Abstract: Network Intrusion Detection System (NIDS) is an important and practical tool for network security. To guarantee a precise detection, the NIDS must detect packets at a wire speed. However, with the recent trend of high-speed networks, the capability of a single NIDS cannot meet the speed’s demand, resulting in rising of false negatives. To address this problem, Specification-based techniques have been proposed as a promising alternative that combine the strengths of misuse and anomaly detection. In this paper, we present an event calculus (EC) based framework towards the formal analysis of NIDS. This framework checks that security requirements and assumptions are preserved at run-time by monitoring the satisfaction of EC formulas that formalize them using the detection rules. This can be done by observing the network at run-time and checking observations against specified network behavior trying to detect deviations from what is specified.

Key words: NIDS, Formal analysis, Formal specification, Event Calculus

1 Introduction

With the growing number of attacks on network infrastructures, the need for techniques to detect and prevent attacks is increasing. Intrusion detection refers to a broad range of techniques that defend against malicious attacks. Intrusion detection techniques generally fall into one of the following categories: misuse detection, anomaly detection and specification-based detection.

Anomaly detection assumes that attacks will result in behavior different from that normally observed in a system, and can be detected by comparing the current behavior with the pre-established normal behavior. Anomaly detection has the advantage that no specific knowledge about security flaws is required in order to detect penetrations. However, it is difficult to set up the anomaly thresholds so that attacks produce significant anomalies. In addition, anomaly detection alone cannot detect all kinds of intrusions, since not all intrusions produce an identifiable anomaly.

Misuse detection attempts to identify known patterns of intrusions (intrusion signatures) when they occur. It can guarantee the detection of an intrusion if a signature of the intrusion is included in the system. However, it cannot detect previously unknown attacks, since it is not possible to specify intrusion signatures for exploiting vulnerability if the vulnerability is still unknown. Also, it is difficult to write signatures that capture all variants of an intrusion.

Specification-based detection relies on program specifications that describe the intended behavior of security-critical programs. The monitoring of executing programs involves detecting deviations of their behavior from these specifications, rather than detecting the occurrence of specific attack patterns. Thus, attacks can be detected even though they may not previously have been encountered.

Specification-based techniques have been proposed as a promising alternative that combine the strengths of misuse and anomaly detection [5]. In this approach, manually developed specifications are used to characterize legitimate networks/programs behaviors. As this method is based on legitimate behaviors, it does not generate false alarms when unusual (but legitimate)
program behaviors are encountered. Thus, its false positive rate can be comparable to that of misuse detection [5]. Since it detects attacks as deviations from legitimate behaviors, it has the potential to detect previously unknown attacks. However, in this context, some questions need to be answered:

- How much effort is required to develop network behavioral specifications? How do these efforts compare with that required for training anomaly detection systems?
- How effective is the approach in detecting novel attacks? Are there classes of attacks that cannot be detected by specification-based techniques that cannot be detected by anomaly detection or vice-versa?
- Can it achieve false alarm rates that are comparable to misuse detection?

In this paper, we present an approach to formal analysis of IDSs for high speed networks. This approach is applicable to IDSs that employ declarative rules for intrusion detection, including signature-based detection (matches the current network activities against a set of predefined attack signatures that represent known attacks and potential intrusive activities) and specification based detection (recognizes attacks as activities of objects that violate their specifications.).

The rest of this paper is organized as follows. We begin with an overview of our formal framework for intrusion detection in high speed networks in Section 2. Following this, we develop our methodology and encoding for specification development in Section 3. Section 4 discusses the most important related works. Finally, we summarize our conclusions in Section 5.

2 Overview of the framework

The approach is inspired by the significant body of formal methods research in designing and building trusted computer systems. Briefly, the process of designing and building a trusted system involves the development of a security model, which consists of a specification of a security policy (the security requirements or what is meant by security) and an abstract behavioral model of the system. Usually, the security policy can be stated as a mapping from system states to authorized (secure) and unauthorized (insecure) states [4] or as a property (often stated as an invariant) of the system (e.g., noninterference). The model is an abstraction of the actual system that provides a high level description of the major entities of the system and operations on those entities. There may be layers of abstractions within the model, each a refinement of the higher level abstraction. Given the security policy and model, one should be able to prove that the model satisfies the security policy, assuming some restrictions on the state of the system.

Then, our analysis framework is depicted in figure 1.

![Figure 1 Detection Framework](image)

It consists of the following elements:

- **Security Requirements** define properties that should be satisfied to guarantee the security of the network.
- **Assumptions** are additional properties injected to facilitate the verification process. Security properties that we are not sure of and more important, properties that cannot be efficiently monitored will be declared as assumptions (e.g., kernel of the system is not subject to attack). One kind of the verification consists of checking security properties of IDSs together with assumptions, with respect to security policies. Security policies are always satisfied with sufficiently strong assumptions. So the key is to identify assumptions that are strong enough but not too strong. An attack can violate a security policy by breaking its assumptions. So it is possible to verify the improvement of security by proving the weakening of assumptions. For example, assuming a policy $P$ is satisfied with assumption $A$ and with the deployment of the mechanism $m$, and $P$ is satisfied with assumption $B$ where $A$ implies $B$, then we can say $m$ improves the security because attacks violating assumption $B$ will also violate $A$, but attacks that violate assumption $A$ may not violate $B$.
- **Network specification** gives an abstract model for security critical entities of the network. One way to develop a specification for a program is to first identify what operations and accesses the program
needs to support its functionality. Based on an examination of the code or its behaviors, one writes rules in the specification to cover the valid operations of the program. The “draft” specification will be tested against the actual execution of the program. Often, the draft specification, when used to monitor the program execution, will produce false positives (i.e., valid operations performed by the program reported as erroneous because they are not included in the specification). Then, one augments the specification to include rules to cover these operations. In general, one needs to be very careful in writing the specification for a program to avoid errors. Writing specifications for a program is subtle and tricky, thus demanding an approach to rule validation.

- **IDS rules** vary dependent on the IDS. In our case, we will focus on Snort rules. Little research has been done on analyzing intrusion-detection rules. Different approaches have been taken to specify and analyze the intrusion signatures and detection rules [1] [6] [2] primarily for signature-based IDSs. A declarative language, MuSigs, is proposed in [2] to describe the known attacks. Temporal logic formulas with variables are used to express specifications of attack scenarios [6]. Pouzol and Ducasse formally specified attack signatures and proved the soundness and completeness of their detection rules.

- **Event logs** are necessary for the model because almost all IDSs are based on the analysis of the audit trails from operating systems, applications, and network components.

- **Verification engine**: the verification can be done either *a-priori*, i.e., at design time, or *a-posteriori*, i.e., after runtime to test and repair design errors, and formally verify whether the process design does have certain desired properties. The need for a-priori verification is important for the specification because it can be very complex, and therefore we need to check if the specified behavior is consistent, which is not a trivial task as soon as a complex security policies and properties can be considered. The a-posteriori verification is also important to provide knowledge about the context of and the reasons of discrepancies between abstract models and related instances. This kind of verification is necessary since some components that constitute the network may be dynamically specified at runtime, causing unpredictable interactions with other components, and making the a-priori verification method insufficient as it only takes into account static aspects.

### 3 Verification and Validation

In this section, we introduce our formal logic to be used, the ingredients of our encoding, the verification strategies, and analysis of attacks reports.

#### 3.1 Event Calculus

Event calculus (EC) is a formal language for representing and reasoning about dynamic systems. Because the language supports a representation of time that is independent of any events that might occur in the system, it is a particularly useful way to specify a variety of event-driven systems. Since its initial presentation [3], a number of variations of the Event Calculus have been presented in the literature [7]. In this work, we use a simple classical logic form of the EC, whose ontology consists of (i) a set of time-points isomorphic to the non-negative integers, (ii) a set of time-varying properties called fluents, and (iii) a set of event types (or actions). The logic is correspondingly sorted, and includes the predicates Happens, Initiates, Terminates and HoldsAt, as well as some auxiliary predicates defined in terms of these. Happens(a,t) indicates that event (or action) a actually occurs at time-point t. Initiates(a,f,t) (resp. Terminates(a,f,t)) means that if event a were to occur at t it would cause fluent f to be true (resp. false) immediately afterwards. HoldsAt(f,t) indicates that fluent f is true at t. The auxiliary predicate Clipped(t1,f,t2) expresses whether a fluent f was terminated during a time interval [t1,t2]. Similarly, the auxiliary predicate Declipped(t1,f,t2) expresses if a fluent f was initiated during a time interval [t1,t2].

The Event Calculus supports deductive, inductive and abductive reasoning. Deduction uses the description of the system behavior together with the history of events occurring in the system to derive the fluents that will hold at a particular point in time. Induction aims to derive the descriptions of the system behavior from a given event history and information about the fluents that hold at different points of time. Abduction can be used, given the descriptions of the behavior of the system, to determine the sequence of events that need to occur such that a given set of fluents will hold at a specified point in time.

Each event calculus theory is composed of axioms. A fluent that holds since the time of the initial state can be described by the following axioms [7]

\[
\text{holdsAt}(f, t) \leftarrow \text{initially}(f) \wedge \sim \text{clipped}(t0, f, t)
\]

\[
\text{holdsAt}(\neg f, t) \leftarrow \text{initially}(\neg f) \wedge \sim \text{declipped}(t0, f, t)
\]

Axioms below are used to deduce whether a fluent holds
or not at a specific time.

\[ holdsAt(f, t) \leftarrow \text{happens}(e, t_1, t_2) \land \text{initiates}(e, f, t_1) \land \neg \text{clipped}(t_1, f, t) \land t_2 < t \]

\[ holdsAt(\neg f, t) \leftarrow \text{happens}(e, t_1, t_2) \land \text{terminates}(e, f, t_1) \land \neg \text{declipped}(t_1, f, t) \land t_2 < t \]

More details and explanations concerning the EC logic are exposed in [7].

3.2 Encoding

Security requirements, assumptions, IDS rules, system specification are expressed using EC predicates and axioms. The event logs will be presented also as EC predicates.

IDS rules. As so far mentioned, we consider Snort rules. Snort rules are simple to write, yet powerful enough to detect a wide variety of hostile or merely suspicious network traffic. There are three base action directives that Snort can use when a packet matches a specified rule pattern: pass, log, or alert. Pass rules simply drop the packet. Log rules write the full packet to the logging routine that was user selected at runtime. Alert rules generate an event notification using the method specified by the user at the command line, and then log the full packet using the selected logging mechanism to enable later analysis. For instance, the following rule

\[ \text{alert tcp any any -> 192.168.1.0/24 any (flags: A; \neg ack: 0; msg: "TCP ping detected");} \]

shows that an alert message will be generated when you receive a TCP packet with the A flag set and the acknowledgement contains a value of 0. The destination of this packet must be a host in network 192.168.1.0/24. This rule can be expressed in EC as follows:

\[ \text{Happens(receive(TCP, A, ACK:0, 192.168.1.0/24), f, t) \Rightarrow holdsAt(ev(Alert, ; msg: "TCP ping detected");)} \]

Security requirements. They aim to protect the network from malicious attacks. Our approach checks that such requirements are preserved at run-time by monitoring the satisfaction of EC formulas that formalize them. This can be done by observing the network at run-time and checking observations. As an example of a monitoring property expressed in our EC-based language consider the formula below:

\[ \forall \_id1,\_id2,\_s,\_r:\text{String};\ :\text{v:ObjType} \ : t:\text{Time} \ \text{Happens}(e_{\_id1,\_s,\_r,\text{REQ-B}}, o(\_v),\_r,\_id1, t_1, t_1+t_1) \]

According to this formula, a sensor \_r which receives an event invoking the operation \( o(\_v) \) in it (i.e. \( e_{\_id1,\_s,\_r,\text{REQ-B}}, o(\_v),\_r,\_id1, t_1, t_1+t_1) \)) should complete the execution of \( o(\_v) \) and respond to the caller \( (\_s) \) within \( tu \) time units following the request.

Assumptions. specify rules on how to derive information about the state of the network that is being monitored based on observations of its behaviour. They are also expressed in EC.

Then, a collection of these EC formulas define the monitoring specification. A monitoring specification effectively describes the security policy of the network. When an attack takes place, we consider that the security policy is violated and, therefore, a monitoring rule is breached. This means that our requirements monitoring approach is capable of detecting immediate attacks to the network (intrusions. A threat, however, is a possible attack to the network; to detect them, we need an approach to be one-step-ahead and foresee future violations of monitoring rules.

Attackers are constantly looking for new ways of attacking networks. We somehow need to devise a strategy to prevent this. This strategy can be based on the security concept of an asset. Assets are the resources that a system must protect from incorrect or unauthorized use. They are attack targets; the motivation of an attack to the system.

3.3 SPIKE theorem prover

For the verification and the analysis of the security requirements we have employed the SPIKE induction prover, which seems particularly adapted to the task. The SPIKE induction prover has been designed to verify quantifier-free formulas in theories built with first order conditional rules. SPIKE was chosen for the following reasons: (i) its high automation degree, (ii) its ability on case analysis (to deal with multiple operations), (iii) its refutational completeness, (to find counter-examples), (iv) its incorporation of decision procedures. SPIKE proof method is based on the so-called cover set induction: Given a theory SPIKE computes in a first step induction variables where to apply induction and induction terms which basically represent all possible values that can be taken by the induction variables. Given a conjecture (rule or a policy) to be checked, the prover selects induction variables according to the previous computation step, and substitutes them in all possible way by induction terms. This operation generates several instances of the conjecture that are then simplified by rules, lemmas, and induction hypotheses.

The encoding of EC in SPIKE and details about the ingredients of this encoding cannot be presented here due to lack of space. However, interested readers can refer to [18] to get these details.
4 Related Works

In this section, we review some important papers that adopt a formal based intrusion detection system and are close to our approach. We plan also to study the reuse of these approaches in High speed networks. Ko and al. [9] proposed a specification-based approach for intrusion detection. The idea is to use traces, ordered sequences of execution events, to specify the intended behaviors of concurrent programs in distributed system. A specification describes valid operation sequences of the execution of one or more programs, collectively called a (monitored) subject. A sequence of operations performed by the subject that does not conform to the specification is considered a security violation. Each specification is called a trace policy. A grammar called parallel environment grammars (PE-grammars) was developed for specifying trace policies. The advantage of this approach is that in theory, it should be able to detect some new types of attacks that intruders will invent in the future. The drawback of this approach is that substantial work is required to specify accurately the behavior of the many privileged system programs, and these specifications will be operating-system specific.

To address this issue, [10] proposed the use of inductive logic programming to synthesize specifications from valid traces. The automatically generated specifications may be combined with manual rules to reduce the work involved in specification of valid program behaviors. The basic idea of [11] is to automatically generate the specification of a program by deriving an abstract model of the programs from the source or binary code. Wagner and Dean studied several alternative models, including the call-graph model and the abstract stack model. Central to these models is the control flow graph of a program; these models adopt different ways to represent the possible system call traces according to the control flow graph. Attractive features of this approach are that it has the potential to detect unknown patterns of attacks and it has no false alerts, although it may miss some attacks.

LAMBDa language [13, 17] was used to describe attack scenarios as a combination of actions. Each action has conditions or requirements that must be satisfied for the action to succeed, and successful actions affect the network and may satisfy conditions for other actions. Actions can be combined using operators that specify sequencing, parallel unconstrained execution, absence of a condition, nondeterministic choice between multiple equivalent actions, and synchronized execution. This language is labor intensive to use, only a few examples are provided, and an automated tool to create scenarios is not presented.

In [14], authors introduced a logic-based approach for modeling and analysis of networks, which supports automatic placement and configuration of sensors of a Network IDS. The approach is based on a very small set of logic predicates, which are suitable for modeling the topology of networks, and a set of derived predicates that express relevant properties of the modeled networks. In particular, the derived predicates introduced in this article formally express anomalous configurations of IP packets, which correspond to possible intrusion efforts by a malicious user.

[15] describes the formalization of a correctness proof for a conflict detection algorithm for firewalls in the Coq Proof Assistant. First, it gives formal definitions in Coq of a firewall access rule and of an access request to a firewall. Formally, two rules are in conflict if there exists a request on which one rule would allow access and the other would deny it. They express their algorithm in Coq, and prove that it finds all conflicts in a set of rules. They obtain an OCaml version of the algorithm by direct program extraction. The extracted program has successfully been applied to firewall specifications with over 200,000 rules.

An approach to firewall policy specification and analysis that uses a formal framework for argumentation based preference reasoning was proposed in [16]. By allowing administrators to define network abstractions (e.g. subnets, protocols etc) security requirements can be specified in a declarative manner using high-level terms. Also it is possible to specify preferences to express the importance of one requirement over another. The use of a formal framework means that the security requirements defined can be automatically analyzed for inconsistencies and firewall configurations can be automatically generated. The authors demonstrate that the technique allows any inconsistency property, including those identified in previous research, to be specified and automatically checked and the use of an argumentation reasoning framework provides administrators with information regarding the causes of the inconsistency.

5 Conclusions

The paper has presented a formal based approach to intrusion detection in the context of high speed networks. We have proposed the global architecture of our approach and detailed its components. Then, after presenting the ingredients of our verification strategy, we have showed how the proposed formalism can be used. The framework is still under development and refinement. We are working on using real scenarios and
a concrete application to show the feasibility of the proposed model. The verification process is also under refinement.

References


