STATL: An Attack Language for State-based Intrusion Detection

Steven T. Eckmann   Giovanni Vigna   Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106
`[eckmann,vigna,kemm]@cs.ucsb.edu`

Abstract

STATL is an extensible state/transition-based attack description language designed to support intrusion detection. The language allows one to describe computer penetrations as sequences of actions that an attacker performs to compromise a computer system. A STATL description of an attack scenario can be used by an intrusion detection system to analyze a stream of events and detect possible ongoing intrusions. Since intrusion detection is performed in different domains (i.e., the network or the hosts) and in different operating environments (e.g., Linux, Solaris, or Windows NT) it is important to have an extensible language that can be easily tailored to different target environments. STATL defines domain-independent features of attack scenarios and provides constructs for extending the language to describe attacks in particular domains and environments. The STATL language has been successfully used in describing both network-based and host-based attacks, and it has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. An implementation of the runtime support for the STATL language has been developed and a toolset of intrusion detection systems based on STATL has been implemented. The toolset was used in a recent intrusion detection evaluation effort, delivering very favorable results. This paper presents the details of the STATL syntax and semantics. Real examples from both the host and network-based extensions of the language are also presented.

1 Introduction

Intrusion detection systems analyze information about the activities performed in a computer system or network, looking for evidence of malicious behavior. The information may come in the form of audit records produced by the operating system auditing facilities, log messages produced by different types of sensors and network devices, or in the form of raw network traffic obtained by eavesdropping a network segment. These data sources are used by intrusion detection systems (IDSs) in two different ways, according to two different paradigms: *anomaly detection* and *misuse detection*. In anomaly detection systems [16, 17, 9], historical data about a system's activity and specifications of the intended behavior of users and applications are used to build a profile of the "normal" operation of the system. Then, the intrusion detection system tries to identify patterns of activity that deviate from the defined profile. Misuse detection systems take a complementary approach [15, 2, 23]. Misuse detection tools are equipped with a number of attack descriptions (or "signatures") that are matched against the stream of audit data looking for evidence that the modeled attack is occurring. Misuse and anomaly detection both have advantages and disadvantages. Misuse detection systems can perform focused analysis of the audit data and they usually produce only a few false positives, but they can detect only those attacks that have been modeled. Anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for in terms of the large number of false positives and the difficulty of training a system with respect to a very dynamic environment. A recent DARPA-sponsored evaluation of intrusion detection systems [18] provided interesting feedback on the characteristics of several anomaly and misuse detection systems confirming the advantages and disadvantages inherent in the two approaches.

A critical feature of misuse detection systems is the description of attack signatures. Several *attack languages* have been proposed by the research community. However, most of the existing languages have been developed to support intrusion detection in particular domains and environments. These languages suffer from the idiosyncrasies of

their operational settings and are difficult to extend to new environments. In addition, the semantic model and the abstractions provided to the signature developer are either too low-level or not clearly defined.

Our experience with intrusion detection in different domains suggested the design of a domain-independent attack description language that could be extended in a well-defined way to match different operating environments. The result of this research is STATL, which is a language to support misuse detection. STATL defines the domain-independent features of attack scenarios and also provides constructs for extending the language to describe attacks in particular domains and environments. The STATL language has been successfully used in describing both network-based and host-based attacks, and it has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. An implementation of the runtime support for the STATL language has also been developed and a toolset of intrusion detection systems based on STATL has been implemented [31]. Even though the language was originally developed to support the development of intrusion detection systems based on the *State Transition Analysis Technique* (STAT) [14], STATL is general and flexible enough to be used as a common language for different misuse detection systems.

This paper presents STATL's syntax and semantics, and describes examples of its use in both network-based and host-based intrusion detection. Section 2 describes the STAT technique and our experience with intrusion detection. Section 3 presents some desirable features for attack description languages and also describes related work. Section 4 gives an overview of the language. Section 5 describes the details of STATL's syntax and informal semantics and gives examples of their use. Section 6 discusses our experience with the language, and the final section draws some conclusions and outlines future work.

2   Experience with intrusion detection

The *State Transition Analysis Technique* (STAT) [14] was conceived in 1992 as a misuse detection method to describe computer penetrations as sequences of actions that an attacker performs to compromise the security of a computer system. A disadvantage of most signature-based approaches has been the inability to detect new attacks. The STAT approach was conceived to mitigate this disadvantage by abstracting from the details of the modeled attacks. Actions are abstracted from their native form (e.g., standard audit records or network packets) to a higher-level repre-

sentation so that similar actions in a system that may have different low-level representations are mapped to a single action type. In addition, the STAT methodology supports a modeling approach that represents only those steps in an intrusion that are necessary for the effectiveness of the attack. By abstracting away from the details of a particular attack, it is possible to detect previously unknown variations of an attack or attacks that exploit similar mechanisms.

The state transition analysis technique has been used for both host-based and network-based intrusion detection. The first implementation of the host-based tool, which is called USTAT, clearly demonstrated the value of the STAT approach for intrusion detection in the UNIX operating system [12, 13, 24]. However, because the original USTAT prototype was developed in an ad hoc way, a number of characteristics of this first prototype were difficult to modify or to extend to match new environments (e.g., Windows NT). The state transition analysis technique was also applied to networks. The resulting tool, called NetSTAT, is aimed at real-time network-based intrusion detection [32]. The first NetSTAT prototype was also developed ad hoc, building a completely new tool that would fit the new domain.

In the original STAT model, attacks were represented using an informal *state transition diagram* language, with states representing snapshots of a system's security-relevant properties and resources, and transitions representing the critical signature actions of the attack. The initial USTAT implementation used a simple forward-chaining rule language, which was hard to extend, even to handle new Solaris attacks. Extending USTAT's rule language to the network domain, or even to other host operating systems, was infeasible, so the first NetSTAT implementation represented attack signatures in C code, which was compiled and linked into the IDS. In both cases, state transition diagrams were used to develop and reason about attacks, but these diagrams were translated by hand to the form required by the target IDS.

The effectiveness of these original STAT-based tools was successfully demonstrated in the 1998 DARPA Intrusion Detection Systems Evaluation [18, 6, 7]. Participation in this evaluation revealed that although USTAT and NetSTAT were both developed in an ad hoc way and virtually independently of each other, there were many similarities in the mechanisms they used to match attack scenarios against the input event streams. As a consequence, it was determined that the STAT-based tool suite could be redesigned as a *family* of systems. The toolset would share a common domain-independent language, called STATL, to represent attack scenarios, and a "core" module to support the domain-independent mechanisms used at runtime to match

attack scenarios against a stream of events. The resulting STATL/core design captures the domain-independent characteristics of the STAT approach.

Because intrusion detection is performed in particular domains (e.g., hosts or networks) and in specific environments (e.g., Windows NT or Solaris), the new core-based design also provides a well-defined way to extend STATL and the core module to obtain a complete intrusion detection system, tailored to the characteristics of the specific domain and environment.

Figure 1 gives a high-level description of the roles of the core module and the STATL language in the creation of an intrusion detection system using the core-based approach. Both the STAT core module and the STATL core language are domain-independent. The core language is extended with domain/environment-specific types and functions to obtain the language used to represent intrusions for a specific STAT-based tool. The language extension is compiled in the corresponding *extension module*, and this module is used, together with the STAT core module, to construct the actual intrusion detection system. Once the STAT-based tool has been developed and the extended language defined, the intrusion detection system user (e.g., the Information Systems Security Officer) writes attack scenarios using the extended language. These attack scenarios are then compiled into dynamically linked modules, called *scenario plugins*. The plugin modules are dynamically loaded into the intrusion detection application at runtime, and the intrusion detection system equipped with the scenario plugins can then process the system's event stream, looking for attack signatures. The details of the compilation/translation process are given elsewhere [33, 31].

The STATL language has been successfully used in describing both network-based and host-based attacks. In addition the language has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. Section 6 describes the core-based STAT toolset in more detail. Developing a family of intrusion detection systems demonstrated that the core-based approach supports reuse, portability, extensibility, and customization. In addition, separating the critical domain-independent runtime mechanisms from the specific features of the different operating environments allowed for the optimization of critical functionalities. As a result, the performance of the tools increased by an order of magnitude.

3   Attack languages

Computer and network attacks are becoming more common and sophisticated and there is a need for representing,

classifying, and sharing information about penetrations, exploits, and intrusion techniques. As a result, attack languages are needed to encode the "manifestations" of an attack into a suitable format, to recognize an attack given a manifestation, and to react to or report an attack. Attack languages are also useful for analyzing the relationships among different attacks in order to identify coordinated attacks against a system. In addition, attack descriptions can be used to describe attack histories/scenarios and they can be used for reproducing attacks, for testing purposes.

Attack languages have recently received a lot of attention from the intrusion detection community. Unfortunately, no common language for describing attacks exists. In fact, many attacks are referred to by just one or two words, such as "buffer overflow," "SYN-flooding," or "land." Yet, it is possible to identify at least six classes of "attack languages": *event languages*, *response languages*, *reporting languages*, *correlation languages*, *exploit languages*, and *detection languages*. These language classes define different scopes and goals.

Event languages are used to describe "events." These events are the basic input for security analysis. This class of languages is mainly focused on the specification of data format. In most cases, there is not a formal language definition; rather, there is some natural language description of the format of each event type and a schematic description of the associated structure. There are many examples of event languages with different formats and/or features: BSM audit record specifications [29], tcpdump packets [20], syslog messages [30], etc. A well-defined common event language would be beneficial to the ID community. It would support component and preprocessor reuse, simplify data sharing among different systems, and allow for the merging of different event streams. An example of a proposed standard format is given in [1]. Other general content specification languages, such as XML [34], could also be used.

Response languages are used to specify the actions to be taken in reaction to the detection of attack manifestations. Most IDSs currently do not have a well-defined response language. Instead they use library functions written in general-purpose languages (e.g., C or Java) and have limited customizability and extendibility. A common response language would be useful so that response procedures could be defined once and installed in different intrusion detection systems. In addition, a common language would support dynamic reconfiguration of response. For example, different types of responses could be activated on the basis of the evolution of an attack.

One of the possible responses to an attack is the reporting to a human Security Officer or to an application. Reporting languages are used to describe alerts containing in-
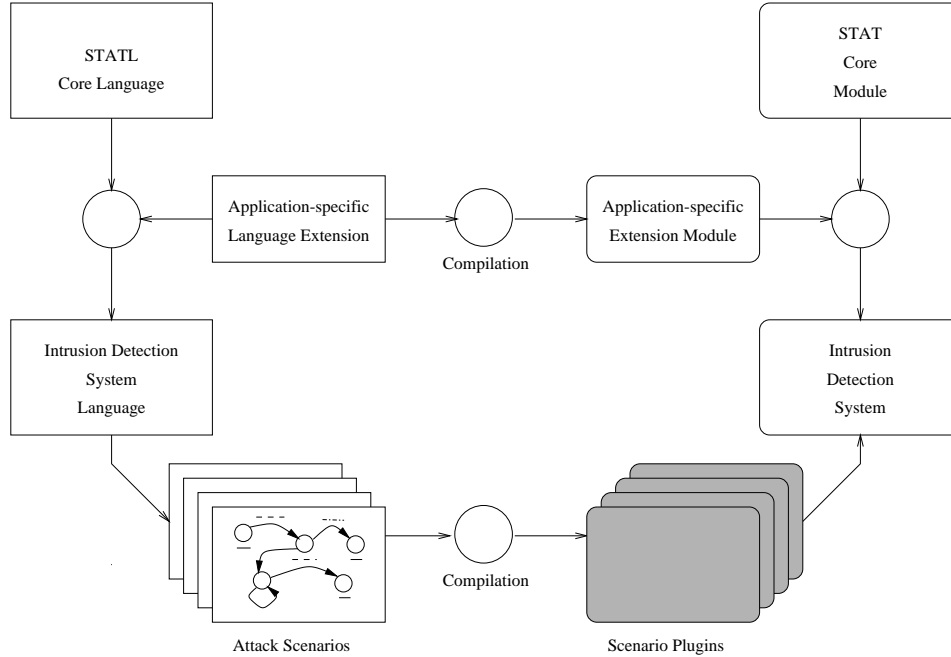
Figure 1: The STAT core module, STATL, and their extensions.

formation about an attack, such as source of the attack, target of the attack, type of the attack (if known), events that are part of the manifestation, etc. Note that a reporting language could also be used as an event language at a higher level, for example as input to correlators. A common reporting language has been identified as an asset by the ID community and standard reporting language formats have been proposed. Two examples of reporting languages are the Common Intrusion Specification Language (CISL) [10] and the Intrusion Detection Message Exchange Format (ID-MEF) [3]. CISL is part of the Common Intrusion Detection Framework (CIDF) [11]. The language is based on the concept Generalized Intrusion Detection Objects, called GIDOs, which, in turn, are specified as S-expressions. ID-MEF is a product of the Intrusion Detection Working Group (IDWG). IDMEF is based on XML and it builds on much of the previous CIDF work. The language is currently undergoing the Internet Engineering Task Force (IETF) standardization process.

Correlation languages are currently the focus of much research in the intrusion detection and response community. These languages rely on the semantically rich alerts provided by different intrusion detection systems and attempt to recognize "the big picture." That is, they specify relationships among attacks to identify coordinated attempts to break the security of an information system and identify the impact on the target system. The application of a correlation

language is sometimes the application of existing detection techniques at a higher level of abstraction. For instance, instead of analyzing BSM events and tcpdump events, correlators may use similar techniques to analyze alerts. Examples, of current approaches to the correlation problem are the use of Bayesian networks, as demonstrated in Honeywell's ARGUS system and SRI's EMERALD, event-based reasoning, as demonstrated in UCSB's STATL [8], and rule-based reasoning, as in SRI's P-Best [19].

Exploit languages are used to describe the steps to be followed to perform an intrusion. They are usually executable general-purpose languages such as C, C++, Perl, Tcl, Bourne Shell, Python, etc. There are also languages that are explicitly designed to support the scripting of attacks. Examples are CASL (Custom Attack Simulation Language) [22] and NASL (Nessus Attack Specification Language) [5]. Both these languages provide language-level support for attack scripting. A common exploit language would be useful to the security community. For example it would simplify the generation of test cases for intrusion detection systems. In addition, it would support the sharing and understanding of attacks among the members of the security community.

Detection languages are designed to support intrusion detection, and they are usually referred to as "attack languages." These languages provide mechanisms and abstractions for identifying the manifestation of an attack. The de-

4

scription of an attack is in a form that can be processed by the language runtime and can be matched against a stream of input events, looking for evidence that an instance of an attack is occurring.

Examples of detection languages are N-code [26], used by the Network Flight Recorder (NFR), P-BEST [19], which is the rule-based component of SRI's Emerald, UCSB's STATL [8], which is used by the STAT Toolset, the languages used by the Bro [23] and Snort [27] systems, and specification languages such as [17] and [28]. NFR, Bro, and Snort are network-based IDSs. Their domain-specific languages include features that support writing concise signatures for network attacks, such as IP addresses and ordered or hashed lists with an associated membership operator. Emerald includes both host-based and network-based sensors; therefore, it needs a more general language than those used in NFR, Bro, or Snort. Thus, Emerald uses a production-based expert system that includes a general-purpose rule language.

Another example detection language is RUSSEL, which was developed as part of the ASAX project [21]. A key goal of the ASAX project was the development of a rule-based language for audit trail analysis. An attack signature is represented as a RUSSEL program, which is composed of one or more modules, and each module contains one or more rules. A RUSSEL rule specifies a condition or pattern to apply to a single audit record, and an action to take when a condition matches. RUSSEL includes special constructs to support audit trail analysis (e.g., `trigger` to activate rules in future states). RUSSEL includes a hook to C for extensibility, such as for implementing new predicates on system state. A RUSSEL program is compiled into a form that the ASAX analyzer interprets.

Although RUSSEL has many of the attributes that are desirable for a detection language, it is too low-level for general use, and its extensibility and portability are limited. For example, timeouts are needed to express signatures for many attacks (see the half-open TCP connection signature in Section 6.2). In RUSSEL timeouts are typically expressed by setting a variable to the expiration time, activating a rule to compare the expiration time to the current time, and applying that rule to each audit record. STATL has timers built in as an abstract concept (see Section 5.10). Another example is that in RUSSEL all scenario evolution must be explicitly specified by the signature writer using rule activation and deactivation. In contrast, STATL provides three kinds of transitions with different high-level evolution semantics (see Section 4). As a last example, many signatures, especially in the network domain, are more conveniently expressed using nested events or conjunctions and disjunctions of events. For example, in STATL one could write

```
[IP ip [TCP tcp]] : f(ip.header, tcp.header)
```

to specify an IP packet containing a TCP segment and satisfying a particular predicate applied to the IP and TCP headers. Specifying this in RUSSEL requires one rule to match the IP part of the pattern and another to match the TCP part of the pattern. The first rule would have to save IP header information and activate the second rule for application to the current record. Mounji, who defined RUSSEL, says in the conclusions of his thesis that "it would be useful to design a higher-level language, which allows the user to express more declarative queries on the audit trail." ([21], p.126).

Although the languages mentioned above are useful, none of them has all the attributes that would make it usable as a common *detection language*. A detection language ideally provides a means of discourse about attack signatures in an abstract, IDS-independent form, but it also includes mechanized support for incorporating attack descriptions into IDSs. These detection language goals depend on the following qualities:

Simplicity: the language should provide just those features needed to represent attack scenarios.

Expressiveness: the language should be able to represent any attack signature that is detectable.

Rigor: the language should have a rigorously defined, implementation-independent syntax and semantics, so that the meaning of any attack description is unambiguous.

Extensibility: new domains have new event types, and require predicates and functions on those types. It should be possible to extend the language to new domains in a well-defined, relatively simple way.

Executability/Translatability: it should be possible to incorporate attack descriptions in an IDS application, without any manual translation. Either the specifications must be executable, or techniques and tools must be provided to translate from an attack specification to an efficient implementation.

Portability: the language processing tools should be easily adaptable to different environments.

Heterogeneity: it should be possible to describe attacks using events from multiple domains, such as IP packets and host audit records.

STATL was developed as a state/transition-based attack detection language that has all the qualities listed above. The next section gives an overview of STATL.

4 STATL Overview

A STATL specification is the description of a complete attack scenario. The attack is modeled as a sequence of steps that bring a system from an initial safe state to a final compromised state. This modeling approach is naturally supported by a state/transition-based language. One of the advantages of this approach is that state/transition specifications can be represented graphically by means of state transition diagrams (STD). Therefore, even though STATL is primarily a text-based language, the STATL development environment includes a graphic editor that allows one to directly visualize the STD representing an attack scenario.

The STATL language provides constructs to represent an attack as a composition of *states* and *transitions*. States are used to characterize different snapshots of a system during the evolution of an attack. Obviously, it is not feasible to represent the complete state of a system (e.g., volatile memory, file system, etc.); therefore, a STATL scenario uses variables to record just those parts of the system state needed to define an attack signature (e.g., the value of a counter or the ownership of a file). A transition has an associated *action* that is a specification of the event that may cause the scenario to move to a new state. For example, an action may be the opening of a TCP connection or the execution of an application. The space of possible relevant actions is constrained by a *transition assertion*, which is a filter condition on events that may possibly match the action. For, example an assertion may require that a TCP connection is opened with a specific destination port or that an application being executed should be part of a predefined set of security-critical applications.

It is possible that several occurrences of the same attack are active at the same time. A STATL attack scenario therefore has an operational semantics in terms of a set of *instances* of the same scenario *prototype*. The scenario prototype represents the scenario's definition and global environment, and the scenario instances represent attacks currently in progress.

The evolution of the set of instances of a scenario is determined by the type of transitions in the scenario definition. A transition can be *consuming*, *nonconsuming*, or *unwinding*. A nonconsuming transition is used to represent a step of an occurring attack that does not prevent further occurrences of attacks from spawning from the transition's source state. Therefore, when a nonconsuming transition fires the source state remains valid, and the destination state becomes valid too. For example, if an attack has two steps that are the creation of a link named "-i" to a SUID shell script and the execution of the script through the created link, then the second step does not invalidate the previous state. That is, another execution of the script through the same link may occur. Semantically, the firing of a nonconsuming transition causes the creation of a *new* instance. The original instance is still in the original state, while the new instance is in the state that is the destination state of the fired transition. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. Semantically, the firing of a consuming transition does not generate a new scenario instance; it simply changes the state of the original one. Unwinding transitions represent a form of "rollback" and they are used to describe events and conditions that may invalidate the progress of one or more scenario instances and require the return to an earlier state. For example, the deletion of a file may invalidate a condition needed for an attack to complete, and, therefore, the corresponding scenario instances may be brought back to a previous state, such as before the file was created. More precisely, the firing of an unwinding transition from state S_y back to a previous state S_x for a certain scenario instance causes the deletion of all the scenario instances that were created by the series of events that brought the unwinding instance from state S_x to state S_y. Due to space limitations, the details of scenario execution are not presented in this paper. The interested reader can find more information in [31].

5 STATL syntax, informal semantics, and examples

As discussed in Section 3, an attack detection language must have a rigorously defined syntax and semantics. This section presents the details of STATL's syntax and semantics. It also includes fragmentary examples for each of the syntax rules. In the syntax rules, literal keywords are in **boldface** and other literal text is enclosed in single quotes. Optional items are enclosed in square brackets '[', ']', items that may appear zero or more times are enclosed in curly braces '{', '}'. Alternatives are separated by '|' and grouped with parentheses where necessary to indicate associativity. Examples may include ellipses (. . .) to indicate that details have been left out; the ellipses are not part of STATL.

5.1 Lexical elements

STATL identifiers consist of letters, digits, and the underscore character '_', and start with a letter. For example `host_name` and `IPaddr2` are identifiers. STATL identifiers are case-sensitive, so `IPaddress` is different from `IPAddress`. STATL compound identifiers use stan-

dard object-oriented dot notation, as in "object.attribute". STATL keywords are reserved words and may not be used as identifiers. For example, since `scenario` is a keyword, it may not be used as a variable name.

STATL includes two kinds of comments: any text between "/*" and "*/" (except "*/"), including the delimiters, is a comment. Any text following "//" to the end of the line, including the "//" marker, is a comment. Whitespace may appear anywhere in a STATL specification except within tokens (keywords, identifiers, and multiple-character operators).

5.2 Data types

STATL includes several builtin types: `int` and `u_int` in various sizes, `bool`, `string`, `timeval` (for timestamps), and `timer` (described in Section 5.10). It also includes arrays, plus containers `vector`, `set`, `list`, and `map`. It is not possible to define new data types within a STATL scenario. Application-specific types must be defined within the application-specific extension library. For example, Net-STAT scenarios may use different types than USTAT scenarios, but both use `int` and `timeval`.

5.3 Scenario

A scenario uses zero or more libraries of application-specific types, events, functions, and predicates. A scenario has a name, may have parameters, may contain annotations and constant and variable declarations, and most importantly, contains the states and transitions that define the "attack signature" – what to match and what to do with matches. A scenario may also define supporting functions to be used in state and transition assertions and code blocks:

Scenario ::=
    { **use** *LibraryID* {',' *LibraryID*} ';' }
    scenario *ScenarioID*
    [*ScenarioParameters*]
    '{'
        [*FrontMatter*]
        {*State* | *Transition* | *NamedAction*}
    '}'
    { *FunctionDefinition* }

A scenario must have at least one transition and two states – the initial state and a final state. The initial state must have no incoming transitions, and final states have no outgoing transitions. Scenario parameters are specified as a list of comma-separated typed identifiers:

ScenarioParameters ::=

    '(' *Parameter* {',' *Parameter*} ')'
Parameter ::= *Type ParameterId*

Example:

```
scenario example (string host, int count)
{ ... }
```

The example scenario has two parameters, `hostname` and `count`. Parameters are accessible by the scenario instances as global constants.

5.4 Front Matter

Scenarios may include annotations, and may declare constants and variables:

Front Matter ::=
    {(*Annotation* | *ConstDecl* | *VarDecl*)}

ConstDecl ::=
    const *Type ConstId* '=' *InitialValue* ';'
VarDecl ::=
    [**global**] *Type VarId* ['=' *InitialValue*] ';'

Annotations are described in Section 5.12. A variable declared "global" is shared by all instances of the scenario. A variable not declared "global" is instantiated privately in each instance of the scenario. Variables may be assigned initial values.

Example:

```
use netstat;
scenario example
{
  const int bufsize = 1024;
  global int count = 0;
  Host server;
  ...
}
```

This example declares a constant integer `bufsize` with value 1024 and declares a global variable `count` with initial value 0. This variable will be shared by all instances of the scenario. That is, if a scenario instance increments the `count` variable, the update is seen by all of the other instances of the scenario. The variable declaration in the example also includes a variable named `server` of type `Host` (a NetSTAT type). Because `server` is a local variable, each instance of the scenario will have its own copy of `server`.

7

5.5 State

"State" is one of the two fundamental concepts in STATL. States have names so they can be referred to in transitions and in the graphical representation of the scenario (i.e., in the STD). Each state may have annotations, an assertion, and a code block, but these elements are optional:

State ::=
    [**initial**]
    state *StateId* {*Annotation*}
    '{'
        [*StateAssertion*]
        [*CodeBlock*]
    '}'

Exactly one state must be designated as the initial state. When a scenario plugin is loaded into an IDS a first instance (the "root" instance) is created in the initial state.

The state assertion, if present, is tested before entry to the state, after testing the assertion of the transition that leads to the state. A state's assertion is implicitly True if none is specified. A state's code block is executed after the incoming transition's assertion and the state's assertion have been evaluated and found to be True, and after the incoming transition's code block (if it exists) is executed.

Example:

```
scenario example
{
  const int threshold = 64;
  int counter;
  ...
  initial
  state s1 { }
  ...
  state s3
  {
    counter > threshold
    { log("counter over \
               threshold limit"); }
  }
  ...
}
```

In this example state `s1` is designated as the initial state. It has neither an assertion nor a code block. State `s3` has an assertion and a code block. The assertion specifies that the value of local variable `counter` is greater than the value of constant `threshold`. The code block calls the built-in procedure `log` to write a message to the IDS's log file.

5.6 Transition

"Transition" is the second of the two fundamental concepts in STATL. Each transition has a name and must indicate the pair of states that it connects. Transitions may have the same source and destination state; that is, loops are allowed. In addition, a transition may have annotations, must specify a type, must specify an event type to match, and may have a code block:

Transition ::=
    transition *TransitionID* '(' *StateId* '->' *StateId* ')'
    (**consuming** | **nonconsuming** | **unwinding**)
    {*Annotation*}
    '{'
        ('[' *EventSpec* ']' | *ActionId*)
        {*Annotation*}
        [':' *Assertion*]
        [*CodeBlock*]
    '}'

A transition's event is specified either directly (see Section 5.7) or by reference to a named signature action (see Section 5.8). In the former case the transition's assertion is just the assertion in the transition. In the latter case, if the named signature action includes an assertion and the transition also includes an assertion, then the resulting assertion is the conjunction of the two assertions. An example is given in Section 5.8, after named signature actions are defined.

A transition's code block is executed after evaluating the transition's assertion and the destination state's assertion, and before executing the destination state's code block. More precisely, the order of evaluation of assertions and the execution of code blocks, after matching an event type (defined in Section 5.7), is as follows:

1. evaluate the transition assertion. If True, then

2. evaluate state assertion. If True, then

3. execute transition code block, possibly modifying local and global environments;

4. execute state code block, possibly modifying local and global environments.

A transition's code block may perform any computation supported by STATL and the IDS extension in use, but is typically used to copy event field values into the global or local environment for later reference.

Example:

```
use ustat;
scenario example
{
  int userid;
  ...
  transition t2
```

```
   (s1 -> s2) nonconsuming
  {
    [READ r] : r.euid != r.ruid
    { userid = r.euid; }
  }
  ...
}
```

In this example, `t2` is a nonconsuming transition that leads from state `s1` to state `s2`. The event spec indicates that the transition should match events of type `READ`, with a filter condition specifying that the `euid` and `ruid` fields of the event must differ for the transition to fire. The transition's code block copies the `euid` field of event `r` into the local variable `userid` for later reference.

5.7 EventSpec

"Event specs" are the essential elements of transitions. They specify what events (signature actions) to match and under what conditions.

EventSpec ::=
        (*BasicEventSpec* [*SubEventSpec*]) | *TimerEvent*

BasicEventSpec ::= *EventType EventId*

SubEventSpec ::= '[' *EventSpec* { ',' *EventSpec* } ']'
EventType ::= **ANY** | *ApplEventType*
        '(' *ApplEventType* {'|' *ApplEventType* } ')'

An event spec is either a basic event spec optionally followed by a subevent spec, or it is a timer event (see Section 5.10. A *basic event spec* identifies the built-in meta-event "type" ANY, which matches any event, or an application-specific event type (e.g., READ) or a disjunction of application-specific event types (e.g., (UDP | TCP)), and a name that will be used to reference the matching event. A basic event spec identifying a single type matches an event of the same type only. A basic event spec that is the disjunction of two or more event types matches an event of any of the types in the disjunction. A subevent spec identifies a set of event specs. Subevent specs enable complex, tree-structured event patterns. A subevent spec matches a set of subevents if each event spec in the subevent spec matches an event in the set of subevents.
Example:

```
[(READ | WRITE) access] :
        access.euid != access.ruid
```

Example:

```
[IP d1 [TCP t1]] :
     d1.src == 192.168.0.1 &&
     t1.dst == 23
```

The first example is a USTAT event spec that matches read or write events in which the effective and real user-ids differ. The second example is a NetSTAT event spec (with a subevent spec) that matches any IP datagram containing a TCP segment, with source IP address 192.168.0.1 and destination port 23.

The built-in meta-event type ANY is effectively the same as disjunction over all application-specific event types, but is easier to specify (and more efficient to implement as a special case).

5.8 NamedSigAction

A named signature action has a name and specifies an event spec, and may have annotations:

NamedSigAction ::=
    action *ActionId* {*Annotation*}
    '{'
        ('[' *EventSpec* ']' | *ActionId*)
        {*Annotation*}
        [':' *Assertion*]
    '}'

Named signature actions may be used to improve clarity and maintainability when multiple transitions have identical or similar actions, for example, having the same action type but slightly different assertions. In such cases the common part can be factored out, put into a named signature action, and then used in the similar transitions.

Example:

```
use ustat;
scenario example
{
  ...
  action a1
  {
    [WRITE r] : r.euid != 0
  }
  transition t1 (s1 -> s2)
  {
    a1: r.euid != r.ruid
  }
  transition t2 (s1 -> s3)
  {
    a1: r.euid == r.ruid
  }
  ...
}
```

9

In this example transitions `t1` and `t2` both use named signature action `a1` as their event spec, but with different assertions. This is equivalent to:

```
scenario example
{
  ...
  transition t1 (s1 -> s2)
  {
    [WRITE r] : (r.euid != 0) &&
                (r.euid != r.ruid)
  }

  transition t2 (s1 -> s3)
  {
    [WRITE r] : (r.euid != 0) &&
                (r.euid == r.ruid)
  }
  ...
}
```

5.9 CodeBlock

Transitions and states may have code blocks that are executed after the corresponding transition and state assertions have been evaluated and found to be True. A code block is a sequence of statements enclosed in braces:

CodeBlock ::=
    '{'
        {*statement*}
    '}'

The statements in a codeblock can be assignments, *for* and *while* loops, *if-then-else*, procedure calls, etc. Semantically, the statements in a STATL code block are executed in order, in the context of the global and local environments of the scenario instance in which the code block is executed.

5.10 Timers

Timers are useful to express attacks in which some event or set of events must (or must not) happen within an interval following some other event or set of events. Timers can also be used to prevent "zombie" scenarios – scenarios that have no possible evolution – from wasting memory resources.

Timers are declared as variables using the built-in type `timer`. There are both local and global timers. All timers must be explicitly declared. Timers are started in code blocks using the built-in procedure `timer_start`:

Example:

```
scenario example
{
  timer t1;
```

```
  state s1
  {
    { timer_start(t1, 30); }
  }

  transition expire (s1->s2)
  { timer t1 }
  ...
}
```

The code block of state `s1` starts timer `t1`, which will expire in 30 seconds (i.e., at a time 30 seconds later than the timestamp on the event that led to state `s1`). Timer expiration is treated as an event, and these events may be matched by using "timer events" as transition event specs. The timer event `timer t1` matches the expiration of the timer named `t1`. When timer `t1` expires, transition `expire` will fire, leading to state `s2`.

Starting a timer that is already "running" resets that timer. A single timer may appear in multiple transitions; every enabled transition that has `timer t` as its event spec fires when the timer expires.

5.11 Assertions

Assertions appear as filter conditions in states and in event specs (which are the matching element of transitions). STATL assertions are built up from literal constants, variable and constant names, function calls, and common arithmetic and relational operators. A STATL assertion is evaluated at runtime in the context of the global and local environments of the scenario instance where it is evaluated.

Assertions may use, but may not change, the value of any name in the global or local environment. In addition, transition assertions may refer to the events named in the event spec and to the fields of those events.

5.12 Annotations

STATL annotations provide a standard way to extend the language with application-specific features. The annotations to a scenario are processed by a specialized component of the offline STAT architecture, called the *Analyzer* [33]. The Analyzer's task is to perform any IDS-specific preprocessing on a STATL scenario before the scenario is actually translated into an executable representation. For example, the NetSTAT Analyzer uses the annotations of a STATL scenario to determine where a probe should be placed in a target network and how the probe should be configured in order to be sure to detect the described attack. The Analyzer relies on another component, the *Factbase*, to obtain information about the target environment. For example, in NetSTAT the Factbase contains information about the network topology and the services deployed.

10

Syntactically, an annotation is an annotation tag optionally followed by a list of expressions, all enclosed in the annotation delimiters "<*" and "*>":

Annotation ::=
    <* *Id* {*Expression* { ',' *Expression*} } ***>**

Annotations may be attached to states, transitions, event specs, named signature actions, and scenarios. Each IDS defines any scenario customization it needs as annotations, and the associated Analyzer is expected to process them (any annotations left by the Analyzer will be ignored by the STATL translator).

An example of an application-specific language extension is the ENDPOINT_PORTS annotation often used in NetSTAT actions to specify which two network interfaces are communicating. For example, the event spec:

```
[IP d [TCP t]]
  <* ENDPOINT_PORTS a, b *> :
  ...
```

represents a TCP/IP segment exchanged between network interfaces a and b.

Annotations are attached to states, transitions, and named signature actions by putting them just before the entity's opening brace. For example,

```
state s1 <* annotation *> { ... }
transition t1 (s1->s2) consuming
  <* annotation *> { ... }
```

An annotation in the scenario front matter is at scenario scope. An example of this kind is the set of conditions in NetSTAT scenarios that customize the attack signature to an installation-specific Factbase. These are specified using the NetSTAT CONSTRAINT annotation:

Example:

```
scenario example ( ... )
{
  Host server;
  Service x;
  ...

  <* CONSTRAINT
      server in Network.hosts &&
      x in server.services
  *>
  ...
}
```

In this example the annotation tag is CONSTRAINT and the argument list consists of one expression.

6   Experience with the language

The STATL/core framework has been used to generate a new tool suite based on the STAT approach. The tool suite includes USTAT, WinSTAT, and NetSTAT. The STAT tool suite was recently used in the 1999 DARPA Intrusion Detection Evaluation effort. For each tool in the suite an extension module containing the application-specific abstract event types and predicates was developed. In addition, an application-specific preprocessor was developed to translate the native audit records into abstract event representations. For each tool a Factbase and Analyzer were designed to translate STATL scenarios into executable plugins.

USTAT is the core-based implementation of the original Unix host-based IDS. It uses Sun Microsystems' Basic Security Module (BSM) as a source of audit data. The USTAT preprocessor filters the incoming audit records and translates the selected records into USTAT events, which are defined in the USTAT-specific extension module. USTAT uses only 35 of the 125 different audit record types generated by BSM, and those 35 are translated into 20 USTAT event types. The USTAT Factbase is composed of a number of *filesets* that characterize the security aspects of critical files and applications in the host's filesystem. For example, the "restricted-write-directories" fileset specifies a list of directories in which non-root users should not make changes (e.g., /bin, /usr/lib). Each fileset has one or more associated scenarios that define the meaning of the fileset. For example, associated with the "restricted-write-directories" fileset is a scenario that raises an alarm if any create, write, or delete action by a non-root user succeeds in one of the listed directories. USTAT's Analyzer makes various changes to scenarios to implement application-specific semantics that are not part of the core. For example, if a USTAT scenario instance depends on the existence of a process, then when that process exits the instance can be unwound. The USTAT Analyzer implements this by adding unwinding transitions to the original scenario as needed.

WinSTAT is a new tool that performs host-based intrusion detection in the Windows NT environment. WinSTAT uses the event logs produced by Windows NT as input and transforms a selection of NT events into WinSTAT events. The WinSTAT Factbase and Analyzer are very similar to those for USTAT.

NetSTAT is a network-based intrusion detection system. The core-based approach has been used to develop the NetSTAT probe component. A NetSTAT probe uses the network traffic sniffed from a network segment as input. It can also work offline by reading tcpdump files. The NetSTAT extension of the STAT core includes 22 event types and more than 50 predicates and functions. The NetSTAT

preprocessor is responsible for the filtering of relevant network packets and for the parsing and abstracting of network events. The NetSTAT preprocessor tasks include reassembling fragmented IP datagrams, reassembling TCP streams, parsing DNS and RPC events, and maintaining relevant information about the state of the network (e.g., active connections). The NetSTAT Factbase component is a repository that stores and manages the security relevant information about a network, such as the network topology and the network services deployed. The Analyzer uses the annotations in STATL scenarios to determine where the probes must be placed in the protected network and how the probes must be configured [33].

STATL has been used as a basis to develop scenarios for USTAT, WinSTAT, and NetSTAT. In the following sections examples of actual STATL scenarios are given for USTAT and NetSTAT respectively. The attacks presented are well-known simple examples. The intent is to allow the reader to concentrate on understanding the language, rather than the attack itself.

6.1 USTAT example

In the *ftp-write* attack, an attacker uses the *ftp* service to create a bogus `.rhosts` file in a world-writable *ftp* daemon home directory. Using the created file the attacker is then able to open a remote session using the *rlogin* service without being required to supply a password. A generalization of this attack is that an attacker creates a bogus `.rhosts` file in any other user's home directory and then uses it to be allowed to login without providing a password. This generalization of the *ftp-write* attack is depicted schematically in the STD in Figure 2. The different types of arrows are used to denote different types of transitions in STDs: a solid arc with a single arrowhead denotes a nonconsuming transition, a solid arc with a double arrowhead denotes a consuming transition, and a dashed arc denotes an unwinding transition. A text-based STATL specification of the generalized attack follows.

```
use ustat;
scenario ftp_write
{
  int user;
  int pid;
  int inode;

  initial state s0 { }

  transition create_file (s0 -> s1)
    nonconsuming
  {
    [WRITE w] : (w.euid != 0) &&
                     (w.owner != w.ruid)
    { inode = w.inode; }
  }
```

```
state s1 { }

transition login (s1 -> s2)
  nonconsuming
{
  [EXEC e] :
    match_name(e.objname, "login")
  {
    user = e.ruid;
    pid = e.pid;
  }
}

state s2 { }

transition read_rhosts (s2 -> s3)
  consuming
{
  [READ r] : (r.pid == pid) &&
                  (r.inode == inode)
}

state s3
{
  {
    string username;
    userid2name(user, username);
    log("remote user %s \
      gained local access",
      username);
  }
}
}
```

The sequence of events detected by this scenario is that a file is created (or written to) by a non-root user who doesn't own the directory containing the file, and then the login program runs and reads the suspicious file. `WRITE`, `EXECUTE`, and `READ` are USTAT-specific event types. The predicate `match_name()` and procedure `userid2name()` are also part of the USTAT extension. This signature was successfully used during the 1999 DARPA Intrusion Detection Evaluation, with the addition of three unwinding transitions, as shown in the STD in Figure 3:

```
transition delete_file1 (s1 -> s0)
  unwinding
{
  [DELETE d] : d.inode == inode
}

transition delete_file2 (s2 -> s0)
  unwinding
{
  [DELETE d] : d.inode == inode
}

transition logout (s2 -> s1)
  unwinding
{
  [EXIT e] : e.pid == pid
}
```

The `delete_file` transitions roll the attack back to the initial state, because no further progress can be made after
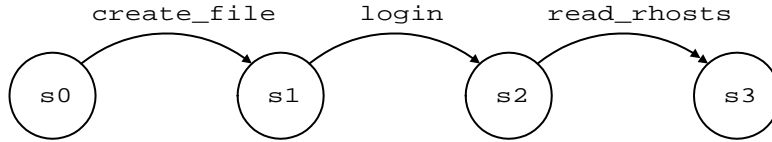
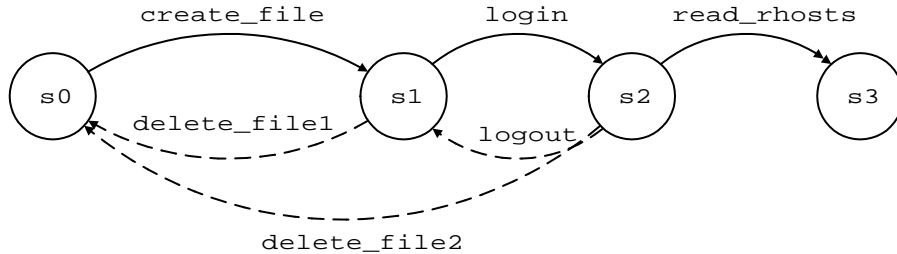Figure 2: *ftp-write* state transition diagram.

Figure 3: *ftp-write* state transition diagram with unwinding transitions.

the suspicious file is deleted. The `logout` transition rolls the attack back to state S_1, because after the login process exits, it obviously cannot read any files.

Even for a simple attack like *ftp-write* there is more than one way to specify the signature. For example, a more precise specification would check that the user in the login transition is the same as the user in whose home directory a file was created. This is done by changing the `create_file` and `login` transitions as follows:

```
transition create_file (s0 -> s1)
  nonconsuming
{
  [WRITE w] : (w.euid != 0) &&
                   (w.owner != w.ruid)
  {
    inode = w.inode;
    user = w.ruid;
  }
}

transition login (s1 -> s2)
  nonconsuming
{
  [EXEC e] :
    match_name(e.objname, "login") &&
    (e.ruid == user)
  { pid = e.pid; }
}
```

6.2 NetSTAT example

This example scenario detects half-open TCP connections. A TCP connection is in a half-open state when the three-way handshake of the TCP protocol is not completed and the server is waiting for the final acknowledgment from the client [25]. This attack is the building block for the *SYN-flooding* denial of service attack [4]. The goal of the *SYN-flooding* attack is to consume all the server-side resources dedicated to the setup of TCP connections, effectively preventing the server from accepting any other TCP connection. The STD for the `halfopen` scenario is shown in Figure 4. The STATL specification for this scenario follows.

```
use netstat;
scenario halfopentcp(int timeout)
{
  IPAddress victim_addr;
  Port victim_port;
  IPAddress attacker_addr;
  Port attacker_port;
  timer t0;

  initial state s0 {}

  transition SYN (s0 -> s1)
    nonconsuming
  {
    [IP ip [TCP tcp]] :
      (tcp.tcp_header.flags & TH_SYN) &&
      !(tcp.tcp_header.flags & TH_ACK)
    {
      victim_addr=ip.header.dst;
      victim_port=tcp.header.dst;
      attacker_addr=ip.header.src;
      attacker_port=tcp.header.src;
    }
  }

  state s1
  {
    { timer_start(t0, timeout); }
  }

  transition ACK (s1 -> s0)
    unwinding
  {
```
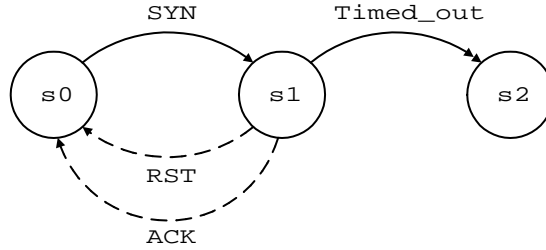
13

Figure 4: *Half-Open TCP* state transition diagram.

```
    [IP ip [TCP tcp]] :
      (ip.header.dst==victim_addr) &&
      (tcp.header.dst==victim_port) &&
      (ip.header.src==attacker_addr) &&
      (tcp.header.src==attacker_port) &&
      !(tcp.header.flags & TH_SYN) &&
      (tcp.header.flags & TH_ACK)
  }

transition RST (s1 -> s0)
  unwinding
  {
    [IP ip [TCP tcp]] :
      (ip.header.src==victim_addr) &&
      (tcp.header.src==victim_port) &&
      (ip.header.dst==attacker_addr) &&
      (tcp.header.dst==attacker_port) &&
      (tcp.header.flags & TH_RST)
  }

transition Timed_out (s1 -> s2)
  consuming
  {
    timer t0;
  }

state s2
{
  {
    HALFOPENTCP e;

    e = new HALFOPENTCP(attacker_addr,
                        attacker_port,
                        victim_addr,
                        victim_port,
                        start);
    enqueue_event(e, HALFOPENTCP, start);
  }
}
}
```

In the `halfopentcp` scenario the attacker sends a TCP SYN segment asking for the creation of a TCP connection. This event fires transition SYN. In the transition code block, the IP addresses and TCP ports involved are saved to local variables. Transition SYN leads to state `s1` and a timer is started using the scenario parameter `timeout` for the timeout value.

In state `s1` two events may invalidate the attack. First, the server may answer with an RST segment shutting down

the connection and freeing the resources dedicated to the establishment of the TCP connection. Second, the client may send the final ACK of the TCP setup hand-shake and switch the connection to an established state. These two events fire transitions RST and ACK respectively. The two transitions are both unwinding and bring the scenario to its initial state, meaning that this instance of the scenario does not correspond to an attack in progress.

If the handshake is not completed by the client and the server does not answer with an RST message, then the timer will fire and transition Timed_out will bring the scenario to state `s2`. In this state a new event of type HALFOPENTCP is created on the basis of the information gathered in the previous steps of the scenario. The event is encapsulated in a generic STAT event and sent to the language runtime. This event, called a *synthetic event*, is inserted in the network event stream and may be processed by other scenarios (e.g., a scenario to detect the *SYN-flooding* attack). Synthetic events enable forward chaining between different scenarios.

7 Conclusions and future work

The STATL language has been designed to meet the requirements for an attack description language as described in Section 3. The language is *simple* and explicitly focused on representing attack scenarios. The resulting language includes a few well-defined abstractions (scenario, state, transition, action) that are used to specify attack signatures. The language is *expressive*. Currently, 10 attack scenarios have been developed for USTAT, 5 attacks have been developed for WinSTAT, and more than 20 attacks have been coded for the NetSTAT tool. In encoding attacks no limit in the expressiveness of the language was found. Different types of attacks were coded: probes, denial-of-service, remote and local user-to-root attacks, threshold-based attacks, and even policy-based attacks. In addition, the modeled attacks differ in the level of abstraction. Some attacks exploit weaknesses in the TCP/IP protocol stack and are low level,

while others can be abstracted and detect a wide number of variations (for example all the local buffer overflow attacks considered so far are detected by a single USTAT/STATL scenario). The language is *rigorous*, that is, it has a precise syntax and a parser for the language. The semantics of the language has been defined operationally, by developing an *executor* for the language. STATL is also *extensible*. Describing attacks in both the network and host domains demonstrated that very different types of events and predicates could be easily developed and integrated into the language. The language extensibility also allows for the management of *heterogenous* data; that is, by composing different language extensions it is possible to develop a STATL extended language that is able to manage events from different audit streams (e.g., network packets and host audit records). Finally, STATL is *portable*. The tools used to develop scenarios and process the resulting executable representations have been developed in portable languages like Java and Tcl and by using a development environment (the GNU Build system), which allows for easy porting to new platforms.

The first prototype of the STATL development environment is almost complete. The translator tool has been implemented, but the application-specific Analyzers have not, so some of the customization and configuration is currently done manually. The language extension process has been recently redesigned as an object-oriented framework supporting the creation of IDS extensions. By using the features of object-oriented software development, it is possible to automate some of the tasks involved in developing event types and predicates for new environments and domains.

To prove the general applicability of STATL to state-based intrusion detection, an analysis of the possible translation between STATL and other attack languages is being conducted. For example, Snort [27] and NFR [26] have been investigated, and STATL subsets that are translatable to each have been identified. Additional attack languages (e.g., P-BEST [19]) will be analyzed in the future, and prototype translators will be developed for some.

Other future work will be focused on the translation process and on the improvement of the features of the STATL language. In addition, attack signatures will be collected and new STATL scenarios and STATL extensions will be developed.

Acknowledgments

References

[1] M. Bishop. Standard Audit Trail Format. In *Proceedings of the 1995 National Information Systems Security Conference*, pages 136–145, Baltimore, Maryland, October 1995.

[2] CISCO. NetRanger Intrusion Detection System. Technical Information, April 1999.

[3] D. Curry. Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition. `draft-ietf-idwg-idmef-xml-01.txt`, July 2000.

[4] daemon9. Project neptune. *Phrack Magazine*, 7(48), July 1996.

[5] R. Deraison. *The Nessus Attack Scripting Language Reference Guide*, 2000. `http://www.nessus.org`.

[6] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Addendum to "Testing and Evaluating Computer Intrusion Detection Systems". *CACM*, 42(9):15, September 1999.

[7] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Testing and Evaluating Computer Intrusion Detection Systems. *CACM*, 42(7):53–61, July 1999.

[8] S. Eckmann, G. Vigna, and R. Kemmerer. STATL. Technical report, UCSB, 2000.

[9] A.K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Application Conference (ACSAC'98)*, Scottsdale, AZ, December 1998.

[10] Common Intrusion Detection Framework Working Group. A CISL Tutorial. *http://www.gidos.org/tutorial.html*, 2000.

[11] Common Intrusion Detection Framework Working Group. Common Intrusion Detection Framework Specification. *http://www.gidos.org/*, 2000.

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

[13] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 1993.

[14] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[15] Internet Security Systems. *Introduction to RealSecure Version 3.0*, January 1999.

[16] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[17] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, 1997.

[18] MIT Lincoln Laboratory. DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/IST/ideval/, 1999.

[19] U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

[20] S. McCanne, C. Leres, and V. Jacobson. Tcpdump 3.4. Documentation, 1998.

[21] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix Namur (Belgium), September 1997.

[22] Secure Networks. *Custom Attack Simulation Language (CASL)*, January 1998.

[23] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[24] P.A. Porras. STAT – A State Transition Analysis Tool for Intrusion Detection. Master's thesis, Computer Science Department, University of California, Santa Barbara, June 1992.

[25] J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[26] M.J. Ranum, K. Landfield, M. Stolarchuck, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool for Network Monitoring. In *Eleventh Systems Administration Conference (LISA '97)*. USENIX, October 1997.

[27] M. Roesch. *Writing Snort Rules: How To write Snort rules and keep your sanity.* http://www.snort.org.

[28] R. Sekar and P. Uppuluri. Synthesizing Fast Intrusion Detection/Prevention Systems from High-Level Specifications. In *Proceedings of the USENIX Security Symposium*, 1999.

[29] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.

[30] syslog(3). UNIX documentation.

[31] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[32] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection Approach. In *Proceedings of the 14th Annual Computer Security Application Conference*, Scottsdale, Arizona, December 1998.

[33] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[34] World Wide Web Consortium (W3C). Extensible Markup Language (XML). W3C Recommendation, February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.