A Declarative Approach to Stateful Intrusion Detection and Network Monitoring

M. Couture, B. Ktari and M. Mejri
Département d'informatique et de génie logiciel
Université Laval, Québec
G1K 7P4
Canada
E-Mail: mathieu.couture@ift.ulaval.ca

F. Massicotte
Communications Research Centre
3701, avenue Carling
C.P.11490, succursale H
Ottawa, Ontario, K2H 8S2, Canada
E-Mail: frederic.massicotte@crc.ca

***Abstract*— In this paper we present a new approach to stateful intrusion detection. It is based on a temporal logic which has the capability to express temporary properties, which are properties lying between events. The detection of those events can in turn depend of others temporary properties. The aim of this logic is to model knowledge gathering. It is basically propositional logic, to which we've added a single temporal operator, which allows to define limits of temporary properties.**

I. Introduction

Signature-based intrusion detection systems, such as Snort [1], are useful to detect events without consideration of the context in which they occur. The need to keep track of the context in intrusion detection systems has already been cited [2], [3], [4], [5]. Even in Snort, some work in this direction has been made by the addition of software add-ins to keep track of active TCP sessions, or to detect multiple-packet attacks such as port scanning. Considering the context thus allows a reduction in both false positives and false negatives.

In this paper, we present a new approach that keeps track of the context in which events occur. It is based on a temporal logic that models both events and temporary properties lying between events. Temporary properties can be seen as gathered information which can be used to improve intrusion detection. In section II, we provide a brief review of the state-of-the-art in intrusion detection. In section III, we present the basic properties of a logic that we have developed. In section IV, we show how our logic can express non-trivial properties that are often desirable to perform intrusion detection. In section V, we show how the logic could be implemented in a fully functional language and work jointly with Snort. We conclude in section VI by explaining the current state of implementation and giving some further possibilities.

II. State of the Art

Once the need to express properties that are true for many events over a period of time is identified, we have to choose which paradigm to use as the foundation for the new language we want to develop. The existing work which inspired us could then be divided in two categories: languages specifically designed for intrusion detection and/or system monitoring, and temporal logics issued from model checking theories.

A. Designated Languages

Some languages designed to perform network intrusion on a many-events base exist already. The first we found is Bro [2], working at two different levels, an "event engine", used to get atomic network events, and a "policy script interpreter", to keep track of some state information and generate real time alarms. Although this two level structure could seem similar to the one we are proposing, there is an important difference lying on the fact that policies written with the Bro languages are procedural, not declarative. That is, our objective of avoiding programming is not met here.

With STATL [6] (State Transition Analysis Technique Language), and more specifically NetSTAT [3], a step toward a declarative language is made. STATL is a language which allows evolving attack scenarios to be specified, each step being modelled by an entity called a *state*, linked by *transitions*, which can be activated by events or by timeouts. This language thus allow to keep track of the network state. It is not yet however purely declarative, in the sense that the user still has to specify *how* transitions are performed.

LAMBDA [4] is a truly declarative language. LAMBDA works from events, and allow to specify how those are modifying our knowledge about the network. A LAMBDA specification is then triplet, consisting of a precondition, representing the prior knowledge, a detection part, associated with the event, and a postcondition, representing the updated knowledge. Such basic specifications can be combined together in order to specify full attack scenarios. Temporal properties can be expressed in LAMBDA by making preconditions that are postconditions of another and by comparing the timestamps of different events.

Recently, we've seen how the Chronicles language [5] can be used to correlate low level events in an application to intrusion detection. A very interesting aspect of Chronicles is that it allows the specification of properties based on repetitions, that is, a number of similar low level events can be composed into a single, higher level event. For example, a given amount of TCP connection requests can be merged together as a syn scan. This higher level event can in turn be used by the system, giving to events a recursive nature.

B. Temporal Logic

Temporal logic has been developed to express dynamic properties of programs. It is generally used together with model checking, which is set of techniques to verify, prior to execution, if a program satisfies some given desirable properties. A program is then considered to be a set of variables which can be read and modified by sequences of primitive actions, and temporal logic is used to express properties about performed sequences of actions and traversed memory states.

Temporal logic can be divided into two categories: linear and branching time [7]. Linear temporal logics express properties of single program execution, while branching time logics allow properties to be expressed about different possible executions. Because the situation we are working on is one of a specific execution trace, given by the monitored network, branching time temporal logics are rejected, and we are left with linear temporal logics.

The most popular of those logics is certainly the one called LTL [8], which has been shown to be of a very practical use with the success of SPIN [9]. However, this logic has been developed to work on infinite program executions, and allows properties to be expressed that could not be verified with only a finite prefix, which is the case of interest to us. Actually, in [10], we can find a modification of LTL semantics that allows reasoning about finite program executions. Although the traces we are working on are finite, they are always *on going* which makes this approach inappropriate for us.

III. Information Gathering Logic

The logic we propose is an extension of propositional logic, which allows seeking for past events and defining the start and end points of temporarily true statements. Those statements can be equated to the knowledge we have about the network. We could say that as an alternative to the LAMBDA language, which says how events modify the knowledge, the focus here is put on the knowledge itself, and we specify which events are modifying it. As in the Chronicles language, events have a recursive nature. That is, an event can be either primitive or generated by a combination of other events. Then, there is no conceptual difference here between events and knowledge, since the semantic of the logic, as we will see, is defined relatively to the moment we are consulting it. An event is thus a punctual knowledge. From temporal logics, we keep the fact that consistency of our specifications can be verified, as well as the correctness and the completeness of the algorithm used to verify it. The latter is outside the scope of this paper.

A. Model

The model we are working on is a simple finite sequence of events, which are subsets of a set of propositional constants.

Definition 3.1: Given a set P of propositional constants, an *event* is a subset of P. A *trace*, usually denoted as σ, is a finite sequence of events. By $\sigma(i)$, we denote the i^{th} event of the trace σ.

In the application to network intrusion detection, a trace can be a *traffic* trace, and events the packets of that trace. Propositional constants can be formed from certain fields in packet headers considered relevant for intrusion detection. The TCP `syn` flag is an example of such field of interest.

It is important to notice that some set of propositional constants, of the form $p(v)$ for $v \in \mathcal{V}$ ($\mathcal{V}$ a given set of values), can define predicates over the events of a trace. When, for each $\sigma(i)$, there is a unique $v \in \mathcal{V}$ such that $v \in \sigma(i)$, p can then be seen as a function mapping events of the trace to $\mathcal{V}$. An important such a case, which will always be taken for granted, is the *timestamp* function, noted by τ. We shall also work under the assumption that the τ function is strictly increasing, that is, two events can not occur at the same time, and events are lying on the trace in the same order they occurred. We will see in section IV-A how we could use that function to specify timeout properties.

ϕ	::=	$p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid [\phi_1, \phi_2]$

TABLE I
Syntax.

$HandShake$	:=	$\texttt{ack} \wedge [\texttt{synack} \wedge [\texttt{syn}, \textbf{ff}], \textbf{ff}]$
$Session$	:=	$[HandShake, \texttt{fin} \vee \texttt{rst}]$
$ValidAttack$	:=	$Attack \wedge Session$

Fig. 1. Attack in the context of an active TCP session.

B. Syntax

The syntax of our logic, shown on table I, is as that of propositional logic, excepted that we have added a temporal operator, $[\phi_1, \phi_2]$, used to seek past events over a given trace. The aim of this operator is to represent the fact that some statements are only true for a given period of time. Moreover, this period can be characterized by a beginning and an end, ϕ_1 and ϕ_2. The formula $[\phi_1, \phi_2]$ should be understood as being true *between* ϕ_1 and ϕ_2. If ϕ_2 never shows to be true, then $[\phi_1, \phi_2]$ is true from the time ϕ_1 is so.

An important such a case is $[\phi_1, \textbf{ff}]$, where **ff** is the propositional constant that is never true. The formula $[\phi_1, \textbf{ff}]$ should then be understood as ϕ_1 *has already been true*. We could then specify sequences of events, using formulas of the form $r \wedge [q \wedge [p, \textbf{ff}], \textbf{ff}]$. This formula will be true for events where r is true, but also q has already been true, and p before it. It then specify the sequence of events $p - q - r$.

In figure 1, we show how such a sequencing formula can be used to specify a TCP handshake. We have to read the formula from inside to outside. The first event which has to occur is a `syn` packet. Once this `syn` packet has occurred, we should then wait to see a `synack` packet. The handshake is completed at the moment that the last acknowledgement, `ack`, is sent. The following formula shows how we can use this handshake to specify an active TCP session. It states that a TCP session is active from the time a TCP handshake is completed up to the moment we see a `fin` or a `rst` packet. Finally, we can use this information to validate the occurrence

$\phi_1 \vee \phi_2$	$\stackrel{\text{def}}{=}$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
$\phi_1 \Rightarrow \phi_2$	$\stackrel{\text{def}}{=}$	$\neg\phi_1 \vee \phi_2$
$\phi_1 \Leftrightarrow \phi_2$	$\stackrel{\text{def}}{=}$	$\phi_1 \Rightarrow \phi_2 \wedge \phi_2 \Rightarrow \phi_1$
$\langle\phi_1, \phi_2\rangle$	$\stackrel{\text{def}}{=}$	$\neg[\neg\phi_1, \neg\phi_2]$

TABLE II
DEFINED OPERATORS.

$\sigma(i) \models_\sigma p$	iff	$p \in \sigma(i)$
$\sigma(i) \models_\sigma \neg\phi$	iff	$\sigma \not\models_\sigma \phi$
$\sigma(i) \models_\sigma \phi_1 \wedge \phi_2$	iff	$\sigma(i) \models_\sigma \phi_1$ and $\sigma(i) \models_\sigma \phi_2$
$\sigma(i) \models_\sigma [\phi_1, \phi_2]$	iff	$\exists j.j < i$ and $\sigma(j) \models_\sigma \phi_1$ and $\nexists k.j < k < i.\sigma(k) \models_\sigma \phi_2$

TABLE III
SEMANTICS.

of a given attack lying over TCP, as does the Flow precompiler of Snort. If the formula $Attack$ represents the completion of an attack over TCP, then the formula $Attack \wedge Session$ states that a complete handshake should have occurred before the attack has been seen, but that the session should'nt have been closed by a `rst` or a `fin` packet.

Note that it is possible, as usual, to define $\vee$, $\Rightarrow$ and $\Leftrightarrow$ operators as usual from $\neg$ and $\wedge$, as shown on table II. We have included in this table the dual operator of $[\phi_1, \phi_2]$, noted $\langle\phi_1, \phi_2\rangle$, although we don't see for the moment how useful it could be in the case of intrusion detection.

C. Semantics

The semantics of our logic is given in table III. It shows how to evaluate formulas of our logic over events of a given trace σ. The first line states that a propositional constant is true in events containing it (remember that, by definition, an event is a set of propositional constants). The second line states that the negation of a formula is true when this formula is not true. The third one states that a conjunction of formulas is true when both formulas are true. The fourth line, the most important one, states that a formula of the form $[\phi_1, \phi_2]$ is true from the moment ϕ_1 has been true, unless ϕ_2 has been true since. In that case, we have to wait again for ϕ_1 to be true, and so on.

IV. EXTENDING THE LOGIC

A. Timing-out the Logic

In section III-B, we have studied the case of a TCP handshake. An important failure of the model we gave was to allow the three events constituting it, the `syn`, the `synack` and the `ack`, to be arbitrarily separated in time. That is, the last `ack` could come an hour after the `synack`. In real life, we know that the operating system of the requested host had resigned waiting for this last `ack` a long time ago, and that considering the handshake to have been successful would have been a mistake. Moreover, if this acknowledgement still hasn't occured after a reasonable period of time, we could then think the `syn` had been spoofed [11] and that a scanning attack may be ongoing.

When presenting the model in section III-A, we discussed the existence of a timestamp function, noted τ. We could use that function to specify, for example, that a specific event is occurring at time 3, using the formula $\tau(3)$. More generally, we could specify that an event is occurring at time x with the formula $\tau(x)$. Then, we could state that an assertion is true for 3 units of time after ϕ has occurred using the formula $[\phi \wedge \tau(x), \tau(x+\delta) \wedge \delta > 3]$. We will abbreviate this by $[\phi, 3]$. The formula $q \wedge [p, 3]$ would then be true in states where q is true after p has occurred, and the period of time separating them is not longer than 3 units of time.

We can now give a more accurate description of a TCP handshake using the formula $\texttt{ack} \wedge [\texttt{synack} \wedge [\texttt{syn}, 3], 2]$. It means that the `synack` has to come 3 units of time after the `syn`, and that the `ack` has to come 2 units of time after the `synack`.

B. Expressing Safety Properties

The examples we've seen indicate that our logic can be used to express security properties, that is, to specify that something bad should never occur. But what about safety properties? A property is said to be a safety property if it states that something good must eventually occur [12]. Since we are always looking backward, and some future events may always be coming, this seems to be a much harder task . Actually, the safety properties we will be able to state will be of the form *something good has to occur before something else*. Generally, this something else will be a timeout. Moreover, we may want this good thing to occur only after something else has also occured.

The general form of the formula which will allow us to solve this problem is the following:

$$[\phi_1 \wedge \tau(x), \mathbf{ff}] \wedge \tau(x + \delta_1) \wedge \delta_1 > \delta$$
$$\Rightarrow [\phi_2 \wedge \tau(x + \delta_2), \mathbf{ff}] \wedge \delta_2 < \delta$$

It states that, if a given statement ϕ_1 has been true, and a certain amount of time δ has elapsed since, then, between the moment when ϕ_1 has been true and the time δ has elapsed, ϕ_2 should have been true.

Since that sentence may look a bit complicated, and that the only parameters that cannot be deduced from the current event are ϕ_1, ϕ_2 and δ, we shall define a new operator, noted $\phi_1 \overset{\delta}{\hookrightarrow} \phi_2$, whose meaning is exactly the up-here formula.

We could use this operator to specify, for example, that when we've seen the first two steps of a TCP handshake, then the third one should occur within 2 units of time. This would be stated :

$$\texttt{synack} \wedge [\texttt{syn}, 3] \overset{2}{\hookrightarrow} \texttt{ack}$$

which means that, if we've seen a `synack` following a `syn` by less than 3 units of time, and that 2 units of time have elapsed since that time, then we also should have seen an `ack` following that `synack`.

```
tcpFlow(sa,sp,da,dp){
  ip.sadr(sa) & tcp.sport(sp) &
  ip.dadr(da) & tcp.dport(dp)
}
```

Fig. 2. Unification.

C. Introducing Recursion

It may occur that we want to express properties for which repetition is fundamental, for example when the time comes to express port scanning, brute force attacks, or denial of service attacks. In such occasions, the formulas used to express those repeating behaviors are of the form:

$$\phi := [\phi_1 \wedge [\phi_1 \wedge [\phi_1 \wedge [\ldots], \phi_2], \phi_2], \phi_2]$$

meaning that ϕ_1 occurs a certain amount of time, without ϕ_2 ever occurring. In the case of a password-cracking, brute force attack on a Telnet service, ϕ_1 could represent a connection request followed by a rejection, while ϕ_2 could express a timeout delay.

Then, we feel that this formula ϕ would somehow respect the property:

$$\phi := [\phi_1 \wedge \phi, \phi_2]$$

and we would like to be able to write down formulas that way.

To overcome this problem, we shall introduce the following notation, based on approximants [13]:

$$\begin{aligned} \nu^0 X.\phi &\stackrel{\text{def}}{=} \mathbf{tt} \\ \nu^n X.\phi &\stackrel{\text{def}}{=} \phi[\nu^{n-1} X.\phi / X] \end{aligned}$$

Then, the formula $\nu^3 X.\mathtt{syn} \wedge [X, 2]$ would represent 3 TCP connection requests separated by no more than two units of time. That is:

$$\begin{aligned} \nu^3 X.\mathtt{syn} \wedge [X, 2] &\stackrel{\text{def}}{=} \mathtt{syn} \wedge [\nu^2 X.\mathtt{syn} \wedge [X, 2], 2] \\ &\stackrel{\text{def}}{=} \mathtt{syn} \wedge [\mathtt{syn} \wedge [\nu^1 X.\mathtt{syn} \wedge [X, 2], 2], 2] \\ &\stackrel{\text{def}}{=} \mathtt{syn} \wedge [\mathtt{syn} \wedge [\mathtt{syn} \wedge [\mathbf{tt}, 2], 2], 2] \end{aligned}$$

V. Implementing Language

In this section, we give a brief description of the way we've implemented our logic in the case of network intrusion detection. The main advantage of the developed language is to allow to perform intrusion detection and passive information gathering in a declarative way.

A. Unification

In the preceding sections, we've used propositional constants like `syn`, `synack` and `syn` as primitives. However, to really perform intrusion detection, we might need to go deeper into details. In particular, we have to access different protocol fields, in order to compare their values to other ones or to given specific values. The way of doing this in our language is to use unification, as with Prolog. On figure 2, we see how we can define the predicate `tcpFlow` using unification. This predicate succeeds for a given packet if the different parameters, `sa`, `sp`, `da` and `dp`, can be unified to the fields `sadr`, `sport`, `dadr` and `dport` of the current packet. Obviously, if the current packet does'nt have a TCP header, the unification with the TCP ports can not succeed.

```
handShake(sa,sp,da,dp){
  snort(tcp (flags:A;)) & tcpFlow(sa,sp,da,dp) &
    [snort(tcp (flags:SA;)) & tcpFlow(da,dp,sa,sp) &
      [snort(tcp (flags:S;)) & tcpFlow(sa,sp,da,dp),
      3sec],
    2sec]
}

sessionClosing(sa,sp,da,dp){
  (snort(tcp (flags:F)) | snort(tcp (flags:R))) &
  (tcpFlow(sa,sp,da,dp) | tcpFlow(da,dp,sa,sp))
}

session(sa,sp,da,dp){
  [handShake(sa,sp,da,dp),
  sessionClosing(sa,sp,da,dp)]
}
```

Fig. 3. TCP Sessions Tracking.

B. Defining Scenarios

The main advantage of our logic is however to allow definition of packet sequences, or scenarios, using the operator $[\phi_1, \phi_2]$. Within our language, the definition of a simple predicate is not different than the one used to define a packet sequence. Thus, it is not surprising to find that the specification of a packet sequence is constituted of the same two main components: the header, specifying the name of the identified sequence and its parameters, and a detection part, specifying how to match each packet.

In figure 3, we can see how to specify a TCP handshake. A TCP handshake is first characterized by its source and destination addresses and ports. Then, each single step of the handshake is detected by a Snort rule, using the special predicate `snort`, which is defined such that it succeeds if and only if the snort detection engine identifies the current packet using the given rule. Finally, using the predicate `tcpFlow` we've just defined, we can verify that each step actually concerns the good TCP session. Once the handshake is defined, we can then state that a session is closing if we see a `fin` or a `rst` packet, and that a session is active between the handshake and the closing.

C. Using Recursion

The last thing we will discuss about the implementing language is the use of recursion, which is a must to be able to express repeating properties in a simple and succint manner. Recursion has been defined in section IV-C using the ν^n notation, which comes from the approximants theory. It is implemented here using the `max` keyword.

In figure 4, we see how we can define a portscan using recursion. It is quite interesting here to notice how we use

```
portScan(sa,da){
  max(X,i=0..5){
    snort(tcp (flags:S;)) &
    tcpFlow(sa,sp[i],da,dp[i]) &
    allDifferent(dp) &
    [X, 2sec]
  }
}
```

Fig. 4. TCP Portscan.

the iteration variable i and arrays to state that all destination ports on connection requests should be different. When the first `syn` is identified, then the variable `dp[0]` is unified with its destination port, and thus the variable `dp` is an array of size 1. When the second `syn` is identified, the variable `dp[1]` is unified with its destination port, and `dp` becomes an array of size 2. At each iteration, the predicate `allDifferent` is used to verify that all destination ports are different. There is no restriction over source ports. However, those still have to be stored in an array instead of a single variable otherwise it would mean that all source ports must be equals, the unification being done on the reception of the first connection request.

VI. Conclusion and future work

In this paper, we've presented a new approach to address stateful intrusion detection. This approach is based on a temporal logic which allows temporary properties to be specified. Temporary properties are properties that lay between given events. The detection of those events can in turn rely on some other temporary properties. Temporary properties can be seen as knowledge gathered from events. We've then shown how our logic could be used to express timing, safety and repetition properties which are often highly desirable in intrusion detection. Finally, we've shown how our logic can be implemented in order to work together with Snort.

A first prototype of such an implementation is currently under development.

Once this prototype is well tuned, the next problem we'll have to look at will be the modelling of the acquired information. That is, different acquired information could together allow us to deduce a new one. A simple example is the relation between MAC addresses and IP addresses. If we have information concerning a host, identified by its IP address, and we have the relationships between MAC addresses and IP addresses, then we should be able to access this information using the MAC address. To solve those problems in a general way, we have to have a well-structured data model. For now, we are thinking about using something similar to M2D2 [14].

Once information gathering and inferring are done, another step forward would be integration of high level security policies. High level security policies don't concern low level events, but do concern directly the gathered information. We are currently studying some other research that has been done already in that field [15].

References

[1] C. Green and M. Roesch, "Snort users manual 2.1.0 - the snort project," Dec. 2003.

[2] V. Paxson, "A system for detecting network intruders in real-time," in *In 7th USENIX Security Symposium*. Berkeley, CA: Usenix Association, January 1998.

[3] G. Vigna and R. A. Kemmerer, "Netstat: A network-based intrusion detection system," *Journal of Computer Security*, vol. 7, no. 1, 1999.

[4] F. Cuppens and R. Ortalo, "Lambda: A language to model a database for detection of attacks," in *Third International Workshop on Recent Advances in Intrusion Detection (RAID'2000)*, Toulouse, October 2000.

[5] B. Morin and H. Debar, "Correlation of intrusion symptoms: an application of chronicles," in *6th International Conference on Recent Advances in Intrusion Detection (RAID'2003)*, Pittsburgh, USA, September 2003.

[6] S. Eckmann, G. Vigna, and R. Kemmerer, "STATL: An Attack Language for State-based Intrusion Detection," *Journal of Computer Security*, vol. 10, no. 1/2, pp. 71–104, 2002.

[7] E. A. Emerson and J. Y. Halpern, "Sometimes and not never revisited: on branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[8] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1985.

[9] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[10] G. Rosu and K. Havelund, "Synthesizing dynamic programming algorithms from linear temporal logic formulae," 2000.

[11] Fyodor, "The art of port scanning," http://www.insecure.org/nmap/nmap_doc.html, 1997.

[12] A. P. Sistla, "Safety, liveness and fairness in temporal logic," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 495–512, 1994.

[13] D. Kozen, "Results on the propositional mu-calculus," *Theoretical Computer Science*, vol. 27, pp. 333–354, 1983.

[14] H. D. Benjamen Morin, Ludovik M and M. Ducass, "M2d2 : A formal data model for IDS alert correlation," in *5th International Conference on Recent Advances in Intrusion Detection*, Zurich, October 2002.

[15] F. Cuppens and A. Miège, "Modelling contexts in the or-bac model," in *19th Annual Computer Security Applications Conference*, Las Vegas, December 2003.