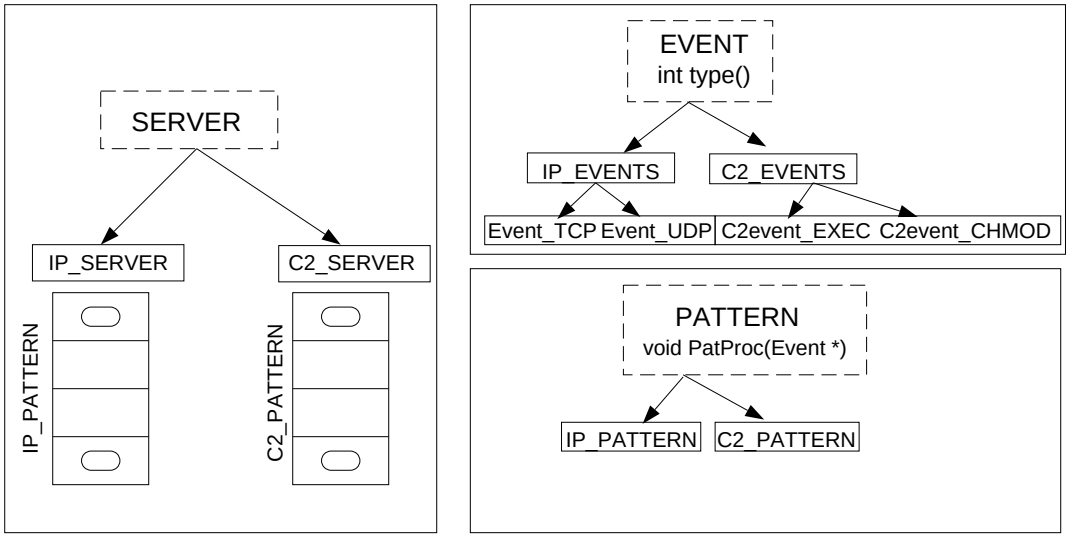


Figure 2: Inter-relationship between the various classes in the detector.

Use of pattern and application global variables discourages parallelism in exercising several tokens simultaneously when several multi-processor threads are available. Several available threads, can, however, simultaneously exercise tokens in different patterns. Application and pattern global variables should ideally become read-only once matching begins so that concurrency of access to these variables is possible.

4 Implementing the Server

This section describes how a server class (e.g. IP_Server) is implemented in our library. The event class associated with the server class is completely encapsulated in the server class and is not visible to the application. The heart of the server class is the member function that translates a pattern description into C++ code that implements the pattern [sec. 4.1]. Because our language for describing patterns is a straightforward representation of the pattern structure, translation into an automaton is direct. Syntactic structures introduced in the language often translates directly into functions that are invoked to perform the operation. Sec. 4.2 describes what the translated automaton looks like, particularly the procedure that accepts incoming events from the server and exercises the automaton with it.

4.1 Server::parse()

The server class associated with each event stream is responsible for translating patterns specific to the event stream. For each pattern (each pattern name is unique), the translation performs the

following actions:

1. It generates a C++ class representing the pattern (`IP_TCP_Conn_Mon` in our example) with all the pattern global variables as static data members of the class (none in our example).
2. It generates a token class (`IP_TCP_Conn_Mon_Tok`) that represents tokens associated with that pattern). The token class has private data members corresponding to each pattern local variable and corresponding public functions to access them. In our example these are `FROM_PORT`, `FROM_HOST`, `TO_PORT` and `TO_HOST`. These were declared in lines 4 and 5 of listing 1.
3. Each guard expression associated with a transition is re-written with several syntactic changes:
   - Pattern local variable references are substituted by calls to token member functions.
   - Certain operations are syntactically changed to library calls, for example, the pattern matching operator `=~` is changed to a call to a regular expression matching routine.
   - Calls of the form `this[...]` are changed to member function calls to the appropriate event object. See for example line 22 of listing 1.
   - Pattern global variable references are changed to Pattern static references.
4. A `PatProc` procedure is generated for the pattern to handle events for the pattern, in our example its signature would be `IP_TCP_Conn_Mon::PatProc(IP_Event *)`.

4.2 Pseudo code for the generated PatProc

The heart of a pattern is its `PatProc` which exercises its automaton on each event that the pattern has requested. Fig. 3 shows the pseudo code of a sample PatProc. For each incoming event, all transitions labeled with that type are tested to see if they fire, i.e. whether the event and the unified token formed by unifying tokens drawn from each input state of the transition satisfy the guard at the transition. All tokens residing in *nodup* states that comprise the unified token are marked for later deletion. Tokens that are added to output states of a transition as a result of its successful firing wait to be added to the states until all transitions have been tried. Then the tokens are added into all the states. When an invariant is satisfied, i.e. a token reaches the final state of the invariant, all the tokens related to the token are destroyed.

```
IP_TCP_Conn_Mon::PatProc(Event *e)
{
   for (all transitions in pattern and invariants of type e->type())
   {
      for (all token sets formed by taking one token from each input of this transition)
      {
         if (the token set does not unify) continue;
         if (the token set fails the guard) continue;
         mark all tokens in this set belonging to nodup states for later deletion;
         put a copy of this token in each successor of this transition;
         if (one of the input states of this transition is a pattern start state)
            put a copy of this token in the start state of each invariant;
      }
   }

   clock the states to merge tokens at its input with tokens already in the state;

   eval post actions for all tokens in the final state and free them;

   delete all marked tokens from all nodup states;
   process all invariant final states;
}
```

Figure 3: Pseudo code of a sample `PatProc`.

5 Design Choices

By far the most significant consideration guiding the design was the run-time efficiency of the detector. For misuse detection using a C2 generated audit trail one might reasonably expect to process events (audit records) at the rate of 50K-500K/user/day [Sma95]. Furthermore, any computer resource required for matching patterns reduces the availability of these resources for general use. We therefore decided not to interpret the pattern automata by using table lookups to determine the pattern structure but instead to compile the pattern description into an automaton. This also has the benefit of compile time optimizations of guard expressions present in the pattern. As the generated code realizing the automaton did not need to be "user friendly", we tried to make it more efficient by using functions as little as possible to avoid function call overhead in cases where functions could not be inlined. This often meant that data structures manipulated by the various pieces of the generated automaton were not encapsulated and were manipulated directly by these pieces. This has not proved to be a problem as the routines that generate this "program" are structured and the generated program logic can be deciphered by following the structure and logic of the generating routine.

The overriding constraint of efficiency combined with the requirement to dynamically create and destroy patterns meant that automaton descriptions be compiled and dynamically linked for the purpose of matching. An additional benefit of the dynamic creation of patterns is that new patterns can be created within an executing program based on its logic and execution flow. For example, it might be desirable to instantiate specific patterns for matching based on the type and degree of

suspicious activity observed. Such patterns may depend on the particular user and other specifics of the suspicious activity.

Our design, which is based on the model of dispatching events to patterns lends itself naturally for distribution. In a distributed design, the event sources (audit trails) may be generated on different machines and their processing on another machine. That is, the patterns, the server and the event sources may all reside on physically different machines. The server can then retrieve events by using any of several well known techniques [BN84, Par90] and dispatch them to patterns. Although our current implementation is single host based, a distributed implementation should be straightforward.

Our current implementation precludes concurrency of exercising a pattern with several events simultaneously because of order of execution guarantees that can be made in a single threaded architecture. We do not consider that to be a hard limitation on the design because concurrency can be exploited by exercising more than one pattern on the same event. In a system where the expected number of patterns are in the order of a hundred, this does not seem to be a stringent limitation.

A limitation of the current design is that patterns cannot directly use more than one event source. In order to use more than one event source, the disparate sources would have to be canonicalized to one event stream and used in the patterns. Many modern audit trails, for example the SUN BSM, allow the creation of user defined event types and applications can generate their own specific audit records through an API.

6 Performance

We have built and tested a prototype `IP_Server` that puts the network interface of a machine running Solaris 2.3 into promiscuous mode and dispatches IP datagrams to its patterns. We are currently in the final stages of the implementation of a `C2_Server` that dispatches C2 audit trail events on a Solaris 2.3 machine running the Basic Security Module. Signatures will be written for vulnerability data drawn from COPS [FS91], CERT advisories [CER] and the bugtraq and 8lgm[3] electronic mailing lists. Performance figures for that will be reported in a subsequent paper.

7 Summary

This paper described a possible architecture for structuring a misuse intrusion detector based on pattern matching. The structure is client-server based in which the server obtains events and dispatches them to clients (patterns) which implement the matching procedure specific to their structure. Implementing this structure as a library permits embedding this type of matching within application programs. Our prototype allows the dynamic creation of patterns. These patterns are translated from a description language into C++ code that implements the pattern and dynamically

[3]Both lists discuss computer security vulnerabilities, their exploitation and steps for prevention and detection. Bugtraq is issued from `bugtraq@crimelab.com` and 8lgm advisories can be retrieved from `fileserv@bagpuss.demon.co.uk`.

links that code into the application.

8 Acknowledgements

We would like to thank all members of the COAST laboratory for their valuable comments on this paper, in particular Christoph Schuba for his extra effort and help with the paper.

References

[BK88] David S. Bauer and Michael E. Koblentz. NIDX – An Expert System for Real-Time Network Intrusion Detection. In *Proceedings – Computer Networking Symposium*, pages 98–106. IEEE, New York, NY, April 1988.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CER] Available by anonymous ftp from cert.sei.cmu.edu:/pub/cert_advisories.

[Com91] Douglas E. Comer. *Internetworking with TCP/IP*, volume I. Prentice Hall, 2nd edition, 1991.

[FS91] Daniel Farmer and Eugene Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Purdue University, Department of Computer Sciences, September 1991.

[HCMM92] Naji Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of ESORICS 92*, Toulouse, France, November 1992.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Ilg92] Koral Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. Master's thesis, Computer Science Department, University of California, Santa Barbara, July 1992.

[Jen92] Kurt Jensen. *Coloured Petri Nets – Basic Concepts I.* Springer Verlag, 1992.

[KS94a] Sandeep Kumar and Eugene Spafford. A Taxonomy of Common Computer Security Vulnerabilities based on their Method of Detection. (unpublished), June 1994.

[KS94b] Sandeep Kumar and Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Purdue University, Department of Computer Sciences, March 1994.

[KS94c] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, October 1994.

[KS95] Sandeep Kumar and Eugene H. Spafford. Misuse Intrusion Detection Viewed as a Pattern Matching Problem. *Journal of Computer Security (to be submitted)*, 1995.

[oDS85] Department of Defense Standard. *Department of Defense Trusted Computer System Evaluation Criteria.* Number DOD 5200.28-STD. U.S. Government Printing Office, December 1985.

[Par90] Graham D. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *USENIX 1990 C++ Conference Proceedings*, pages 37–50, 1990.

[Pet92] Charles Petzold. *Programming Windows 3.1.* Microsoft Press, 1992.

[PK92] Phillip A. Porras and Richard A. Kemmerer. Penetration State Transition Analysis – A Rule-Based Intrusion Detection Approach. In *Eighth Annual Computer Security Applications Conference*, pages 220–229. IEEE Computer Society press, IEEE Computer Society press, November 30 – December 4 1992.

[Sma88] Stephen E. Smaha. Haystack: An Intrusion Detection System. In *Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Tracor Applied Science Inc., Austin, TX, Dec 1988.

[Sma95] Steve Smaha. Talk given at the third Computer Misuse and Anomaly Detection Workshop (CMAD III) in Sonoma, CA, January 1995.

[Spa89] Eugene Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[SSHW88] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert Systems in Intrusion Detection: A Case Study. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

9 Appendix

We give here an example signature pattern that can be used to detect a common vulnerability. This is a possible encoding of the 8lgm advisory of 5/11/94. The advisory deals with a vulnerability in the `passwd` program supplied with the SunOS 4.1.x operating system. This version of `passwd` allowed any user to specify the password file to be used by it. `passwd` updates the file as root. Using a program which changes the absolute path of the supplied file at carefully selected points during the execution of `passwd`, changes can be written at arbitrary places in the file system.

This signature monitors for a process opening a path name twice, once for reading, then for writing in which the object pointed to by the pathname changes. That is:

`open for read link` (link_1)
`open for write link` (link_2)
$\text{link}_1 \neq \text{link}_2$

```
1    extern int inode(str);
2
3    pattern passwd_attack "passwd -F" priority 7
4      state start;
5      nodup state after_read;
6      state after_write;
7      int PID, INODE;
8      str FILE;
9
10     post_action {
11       printf("Pid %d opened different links (same path) for reading & writing %s\n", PID, FILE);
12     }
13
14     neg invariant inv /* negative invariant */
15       state start_inv, final_inv;
16
17       trans proc_exit(EXIT)
18         <- start_inv;
19         -> final_inv;
20         |_ { this[PID] = PID; }
21       end proc_exit;
22     end inv;
```

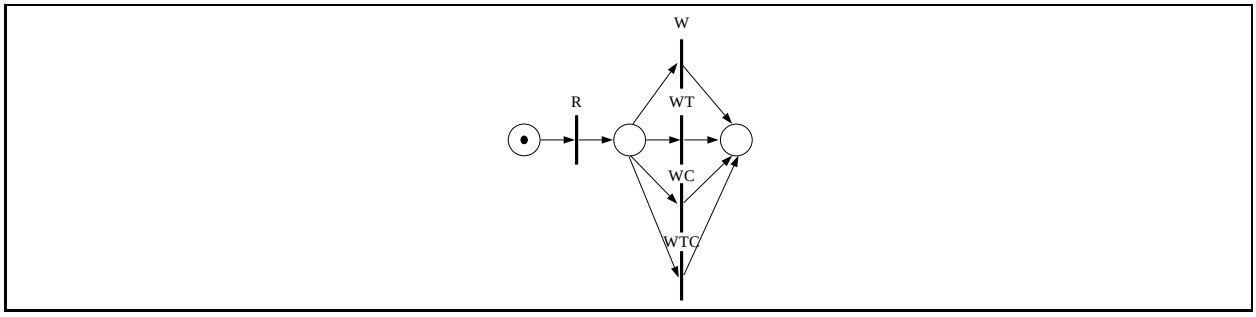
The invariant garbage collects all tokens related to partial matches for a particular process once the process has exited.

```
23     /* pattern description follows */
24     trans read(OPEN_R)
25       <- start;
26       -> after_read;
27       |_ { this[ERR] = 0 && PID = this[PID] && islink(this[OBJ]) &&
28            INODE = inode(this[OBJ]) && FILE = this[OBJ]; }
29     end read;
```

Whenever the process successfully opens a path which is a link, its inode is stored. Link may be a symbolic link or a link to a file object with a reference count > 1.

```
30     trans write1(OPEN_W)
31       <- after_read;
32       -> after_write;
33       |_ { this[ERR] = 0 && PID = this[PID] && islink(this[OBJ]) &&
34            INODE = inode(this[OBJ]) && FILE = this[OBJ];
35          }
36     end write1;
```

The SUN BSM auditing mechanism generates an event for each option combination of the `open` system call. This transition detects the case when an open for read is followed by an open for write with create. The other write transitions following this one detect the case when the open for read is followed by other kinds of open for write. Pictorially this looks like:




```
37      trans write2(OPEN_WC)
38        <- after_read;
39        -> after_write;
40        |_ { this[ERR] = 0 && PID = this[PID] && islink(this[OBJ]) &&
41             INODE = inode(this[OBJ]) && FILE = this[OBJ];
42           }
43      end write2;
44
45      trans write3(OPEN_WT)
46        <- after_read;
47        -> after_write;
48        |_ { this[ERR] = 0 && PID = this[PID] && islink(this[OBJ]) &&
49             INODE = inode(this[OBJ]) && FILE = this[OBJ];
50           }
51      end write3;
52
53      trans write4(OPEN_WTC)
54        <- after_read;
55        -> after_write;
56        |_ { this[ERR] = 0 && PID = this[PID] && islink(this[OBJ]) &&
57             INODE = inode(this[OBJ]) && FILE = this[OBJ];
58           }
59      end write4;
```

Whenever the process later opens the same path for writing and the object represented by the path has changed the pattern is triggered and the `post_action` executed.

```
60    end passwd_attack;
```