Chapter 1

SENSOR FAMILIES FOR INTRUSION DETECTION INFRASTRUCTURES

Richard A. Kemmerer and Giovanni Vigna
Reliable Software Group
Department of Computer Science
University of California Santa Barbara
[kemm,vigna]@cs.ucsb.edu

Abstract Intrusion detection relies on the information provided by a number of *sensors* deployed throughout a protected network. Sensors operate on different event streams, such as network packets and application logs, and provide information at different abstraction levels, such as low-level warnings and correlated alerts. In addition, sensors range from lightweight probes and simple log parsers to complex software artifacts that perform sophisticated analysis. Therefore, deploying, configuring, and managing, a large number of heterogeneous sensors is a complex, expensive, and error-prone activity.

Unfortunately, existing systems fail to manage the complexity that is inherent in today's intrusion detection infrastructures. These systems suffer from two main limitations: they are developed *ad hoc* for certain types of domains and/or environments, and they are difficult to configure, extend, and control remotely.

To address the complexity of intrusion detection infrastructures, we developed a framework, called STAT, that overcomes the limitations of current approaches. Instead of providing yet another system tailored to some domain-specific requirements, STAT provides a software framework for the development of new intrusion detection functionality in a modular fashion.

According to the STAT framework, intrusion detection sensors are built by dynamically composing domain-specific components with a domain-independent runtime. The resulting intrusion detection sensors represent a software family. Each sensor has the ability to reconfigure its behavior dynamically. The reconfiguration functionality is supported by a component model and by a control infrastructure, called MetaSTAT. The final product of the STAT framework is a highly-configurable, well-integrated intrusion detection infrastructure.

Keywords: Security, Intrusion Detection, Intrusion Detection Infrastructures, Intrusion Detection Frameworks, Software Engineering, STAT.

Introduction

In recent years, networks have evolved from a mere means of communication to a ubiquitous computational infrastructure. Networks have become larger, faster, and highly dynamic. In particular, the Internet, the world-wide TCP/IP network, has become a mission-critical infrastructure for governments, companies, financial institutions, and millions of everyday users.

The surveillance and security monitoring of the network infrastructure is mostly performed using Intrusion Detection Systems (IDSs). These systems analyze information about the activities performed in computer systems and networks, looking for evidence of malicious behavior. Attacks against a system manifest themselves in terms of events. These events can be of a different nature and level of granularity. For example, they may be represented by network packets, operating system calls, audit records produced by the operating system auditing facilities, or log messages produced by applications. The goal of intrusion detection systems is to analyze one or more event streams and identify manifestations of attacks.

Event streams are used by intrusion detection systems in two different ways, according to two different paradigms: *anomaly detection* and *misuse detection*. In anomaly detection systems [14, 17, 7, 34], 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 [21, 23, 28, 20, 13]. Misuse detection systems 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.

The intrusion detection community has developed a number of different tools that perform intrusion detection in particular domains (e.g., hosts or networks), in specific environments (e.g., Windows NT or Solaris), and at different levels of abstraction (e.g., kernel-level tools and alert correlation systems). These tools suffer from two main limitations: they are developed *ad hoc* for certain types of domains and/or environments, and they are difficult to configure, extend, and control remotely.

In the specifi c case of signature-based intrusion detection systems, the sensors are equipped with a number of attack models that are matched against a stream of incoming events. The attack models are described using an *ad hoc*, domain-specifi c language (e.g., NFR's N-code [27]). Therefore, performing intrusion detection in a new environment requires the development of both a new system and a new attack modeling language. As intrusion detection is applied to new and previously unforeseen domains, this approach results in increased development effort.

Today's networks are not only heterogeneous; they are also dynamic. Therefore, intrusion detection systems need to support mechanisms to dynamically change their confi guration as the security state of the protected system evolves. The confi guration and management of a large number of sensors raises multiple issues.

One issue is the static confi guration of the data sources used for analysis. The *ad hoc* nature of existing IDSs does not allow one to dynamically confi gure a running sensor so that new event streams can be used as input for the security analysis. This is a limitation because new attacks may have manifestations in event streams that are not currently analyzed by a specifi c IDS. Being bound statically to a single source of events may result in limited effectiveness.

A second issue is the static confi guration of the attack models used for analysis. Most existing intrusion detection systems (e.g., [28]) are initialized with a set of signatures at startup time. Updating the signature set requires stopping the IDS, adding new signatures, and then restarting execution. Some of these systems provide a way to enable/disable some of the available signatures, but few systems allow for the dynamic inclusion of new signatures at execution time.

A third issue is the relatively static confi guration of responses in existing intrusion detection systems. In most cases it is possible to choose only from a specifi c subset of possible responses. In addition, to our knowledge, none of the systems allows one to associate a response with *intermediate* steps of an attack. This is a severe limitation, especially in the case of distributed attacks carried out over a long time span.

Finally, managing a large number of sensors requires an effective control infrastructure. Most systems provide some sort of management console that allows the Security Administrator to remotely tune the confi guration of specifi c sensors. This reconfi guration procedure is mostly performed manually and at a very low level. This task is particularly error-prone, especially if the intrusion detection sensors are deployed across a very heterogeneous environment and with very different confi gurations. The challenge is to determine if the current confi guration of one or more sensors is valid or if a reconfi guration is meaningful.

This chapter describes a framework for the development of intrusion detection systems, called STAT, and a sensor control infrastructure, called MetaSTAT, which have been developed to address the issues above and to overcome the limitations of existing approaches.

The STAT framework includes a domain-independent attack modeling language and a domain-independent event analysis engine. The framework can be extended in a well-defined way to match new domains, new event sources, and new responses. The framework has been used by the authors to develop a number of different intrusion detection systems, from a network-based intrusion detection system, to host-based and application-based systems, to alert correlators.

The resulting set of intrusion detection systems can be seen, in Software Engineering terms, as a *software family*. Members of the family share a number of features, including dynamic reconfigurability and a fine-grained control over a wide range of characteristics [33]. The STAT framework is the only known framework-based approach to the development of intrusion detection systems. Our experience with the framework shows that by following this approach it is possible to develop intrusion detection systems with reduced development effort, with respect to an *ad hoc* approach. In addition, the approach is advantageous in terms of the increased reuse that results from using an object-oriented framework and a component-based approach.

The configuration of sensors in the STAT family can be controlled at a very fine grain using the MetaSTAT infrastructure. MetaSTAT provides the basic mechanisms to reconfigure, at run-time, which input event streams are analyzed by each sensor, which scenarios have to be used for the analysis, and what types of responses must be carried out for each stage of the detection process. In addition, MetaSTAT supports the explicit modeling of the dependencies among the modules composing a sensor so that it is possible to automatically identify the steps that are necessary to perform a reconfiguration of the deployed sensing infrastructure.

The result of applying the STAT/MetaSTAT approach is a “web of sensors”, composed of distributed components integrated by means of a communication and control infrastructure. The task of the web of sensors is to provide fine-grained surveillance inside the protected network. The web of sensors implements *local surveillance* against both outside attacks and local misuse by insiders in a way that is complementary to the mainstream approach where a single point of access (e.g., a gateway) is monitored for possible malicious activity. Multiple webs of sensors can be organized either hierarchically or in a peer-to-peer fashion to achieve scalability and to be able to exert control over a large-scale infrastructure from a single control location.

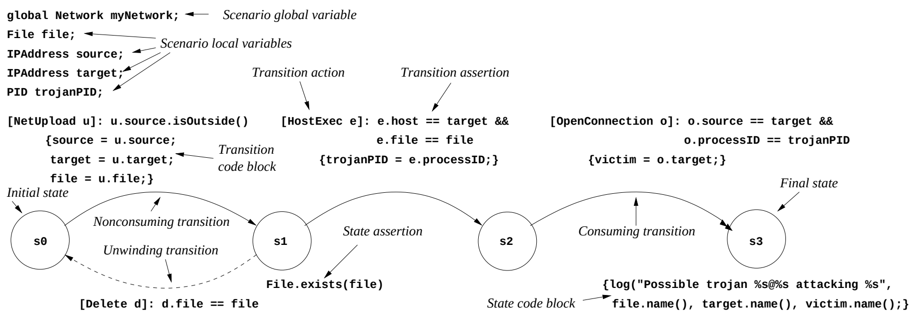
This chapter is structured as follows. Section 1.1 introduces the STAT framework. Section 1.2 presents a family of intrusion detection systems developed using the framework. Section 1.3 describes the MetaSTAT control infrastructure, shared by all the IDSs in the family. Section 1.4 presents relevant related work. Finally, Section 1.5 draws some conclusions.

1. The STAT Framework

The *State Transition Analysis Technique* [10] is a methodology to describe computer penetrations as *attack scenarios*. Each attack scenario is represented as a sequence of transitions that characterize the evolution of the security state of a system. In an attack scenario *states* represent snapshots of a system's security-relevant properties and resources. A description of an attack has an "initial" starting state and at least one "compromised" ending state. States are characterized by means of *assertions*, which are predicates on some aspects of the security state of the system. For example, in an attack scenario describing an attempt to violate the security of an operating system, assertions would state properties such as file ownership, user identification, or user authorization. *Transitions* between states are annotated with *signature actions* that represent the key actions that if omitted from the execution of an attack scenario would prevent the attack from completing successfully. For example, in an attack scenario describing a network port scanning attempt, a typical signature action would include the TCP segments used to test the TCP ports of a host.

The characterization of attack scenarios in terms of states and transitions allows for an intuitive graphic representation by means of *state transition diagrams*. Figure 1.1 shows a state transition diagram for a pedagogical example of a STATL attack scenario specification. The attack scenario detects a Trojan horse attack, where an apparently benign program (e.g., an MP3 player) is first downloaded by a user (first transition), and then installed and executed (second transition). The Trojan horse program contains "hidden" functionality (the warriors hidden in the Trojan horse) that allows the creator of the program to take control of the user's account. When executed, the Trojan horse opens a network connection back to an attacker controlled host that is outside the local network, and it waits for commands to be executed (third transition). When the scenario reaches the final state (represented as a double circle) the attack is considered completed. Note that even though this scenario is fairly representative of this type of attack, it is not to be considered a complete, detailed specification.

In the early 1990s, the State Transition Analysis Technique was applied to host-based intrusion detection, and a system, called USTAT [8, 9, 25], was developed. USTAT used state transition representations as the basis for rules to interpret changes in a computer system's state and to detect intrusions in real-time. The changes in the computer system's state were monitored by leveraging the auditing facilities provided by security-enhanced operating systems,



```
global Network myNetwork;
File file;
IPAddress source;
IPAddress target;
PID trojanPID;

[NetUpload u]: u.source.isOutside()
     {source = u.source;
      target = u.target;
      file = u.file;}

[HostExec e]: e.host == target &&
              e.file == file
     {trojanPID = e.processID;}

[OpenConnection o]: o.source == target &&
                    o.processID == trojanPID
     {victim = o.target;}

File.exists(file)

[Delete d]: d.file == file

{log("Possible trojan %s@%s attacking %s",
 file.name(), target.name(), victim.name();}
```

Figure 1.1. A sample state transition diagram of an attack scenario. The attack is a very simplified version of a Trojan horse installation attack. The first transition is fired when the upload of a file from a host outside the local network is detected. The second transition fires when the same file is executed. The final transition fires when the program being executed opens a network connection to another host.

such as Sun Microsystems' Solaris equipped with the Basic Security Module (BSM) [30]. The first implementation of USTAT clearly demonstrated the value of the STAT approach, but USTAT was developed in an *ad hoc* way and several characteristics of the first USTAT prototype were difficult to modify or to extend to match new environments (e.g., Windows NT/2000).

During the 90s, the focus of intrusion detection shifted from the host and its operating system to the network and the protocols used to exchange data. Therefore, the natural evolution of state transition analysis was its direct application to networks. The NetSTAT intrusion detection system was the result of this evolution [32]. NetSTAT was aimed at real-time state-transition analysis of network data. The NetSTAT system proved that the STAT approach could be extended to new domains. However, NetSTAT was also developed *ad hoc*, by building a completely new IDS that would fit the new domain.

In 1998, both NetSTAT and USTAT were used to participate in a DARPA-sponsored intrusion detection evaluation effort. The evaluation exercises included off-line analysis of audit logs and traffic dumps provided by the MIT Lincoln Laboratory [19] and the installation of the systems in a large testbed at the Air Force Research Laboratory (AFRL) [3, 4]. Intrusion detection systems from a number of universities, research centers, and companies were tested with respect to different classes of attacks, including port scans, remote compromise, local privilege escalation, and denial-of-service attacks. A detailed description of the attacks used in the MIT Lincoln Laboratory evaluation can be found in [16]. In both efforts the STAT-based systems performed very well and their combined results scored at the highest level in the evaluations.

Participating in this event gave strong positive feedback on the research that had been performed so far, and it also gave new insights into the STAT approach. In particular, running NetSTAT and USTAT at the same time revealed a number of similarities in the way attack scenarios were represented and in the runtime architecture of the systems. A closer analysis of the mechanisms used by the STAT-based systems to match attack scenarios against a stream of events suggested that the STAT-based IDSs could be redesigned as a family of systems that leverages an object-oriented framework.

The approach taken was to factor-out the mechanisms and techniques used by the intrusion detection analysis and to design an extension process that would support the development of intrusion detection systems for many different target environments. The result of this redesign was the *STAT Framework*. The STAT Framework consists of a domain-independent language, called *STATL*, and a runtime for the language, called the *STAT Core*. These elements can be extended following a well-defined process to match a specific target domain. Section 1.1.1 presents STATL, Section 1.1.2 describes the STAT Core, and Section 1.1.3 describes the framework extension process.

1.1 STATL

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 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 (STDs). 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.

1.1.1 STATL Overview. 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); 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 can cause the scenario to move to a new state. For example, an action can 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 could possibly match the action. For example, an assertion can 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 for several occurrences of the same attack to be 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*, *non-consuming*, 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. An example of a nonconsuming transition is given in Figure 1.1. The transition between states `s1` and `s2` represents the execution of a file. This step does not invalidate the previous state, that is, another execution of the program may occur. Semantically, the firing of a nonconsuming transition causes the creation of a new scenario instance. The original instance is still in the original state, while the new instance is in 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. The transition between states `s2` and `s3` in Figure 1.1 is an example of a consuming transition. The transition is fired when the executed Trojan program opens a connection. This invalidates state `s2`. It is no longer necessary to check if the program is opening a network connection since the program has already been identified as a Trojan. Unwinding transitions represent a form of "rollback" and they are used to describe events and conditions that invalidate the progress of one or more scenario instances and require the return to an earlier state. The transition between states `s1` and `s0` in the example in Figure 1.1 is an unwinding transition. The deletion of the uploaded file invalidates the condition needed for the attack to complete, and, therefore, the scenario instance is brought back to the previous state before the file was created.

1.1.2 STATL Syntax. This section presents STATL's syntax. 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.

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 standard 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).

Data Types. STATL includes several built-in types: `int` and `u_int` in various sizes, `bool`, `string`, `timeval` (for timestamps), and `timer`. 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 (see Section 1.1.3). For example, network-based scenarios may use different types than host-based scenarios, but both use `int` and `timeval`.

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 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, `host` and `count`. Parameters are accessible by the scenario instances as global constants.

Front Matter. Scenarios may declare constants and variables:

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

ConstDecl ::=
const *Type ConstId* { ’[’ [*size*] ’]’ } ‘=’ *InitialValue* ‘;’
VarDecl ::=
[**global**] *Type VarId* { ’[’ [*size*] ’]’ } [‘=’ *InitialValue*] ‘;’

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 tcpip;
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 other instances of the scenario. The variable declaration in the example also includes a variable named `server` of type `Host` (a type defi ned in the network-based language extension called `tcpip`). Because `server` is a local variable (i.e., its declaration does not contain the keyword **global**), each instance of the scenario will have its own copy of `server`.

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 an assertion and a code block, but these elements are optional:

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

Exactly one state must be designated as the initial state. When a scenario plugin is loaded into an IDS a first 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.

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 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)
    '{'
        ( '[' EventSpec ']' | ActionId )
        [':' Assertion]
        [CodeBlock]
    '}'
```

A transition's event is specified either directly (see section on EventSpecs) or by reference to a named signature action (see section on NamedSigAction). 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 later, 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 later), is as follows:

1 evaluate the transition assertion. If True, then

2 evaluate the state assertion. If True, then

3 execute the transition code block, possibly modifying local and global environments, and then

4 execute the state code block, possibly modifying local and global environments[1].

Transitions are deterministic, which means that every enabled transition fires if its assertion and the destination state's assertion are satisfied. 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 bsm, unix;
```

```
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. Note that this scenario uses both `bsm` and `unix` extensions, which define BSM events and UNIX-related abstractions, respectively.

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. 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 one of the events in the set.

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).

NamedSigAction. A named signature action has a name and specifies an event spec:

NamedSigAction ::=
    action *ActionId*
    '{'
        ('[' *EventSpec* ']' | *ActionId*)
        [':' *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 bsm, unix;
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
  }
  ...
}
```

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:

```
use bsm, unix;
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)
  }
  ...
}
```

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.

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`. Timer expiration is treated as an event, and these events may be matched by using "timer events" as transition event specs.

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`). 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.

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.

1.2 STAT Core

The STAT Core module is the runtime for the STATL language. The Core implements the concepts of state, transition, timer, etc. In addition, the Core performs the event processing task, which is the basic mechanism used to detect intrusions by matching event streams against attack scenarios.

The STAT Core module has an event-based multi-threaded architecture (see Figure 1.2). Events are sent to or received from the Core through four separate event queues.

- The *control queue* is used to send control events to the Core. These events modify the Core's behavior or its configuration (e.g., by requesting the activation of a new attack scenario).
- The *info queue* is used by the Core to publish control-related information, such as the result of a reconfiguration request. The events in this

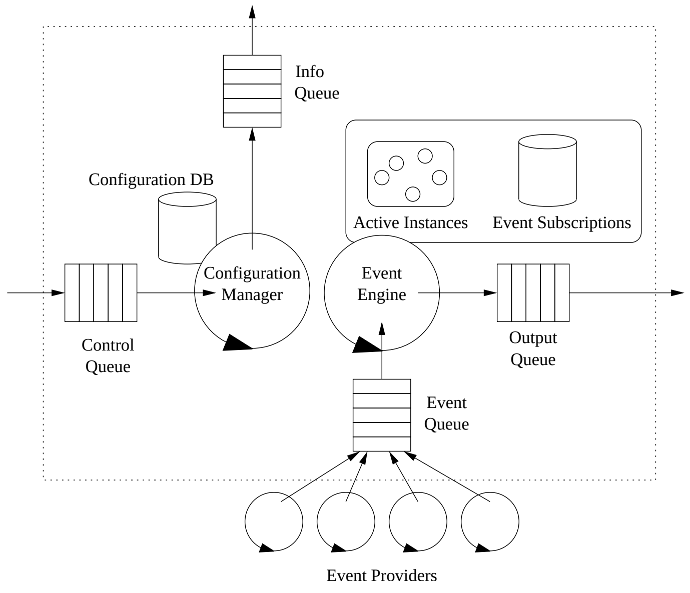


Figure 1.2. The STAT Core Architecture.

queue are used by external components (e.g., a MetaSTAT Proxy, see Section 1.3) to monitor the status of a Core component.

- The *input queue* is the source of the event stream for the intrusion detection analysis. Multiple external Event Providers (see Section 1.1.3) can contribute events to this queue.

- The *output queue* is used by the Core to publish events related to the intrusion detection process, such as detection alerts. This event queue can be connected to the input event queue of another Core component to realize a multi-core pipelined architecture.

The most important task of the Core is to keep track of active attack instances, which are called, in STATL terms, scenario instances. The Core maintains a data structure for each scenario instance. The data structure contains the current state of the scenario, its local environment, and the list of transitions that are are enabled, that is, the transitions that may possibly fi re. These transitions have an associated action and a corresponding assertion, which, together, represent the subscription for an event of interest. The set of all current event subscriptions for all the active scenario instances is maintained by the Core in an internal database.

The Event Engine component of the Core is responsible for extracting events from the input queue and matching each event against the active event subscriptions. For each matching event subscription the tuple $\langle$*scenario*, *transition*, *event*$\rangle$ is inserted in the set of transitions to be fired. There are three separate sets depending on the type of transition: nonconsuming, consuming, and unwinding.

Once all the enabled transitions have been collected, the transitions are fired one by one. First, nonconsuming transition are fired. When a nonconsuming transition of a scenario instance is fired, a new scenario instance is created. The original instance becomes the *parent* of the new instance which, in turn, becomes one of the original instance's *children*. The child instance has a copy of the parent's local environment and a copy of the parent's timers. The state of the child instance is set to the destination state of the transition that fired. Then, the destination state code fragment is executed in the context of the child instance. If the destination state is a final state the child instance is removed. Otherwise, for each outgoing transition of the destination state a subscription for the associated event is inserted in the event subscription database.

After all the nonconsuming transitions have been fired, consuming transitions are fired. In the most common case, the instance state is changed to the destination state, previous subscriptions are canceled, and new subscriptions for the events associated with the transitions outgoing from the new state are inserted in the event spec database. Then, the destination state code is executed. If there are multiple enabled consuming transitions to be fired associated with the same scenario instance, then for each transition firing, except for the last one, a *clone* of the scenario instance is created. A cloned instance differs from a child instance in that a clone instance has the same parent as the original instance. After the creation of the clone, the execution process follows the steps of the previous case. Another special case is represented by a scenario instance that is in a state that can be the destination of an unwinding transition, that is an *unwindable state*. In this case, if the instance has any descendants, it is possible that at some time in the future one of the descendants may want to unwind to the ancestor instance as it is in its current state. If the instance's state changes because of the firing of a consuming transition, the system would reach an inconsistent state. To avoid this, a clone instance is created and the original instance is put in an *inactive status*. In the inactive status, the current subscriptions of the instance are removed and they are not replaced with new subscriptions. The instance will be restored to an active status if one of the children actually unwinds to the instance in the specified state.

After both consuming and nonconsuming transitions have been fired, the Core proceeds to fire the unwinding transitions. The firing of an unwinding transition with respect to a scenario instance has the effect of undoing the steps that brought the scenario instance to its current state. This means that other scenario instances may be affected by the unwinding procedure. More precisely, if we consider an unwinding transition from state S_x to state S_y we have to

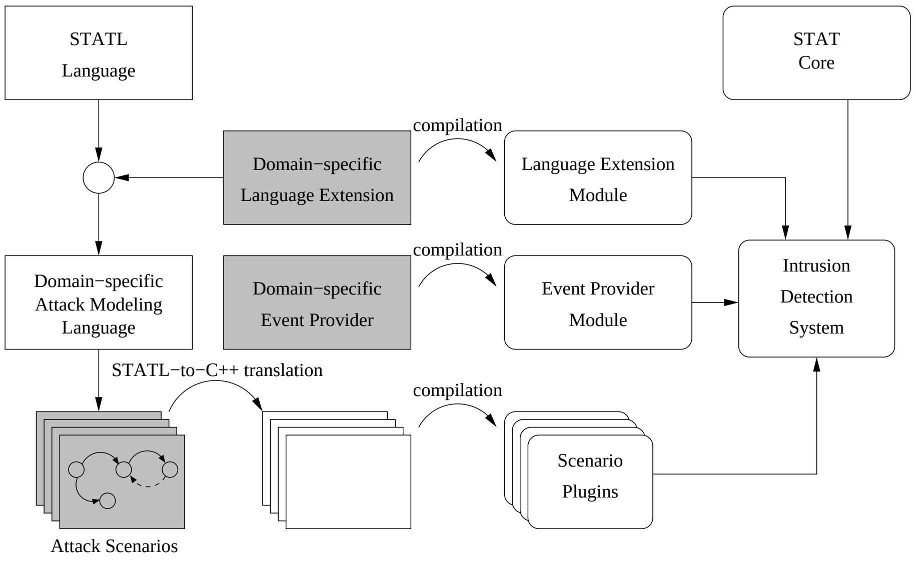

STATL Language
STAT Core
compilation
Domain−specific Language Extension
Language Extension Module
compilation
Domain−specific Attack Modeling Language
Domain−specific Event Provider
Event Provider Module
Intrusion Detection System
STATL−to−C++ translation
compilation
Scenario Plugins
Attack Scenarios

Figure 1.3. The STAT Framework extension process. The grayed boxes are the modules that need to be developed to extend the framework. The other components are generated automatically through either translation or compilation.

remove all the instances that were created by the series of events that brought the unwinding instance from state S_y to state S_x. In the Core, this is achieved by traversing back the parent/child chain until an instance in state S_y is found. Then the instance subtree rooted in the last visited instance is removed.

After all the transitions have been fi red, the Confi guration Manager component takes control of the Core. If a new control message is found in the control queue, the necessary reconfi guration of the Core is performed, and then the event processing is resumed in the new confi guration.

1.3 STAT Extensions

The STATL language and the Core runtime are domain-independent. They do not support any domain-specifi c features that may be necessary to perform intrusion detection analysis in particular domains or environments. For example, network events such as an IP packet or the opening of a TCP connection cannot be represented in STATL natively. Therefore, the STAT Framework provides a number of mechanisms to extend the STATL language and the runtime to match the characteristics of a specifi c target domain.

The framework extension process is performed by developing subclasses of existing STAT Framework C++ classes. The framework root classes are `STAT_Event`, `STAT_Type`, `STAT_Provider`, `STAT_Scenario`, and `ST-AT_Response`. In the following paragraphs, the extension process is presented in detail. A graphic description of the extension process is given in Figure 1.3.

The first step in the extension process is to create the events and types that characterize a target domain. A STAT event is the representation of an element of an event stream to be analyzed. For example, an `IP` event may be used to represent an IP datagram that has been sent on a link. The event stream is composed of IP datagrams and other event types, such as Ethernet frames and TCP segments. All event types must be subclasses of the `STAT_Event` class. Basic event types can be composed into complex tree structures. For example, it is possible to use a tree of events to express encapsulation, such as Ethernet frames that encapsulate IP datagrams, which, in turn, contain TCP segments.

All of the types used to describe the components of an event and other auxiliary data structures must be subclasses of the `STAT_Type` class. For example, the `IPAddress` class is a type used in the definition of the `IP` event, and, therefore, it is a subclass of `STAT_Type`.

A set of events and types that characterize the entities of a particular domain is called a *Language Extension*. The name comes from the fact that the events and types defined in a Language Extension can be used when writing a STATL scenario once they are imported using the `use` STATL keyword. For example, if the `IP` event and the `IPAddress` type are contained in a Language Extension called `tcpip`, then by using the expression `use tcpip` it is possible to use `IP` events and `IPAddress` objects in attack scenario descriptions.

The events and types defined in a Language Extension must be made available to the runtime. Therefore, Language Extensions are compiled into dynamically linked libraries (i.e., a "`.so`" file in a UNIX system or a DLL file in a Windows system). The Language Extension libraries are then loaded into the runtime whenever they are needed by a scenario.

Attack scenarios are written in STATL, extended with the relevant Language Extensions. For example, a signature for a port scanning attack can be expressed in STATL extended with the `tcpip` Language Extension. STATL attack scenarios are then automatically translated into a subclass of the `STAT_Scenario` class. Finally, the attack scenarios are compiled into dynamically linked libraries, called *Scenario Plugins*. When loaded into the runtime, Scenario Plugins analyze the incoming event stream looking for events or sequences of events that match the attack description.

Once Language Extensions and Scenario Plugins are loaded into the Core it is necessary to start collecting events from the environment and passing them to the STAT Core for processing. The input event stream is provided by one or more *Event Providers*. An Event Provider collects events from the external environment (e.g., by obtaining packets from the network driver), creates STAT events as defined in one or more Language Extensions, and inserts these events into the event queue of the STAT Core.

Event Providers are created by subclassing the `STAT_Provider` framework class. This class defines a minimal set of methods for initialization/finalization of a provider and the retrieval of events from the environment. An Event Provider component is compiled into a dynamically linked library. An Event Provider library module can be loaded into the STAT Core at runtime. Once a Provider has been loaded, it has to be activated with specific parameters. The activated Event Provider will then start collecting events from the external environment. A single Event Provider can be activated in many instances and many different Event Providers can be loaded and activated at one time. Each activation of an Event Provider is associated with a dedicated thread. The thread uses the functions defined in the Event Provider module to retrieve events from the environment and insert them into the Core event queue for processing.

A runtime equipped with Language Extensions, Scenario Plugins, and Event Providers represents a functional intrusion detection system. In addition, the STAT Framework also provides classes that define *Response Modules*. A Response Module is created by subclassing the `STAT_Response` class. A Response Module contains a library of actions that may be associated with the evolution of a scenario. For example, a network-based response action could reset a TCP connection, or it could send an email to the Network Security Officer. Response Modules are compiled into dynamically linked libraries that can be loaded into the runtime at any moment. Functions defined in a Response Module can be associated with any of the states defined in a Scenario Plugin that has been loaded in the runtime. This mechanism provides the ability to associate different types of response functions with the intermediate steps of an intrusion.

Figure 1.4 presents the high-level class structure of the STAT Framework. The classes in the top part of the hierarchy are the STAT Framework classes. The lower part of the hierarchy is represented by the classes used to create a simple network-based intrusion detection system. The Language Extension Module is created by extending `STAT_Event` with subclasses `IP`, `UDP`, and `TCP`, which represent instances of the corresponding protocol units. The `STAT_Type` class is subclassed by `IPAddress` and `Port`, which are used to represent IP addresses and TCP/UDP ports, respectively. `NetSniffer` is an Event Provider (a subclass of `STAT_Provider`) that reads the packets sent on a network link and creates instances of the `IP`, `UDP`, and `TCP` events. The three subclasses `UDPFlood`, `RemoteBufferOverflow`, and `Portscan` extend the framework with descriptions of three network-based attacks. Finally, the subclass `NetworkResponse` contains network-specific response functions such as firewall reconfiguration directives and TCP connection shutdown.

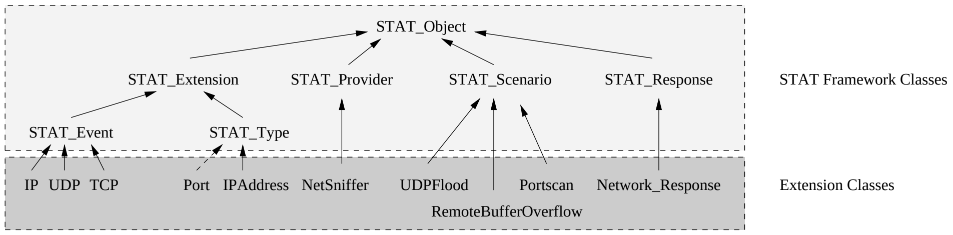


Figure 1.4. The STAT Framework class hierarchy.

2. The STAT Family

The framework described in the previous section has been used to develop a number of STAT-based intrusion detection systems. These IDSs are constructed by extending the STAT runtime with a selection of Language Extensions, Event Providers, Scenario Plugins, and Response Modules.

To be more precise, we developed an application, called *xSTAT*, that acts as a generic wrapper around the STAT Core runtime. xSTAT can be configured with different components. For example, xSTAT may load a network-centered Language Extension (e.g., the `tcpip` extension described in Section 1.1), a network-based Event Provider, and some network attack scenarios. The resulting system would be a network-based intrusion detection system, similar to Snort [28] or ISS RealSecure [13]. Note that loading a different set of components would create a completely different IDS. In addition, the STAT Framework has been ported to a number of platforms, including Linux, Solaris, Windows NT/2000/XP, FreeBSD, and MacOS X. Therefore, it is possible to create IDSs for these platforms by recompiling the necessary components.

By extending the STAT runtime with different modules it is possible to produce a potentially unlimited number of IDSs. In the past few years, we concentrated on the most important applications of intrusion detection, and we developed a family of intrusion detection systems based on the STAT Framework. The following subsections give a brief description of the current toolset.

2.1 USTAT

USTAT was the first application of the STAT technique to host-based intrusion detection. Even though the type of analysis that is performed on the event stream has mostly remained unchanged, the tool architecture has been completely re-designed [25]. USTAT performs intrusion detection using BSM audit records [30] as input. The record contents are abstracted into events described in a BSM-based Language Extension. USTAT also uses a UNIX-centered Language Extension that contains the definitions of a number of UNIX entities, such as user, process, and file. USTAT uses a BSM-based Event

Provider that reads BSM events as they are produced by the Solaris auditing facility, transforms them into STAT events, and passes them to the STAT Core. The events are matched against a number of Scenario Plugins that model different UNIX-based attacks, such as buffer overflows and access to sensitive files by unprivileged applications.

2.2 NetSTAT

NetSTAT is a network-based IDS composed of a network-centered Language Extension, an Event Provider that collects traffic from traffic dumps or network links, and a number of scenarios that describe network-based attacks, such as scanning attacks, remote-to-local attacks, and traffic spoofing. NetSTAT is similar to other network-based intrusion detection systems. However, it has some unique features that are the result of being part of the STAT family. For example, NetSTAT scenarios can be written in a well-defined language that has a precise semantics [5]. In addition, it is possible to perform stateful analysis that takes into account the multi-step nature of some attacks. This is in contrast to most existing network-based intrusion detection systems, which are limited to the analysis of single packets and do not provide a well-defined language for the description of multi-step scenarios.

2.3 WebSTAT and logSTAT

WebSTAT and logSTAT are two systems that operate at the application level. They both apply STAT analysis to the events contained in log files produced by applications. More precisely, WebSTAT parses the logs produced by Apache web servers [1], and logSTAT uses UNIX syslog files as input. In both cases, Language Extension modules that define the appropriate events and types have been developed, as well as Event Providers that are able to parse the logs and produce the corresponding STAT events.

2.4 AlertSTAT and afedSTAT

AlertSTAT is a STAT-based intrusion detection system whose task is to fuse, aggregate, and correlate alerts from other intrusion detection systems. Therefore, AlertSTAT uses the alerts produced by other sensors as input and matches them with respect to attack scenarios that describe complex, multi-step attacks. For example, an AlertSTAT scenario may identify the following three-step attack. The first step is a scanning attack detected by a network-based intrusion detection system, such as Snort or NetSTAT. This is followed by a remote buffer overflow attack against a Web Server (as detected by WebSTAT). Next, an alert produced by a host-based intrusion detection system (e.g., USTAT) located on the victim host indicates that the Apache process is trying to access the `/etc/exports` file on the local machine. The resulting alert is an aggregated report that conveys a much higher level view of the overall attack process.

AlertSTAT operates on alerts formatted according to the IETF's Intrusion Detection Message Exchange Format (IDMEF) proposed standard [2]. The application is built by composing an IDMEF-based Language Extension with an Event Provider that reads IDMEF events from files and/or remote connections and feeds the resulting event stream to the STAT Core. A number of attack scenarios have been developed, including the detection of complex scans, "many-to-one" and "one-to-many" attacks, island hopping attacks, and privilege escalation attacks.

Another correlator, called *afedSTAT*, has also been developed. The afedSTAT IDS uses the events contained in a database of alerts, called AFED, which was developed by the Air Force Research Labs. In this case, the Event Provider is a format translator. More precisely, the Event Provider used in afedSTAT reads events from the database and transforms them into IDMEF events as specified by the IDMEF Language Extension. As a consequence, it was possible to reuse all of the scenarios developed for AlertSTAT in the analysis of the AFED data without change.

2.5 WinSTAT and LinSTAT

WinSTAT and LinSTAT are two host-based systems similar to USTAT. WinSTAT uses the event logs produced by Windows NT/2000/XP. LinSTAT uses the event logs produced by the Snare Linux kernel module [12]. These two systems are an interesting example of component reuse to implement similar functionality in different environments/platforms. The Event Providers for USTAT, LinSTAT, and WinSTAT are obviously different. However, some of the entities used in scenarios are the same, and so are some of the scenarios (e.g., a scenario that detects privileged access from unprivileged applications).

2.6 AodvSTAT and AgletSTAT

The versatility of the STAT Framework was tested in developing very different systems. A well-defined framework extension process is not only a good way to develop a family of systems; it is also useful to produce proof-of-concept prototypes in a short amount of time. This is the case for two systems, called AodvSTAT and AgletSTAT. AodvSTAT is an IDS that interprets AODV [24] protocol messages and detects attacks against ad hoc wireless networks. AgletSTAT is an IDS that analyzes the events generated by a mobile agent system, called *Aglets* [18], and detects attacks that exploit mobile agents.

2.7 Family Issues

Developing a family of systems using an object-oriented framework has a number of advantages. First, the members of the program family benefit from the characteristics of the common code base. For example, all of the STAT applications use extended versions of STATL, and, therefore, they all have a well-defined language to describe attack scenarios. Second, it is possible to embed command and control functionality within the shared part of the framework. As a consequence a single configuration and control paradigm can be used to control a number of different systems. This is an issue that is particularly relevant for the domain of intrusion detection, and it is explained further in Section 1.3. Third, by factoring-out the commonalities between members of the family, it is possible to reuse substantial portions of the code. Finally, the use of a framework-based approach reduces the development time and allows one to build complete intrusion detection systems in a small amount of time.

3. MetaSTAT

MetaSTAT is an infrastructure that enables dynamic reconfiguration and management of the deployed STAT-based IDSs. MetaSTAT is responsible for the following tasks:

- **Route control messages to STAT sensors and other MetaSTAT instances.** MetaSTAT components can remotely control STAT-based sensors [2] through control messages. These messages may also cross the boundary of a web of sensors if the infrastructure security policy allows one to do so.

- **Collect, store, and route the alerts produced by the managed sensors.** Alerts about ongoing attacks are collected in a database associated with a single web of sensors. In addition, MetaSTAT components and STAT-based sensors can subscribe for specific alerts. Alerts matching a subscription are routed to the appropriate MetaSTAT endpoints. Alerts can also be sent across webs of sensors, to support high-level correlation and alert fusion.

- **Maintain a database of available modules and relative dependencies.** Every STAT component is stored in a *Module Database* together with meta-information, such as the dependencies with respect to other modules and the operational environment where the module can be deployed.

- **Manage sensor reconfiguration.** MetaSTAT uses the Module Database and the information regarding the components that are active or installed

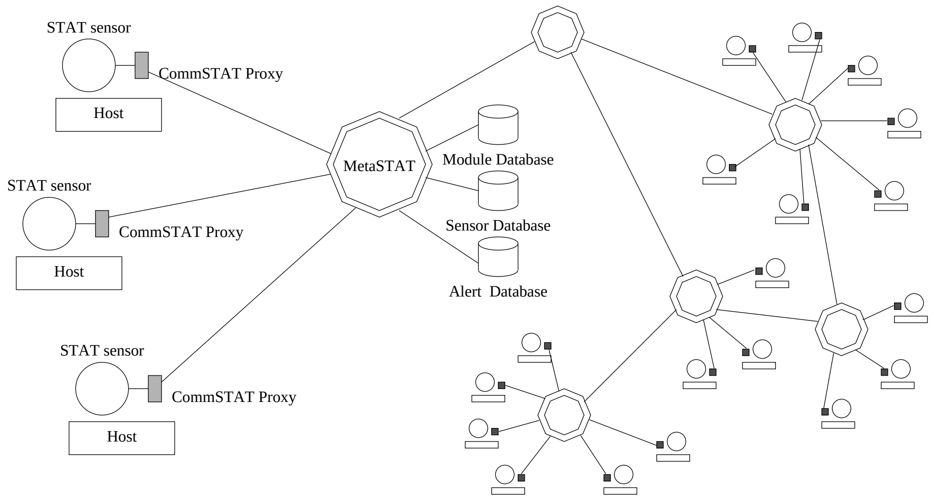


Figure 1.5. Architecture of a web of sensors.

at each STAT-based sensor as the basis for controlling the sensors and planning reconfi gurations of the surveillance infrastructure.

3.1 Control Infrastructure

The high-level view of the architecture of the STAT-based web of sensors is given in Figure 1.5. MetaSTAT uses a communication infrastructure, called *CommSTAT*, to route messages and alerts between the different MetaSTAT endpoints in a secure way. CommSTAT messages are based on the IDMEF format, which defi nes two events, namely `Heartbeat` and `Alert`. This original set of events has been extended to include STAT-related control messages that are used to control and update the confi guration of STAT sensors. For example, messages to ship a Scenario Plugin to a remote sensor and have it loaded into the Core have been added, as well as messages to manage Language Extensions and other modules.

MetaSTAT-enabled sensors are connected to a *MetaSTAT proxy*, which serves as an interface between the MetaSTAT infrastructure and the sensors. The proxy application performs preprocessing of messages, authentication of the MetaSTAT endpoints, and integration of third-party applications into the MetaSTAT infrastructure. When receiving messages from a *MetaSTAT controller*, the proxy passes the control message on to the connected sensors, which execute the control command. Three different classes of control messages are supported:

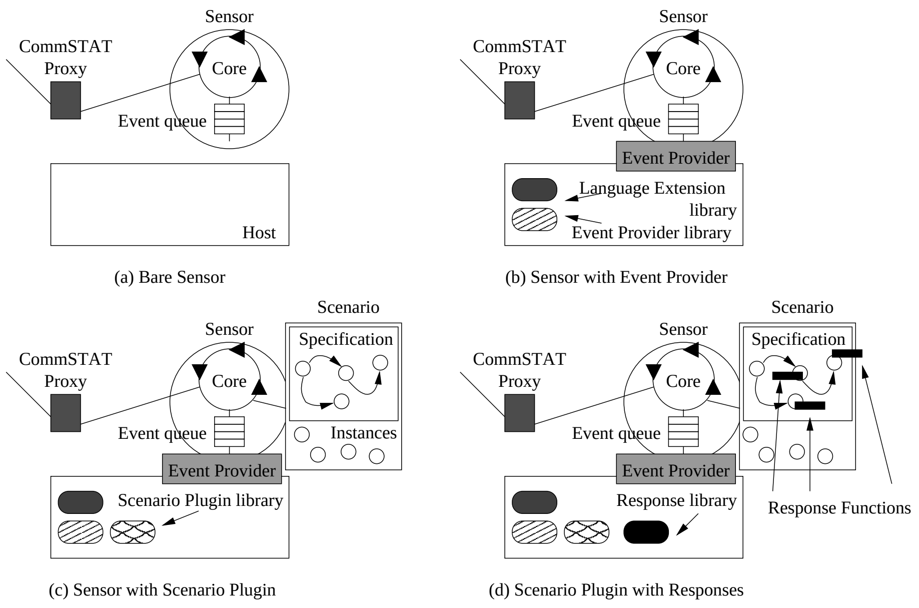


Figure 1.6. Evolution of a STAT-based sensor.

- **Install/uninstall messages.** An install message copies a software component to the local file system of a sensor, and an uninstall message removes the component from the file system.
- **Load/unload messages.** A load message instructs a sensor to load a STAT module into the address space of the sensor. After the processing of the message is completed the loaded module is available for the sensor to use. An unload message removes an unused module from the address space of a sensor.
- **Activate/deactivate messages.** An activate message starts an instance of a previously loaded STAT module. The activate message supports the passing of parameters to a STAT module. It is common to activate several instances of the same module with different parameters. A deactivate message stops the execution of an instance.

The configuration of a STAT sensor can be changed at run-time through control directives sent by the MetaSTAT controller to the proxy component responsible for the sensor. A set of initial modules can be (and usually is) defined at startup time to determine the initial configuration of a sensor. In the following paragraphs, an incremental configuration of a STAT-based sensor will be described to better illustrate the role of each sensor module, provide a hint of the high degree of configurability of sensors, and describe the dependencies between the different modules.

When a sensor is started with no modules, it contains only an instance of the STAT Core waiting for events to be processed. The Core is connected to a proxy, which, in turn, is connected to a MetaSTAT controller instance. This initial "bare" configuration, which is presented in Figure 1.6 (a), does not provide any intrusion detection functionality.

The first step is to provide a source of events. To do this, an Event Provider module must be loaded into the sensor and then activated. This is done through MetaSTAT by requesting the shipping of the Event Provider shared library to the sensor, and then requesting its loading and activation. An Event Provider relies on the event definitions contained in one or more Language Extension modules. If these are not available at the sensor's host, then they have to be installed and loaded. Once both the Event Provider and the Language Extensions are loaded into the sensor, the Event Provider is activated. As a consequence, a dedicated thread of execution is started to execute the Event Provider. The provider collects events from an external source, filters out those events that are not of interest, transforms the remaining events into event objects (as defined by a Language Extension), and then inserts the event objects into the Core input queue. The Core, in turn, consumes the events and checks if there are any STAT scenarios interested in the specific event types. At this point, there are no scenarios, and, therefore, there are no events of interest to be processed. This configuration is described in Figure 1.6 (b).

To start doing something useful, it is necessary to load one or more Scenario Plugins into the Core and activate them. To do this, first a Scenario Plugin module, in the form of a shared library, is installed on the sensor's host. A scenario may need the types and events of one or more Language Extension modules. If these are not already available at the destination host then they are installed and loaded. Once all the necessary components are available, the scenario is loaded into the Core and activated, specifying a set of initial parameters. When a Scenario Plugin is activated, an initial scenario prototype is created. The scenario prototype contains the data structures representing the scenario's definition in terms of states and transitions, a global environment, and a set of activation parameters. The prototype creates a first instance of the scenario. This instance is in the initial state of the corresponding attack scenario. The Core analyzes the scenario definition and subscribes the instance for the events associated with the transitions that start from the scenario's initial state. At this point the Core is ready to perform event processing, as shown in Figure 1.6 (c).

As a scenario evolves from state to state, it may produce some output. A typical case is the generation of an alert when a scenario completes. Another example is the creation of a *synthetic event*, which is a STAT event that is generated by a scenario plugin and inserted in the Core event queue. The event is processed like any other event and may be used to perform forward chaining of scenarios.

Apart from logging (the default action when a scenario completes) and the production of synthetic events (that are specifi ed internally to the scenario definition), other types of responses can be associated with scenario states using *response modules*. Response modules are collections of functions that can be used to perform any type of response (e.g., page the administrator, reconfi gure a fi rewall, or shutdown a connection). Response modules are implemented as shared libraries. To activate a response function it is necessary to install the shared library containing the desired response functionality on the sensor's host, load the library into the Core, and then request the association of a function with a specifi c state in a scenario defi nition. This allows one to specify responses for any intermediate or fi nal state in any attack scenario. Each time the specifi ed state is reached by any of the instances of the scenario, the corresponding response is executed. Responses can be installed, loaded, activated, and removed remotely using the MetaSTAT component. Figure 1.6 (d) shows a response library and some response functions associated with particular states in the scenario defi nition.

At this point, the sensor is confi gured as a full-fledged intrusion detection system. Event providers, scenario plugins, language extensions, and response modules can be loaded and unloaded following the needs of the overall intrusion detection functionality. As described above, these reconfi gurations are subject to a number of dependencies that must be satisfi ed in order to successfully load a component into the sensor and to have the necessary inputs and outputs available for processing. These dependencies are managed by the MetaSTAT component, and they are discussed in the next section.

3.2 Sensor Reconfiguration

The flexibility and extendibility supported by the STAT-based approach is a major advantage: the confi guration of a sensor can be reshaped in real-time to deal with previously unknown attacks, changes in the site's policy, different levels of concern, etc. Fine-grained confi gurability requires careful planning of module installation and activation, and this activity can be very complex and error-prone if carried out without support. For this reason the MetaSTAT component maintains a database of modules and their associated dependencies and a database of the current sensor confi gurations. These databases provide the support for consistent modifi cations of the managed web of sensors. In the following, the term *module* is used to denote language extensions, event providers, scenario plugins, and response modules. The term *external component* is used to characterize some host facility or service that is needed by an event provider as a source of raw events or by a response function to perform some action. External components are outside the control of MetaSTAT. For example, a BSM event provider needs the actual BSM auditing system up and running to be able to access audit records and provide events to the STAT Core.

Dependencies between modules can be classified into *activation* dependencies and *functional* dependencies. Activation dependencies must be satisfied for a module to be activated and run without failure. For example, consider a scenario plugin that uses predicates defined in a language extension. The language extension must be loaded into the Core before the plugin is activated. Otherwise, the plugin activation will fail with a run-time linking error. Functional dependencies are associated with the *inputs* of a module. The functional dependencies of a module are satisfied if there exist modules and/or external components that can provide the inputs used by the module. Note that a module can successfully be activated without satisfying its functional dependencies. For example, suppose that a scenario plugin that uses BSM events has been successfully activated, but there is no BSM event provider to feed BSM events to the Core. In this case, the scenario is active but completely useless. The inputs and outputs of the different module types, and the relative dependencies are summarized in Table 1.1.

Module	Inputs	Outputs	Activation Dependencies	Functional Dependencies
Event Provider	External event stream	STAT events	Language Extension modules	External components
Scenario Plugin	STAT events, synthetic events	Synthetic events	Language Extension modules	Scenario plugins, Event providers
Response Module	Parameters from plugin	External response	Language Extension modules	External components
Language Extension	None	None	Language Extension modules	None

Table 1.1. Input and output, and dependencies of STAT sensor modules.

Information about dependencies between modules is stored in MetaSTAT's *Module Database*.

Determining the functional dependencies on other modules requires that two queries be made on the Module Database. The first query gets the inputs required by the module. The second query determines which modules are generating the inputs that were returned from the first query. The results returned from the second query identify the modules that satisfy the functional dependencies of the original module. The functional dependencies on external components are modeled explicitly by the database. In addition to dependencies, the Module Database also stores information such as version and OS/architecture compatibility information.

The Module Database is used by MetaSTAT to automatically determine the steps to be undertaken when a sensor reconfiguration is needed. Since sensors do not always start from a "bare" configuration, as shown in Figure 1.6 (a), it is usually necessary to modify an existing sensor configuration. Therefore, the MetaSTAT component maintains a second database called the *Sensor Database*, which contains the current configuration for each sensor. This database is updated at reconfiguration time by querying the current configuration of the sensor.

To be more precise, the term *configuration* is defined as follows: *A STAT sensor configuration is uniquely defined by a set of installed and activated modules and available external components.* The term *installed* is used to describe the fact that a module has been transferred to and stored on a file system accessible by the sensor and in a location known by the sensor. The term *activated* is used to describe the fact that a *module* has been dynamically loaded in a sensor as the result of a control command from MetaSTAT. The term *loaded* has the same meaning as *activated* in relation to language extension modules.

A configuration can be *valid* and/or *meaningful*. A configuration is valid if all activated modules have all their activation dependencies satisfied. A configuration is meaningful if the configuration is valid and all functional dependencies are also satisfied.

3.3 Reconfiguration Example

To better describe the operations involved in a reconfiguration and the support provided by MetaSTAT, an example will be used.

Suppose that the Intrusion Detection Administrator (IDA) noted or was notified of some suspicious FTP activity in a subnetwork inside the IDA's organization. Usually, the IDA would contact the responsible network administrator and would ask him/her to install and/or activate some monitoring software to collect input data for further analysis. The IDA might even decide to login remotely to particular hosts to perform manual analysis. Both activities are human-intensive and require considerable setup time.

MetaSTAT supports a different process in which the IDA interacts with a centralized control application (i.e., the MetaSTAT console) and expresses an interest in having the subnetwork checked for possible FTP-related abuse. This request elicits a number of actions:

1 The scenario plugins contained in the Module Database are searched for the keyword “FTP”. More precisely the IDA’s request is translated into the following SQL query:

```
SELECT module_id, name, os_platform, description
  FROM Module_Index
  WHERE (name LIKE ’%ftp%’ OR
         description LIKE ’%ftp%’)
    AND type="plugin";
```

The following information is returned:

module_id	name	os_platform	description
module_1	wu-ftpd-bovf	Linux X86	BOVF attack against ftpd
module_2	ftpd-quote-abuse	Linux X86	QUOTE command abuse
...	...	...	...
module_9	ftpd-protocol-verify	Linux X86	FTP protocol verifier

The IDA selects the `wu-ftp-bovf` and `ftpd-quote-abuse` scenario plugins for installation.

2 The Module Database is examined for possible activation dependencies. The `wu-ftp-bovf` activation dependencies are determined by the following query:

```
SELECT dep_module_id FROM Activation_Dependency
  WHERE module_id="module_1";
```

The query results (not shown here) indicate that the scenario plugin requires the `ftp` language extension. This is because events and predicates defined in the `ftp` extension are used in states and transitions of the `wu-ftp-bovf` scenario. A similar query is performed for the `ftpd-quote-abuse` scenario plugin. The query results indicate that the `syslog` language extension is required by the plugin.

3 The Module Database is then searched for possible functional dependencies. For example in the case of the `wu-ftp-bovf` scenario the following query is executed:

```
SELECT input_id FROM Module_Input
  WHERE module_id="module_1";
```

The query returns an entry containing the value `FTP_PROTOCOL`. This means that the `wu-ftp-bovf` scenario uses this type of event as input. Therefore, the `wu-ftp-bovf` scenario plugin has a functional dependency on a module providing events obtained by parsing the FTP protocol. A similar query indicates that the `ftpd-quote-abuse` plugin has a functional dependency on a provider of `SYSLOG` events.

4 These new requirements trigger a new search in the Module Database to find which of the available modules can be used to provide the required inputs. `SYSLOG` events are produced by three event providers: `syslog1`, `syslog2`, and `win-app-event`. The `FTP_protocol` events are produced, as synthetic events, by the `ftp-protocol-verify` scenario.

5 Both the `syslog1` and `syslog2` event providers require an external source, which is the syslog facility of a UNIX system. In particular, `syslog2` is tailored to the *syslogkd* daemon provided with Linux systems. Both event providers have an activation dependency on the `syslog` language extension. The `win-app-event` event provider is tailored to the Windows NT platform. It depends on the NT event log facility (as an external component) and relies on the NT event log language extension (`winevent`). The `ftp-protocol-verify` is a network-based scenario and, as such, requires a network event provider that produces events of type `STREAM`, which are events obtained by reassembling TCP streams. The scenario has two activation dependencies; it needs both the `tcpip` and the `ftp` language extensions. The first is needed because `STREAM` events are used in the scenario's transition assertions. The second is needed to be able to generate the `FTP_protocol` synthetic events.

6 Events of type `STREAM` are produced by an event provider called `netproc`. This event provider is based on the `tcpip` language extension, and requires, as an external component, a network driver that is able to eavesdrop traffic.

7 At this point, the dependencies between the modules have been determined (see Figure 1.7). The tool now identifies the sensors that need to be reconfigured. This operation is done by querying the Sensor Database to determine which hosts of the network under examination have active STAT-based sensors. The query identifies two suitable hosts. Host `lucas`, a Linux machine, has a bare sensor installed. Host `spielberg`,

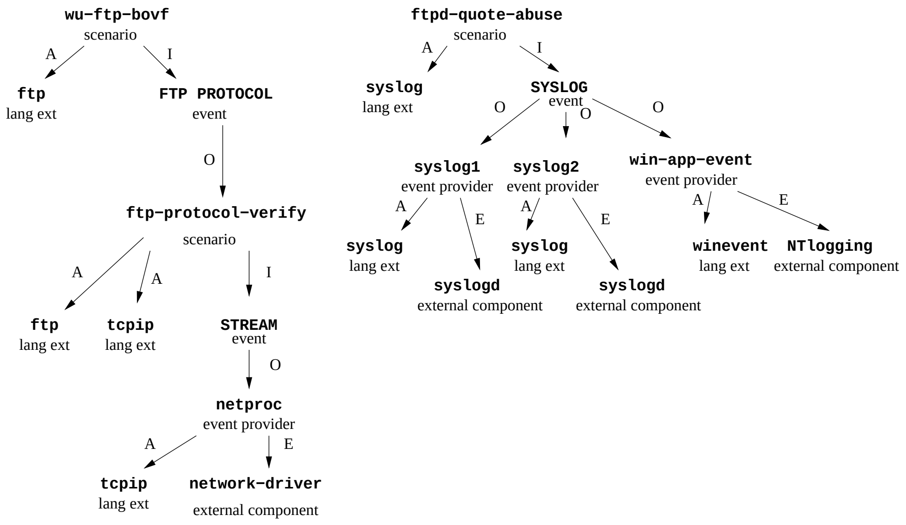


Figure 1.7. Dependency graph for scenarios `wu-ftp-bovf` and `ftpd-quote-abuse`. In the figure, arrows marked with the letter "A" are used to represent activation dependencies. Arrows marked with "I" represent the relationship between a module and the input events required. Arrows marked with an "O" represent the relationship between an event type and the module that produce that type of event as output. Arrows marked with "E" represent a dependency on an external component.

another Linux machine, runs a STAT-based sensor equipped with the `netproc` event provider, the `tcpip` language extension, and some scenario plugins. Both hosts provide the network driver and UNIX syslog external component. The tool decides (possibly with intervention from the IDA) to install the `ftpd-quote-abuse` scenario on `lucas` and the `wu-ftp-bovf` scenario on `spielberg`.

8 The `syslog` language extension is sent to `lucas`, and it is installed in the fi le system. This is done using the following CommSTAT messages:

```
<x-stat-extension-lib-install id="1">
   <extension_lib name="syslog" version="1.0.1">
      [... encoded library ...]
   </extension-lib>
</x-stat-extension-lib-install>

<x-stat-extension-lib-activate id="2">
   <extension_lib name="syslog" version="1.0.1">
   </extension-lib>
</x-stat-extension-lib-activate>
```

The `syslog2` event provider is sent, installed, and loaded in the sensor by means of similar commands. At this point syslog events are being fed to the Core of the sensor on host `lucas`. The `ftpd-quote-abuse` scenario plugin is sent to the host, installed on the file system, and eventually loaded into the Core.

9 The `ftp` language extension is sent to host `spielberg`. The `tcpip` language extension is already available, as is the `netproc` event provider. Therefore, the `ftp-protocol-verify` scenario plugin can be shipped to host `spielberg`, installed, and loaded into the Core. The scenario starts parsing `STREAM` events and producing `FTP_PROTOCOL` synthetic events. As the final step, the `wu-ftpd-bovf` scenario is shipped to host `spielberg`, installed, and loaded into the Core, where it immediately starts using the synthetic events generated by the `ftp--protocol-verify` scenario.

After the necessary reconfigurations are carried out, the IDA may decide to install specific response functions for the newly activated scenarios. A process similar to the one described above is followed. Response modules, in the form of shared libraries, may be installed on a remote host and linked into a sensor. Additional control commands may then be used to associate states in a scenario with the execution of specific functions of the response module.

4. Related Work

Object-oriented frameworks are “sets of cooperating classes that make up a reusable design for a specific class of software” [6]. Generally, frameworks are targeted for specific domains to maximize code reuse for a class of applications [15]. The STAT Framework is targeted for the development of event-based intrusion detection systems. In this context, the use of a framework differs from traditional approaches [11, 29], because all of the components that are developed as part of the framework are highly independent modules that can be composed (almost) arbitrarily through dynamic loading into the framework runtime. In addition, the framework extension process is not limited to the creation of a domain-specific intrusion detection system. The same process produces products for different domains, depending on the events, types, and predicates defined in the Language Extensions. The product of the STAT Framework is a family of intrusion detection systems.

The concept of program families was introduced by Parnas in [22] and has received considerable attention from the Software Engineering community ever since. Unfortunately, the criteria, methodologies, and lessons learned in developing software families in a number of fields have not been applied to intrusion detection. Even though in recent years the focus of intrusion detection has moved from single-domain approaches (e.g., network-based only) to multi-domain approaches (e.g., correlation of alerts from both network-level and OS-level event analysis), this change of focus has not been matched by a corresponding shift in development methodology. As a consequence, while IDS are becoming more common, their development is still characterized by an *ad hoc* approach. Notable examples are SRI's Emerald [26, 20], ISS RealSecure [13], and Prelude [31]. All of these toolsets include a number of different sensor components and high-level analysis engines. For example, Emerald has a host-based intrusion detection system, two network-based analyzers, and a correlation/aggregation component. Even though the toolset covers a number of different domains, there is no explicit mechanism in the Emerald approach that is exclusively dedicated to support the extension of the system to previously uncovered domains. The same limitation appears in both RealSecure, which is a mainstream commercial tool, and Prelude, which is an open-source project.

5. Conclusions

The STAT Framework is an approach for the development of intrusion detection systems based on the State Transition Analysis Technique. This chapter described the framework, the corresponding extension process, and the result of applying the framework to develop a family of systems.

The work reported in this chapter makes contributions in several areas. By using object-oriented frameworks and by leveraging the properties of program families it was possible to manage the complexity of implementing intrusion functionality on different platforms, environments, and domains. The framework supports effi cient development of new intrusion detection sensors because the main mechanisms and the semantics of the event processing are implemented in a domain-independent way. Therefore, the IDS developer has to implement only the domain/environment-specifi c characteristics of the new sensor. Practitioners in the fi eld of intrusion detection can certainly gain from the lessons learned. Hopefully, they will use the STAT framework or adapt a component-based software family approach for their own development.

Two areas where the reported work contributes to previous work in the component and framework communities is in leveraging the architecture to have a common confi guration and control infrastructure and in having the attack specifi cation language tightly coupled with the application development. STAT-based intrusion detection systems that operate on different event streams (e.g., OS audit records and network packets) and at different abstraction levels (e.g., detection and correlation) share a similar architecture and similar control primitives. As a consequence, a single confi guration and control infrastructure can be used to manage a large number of heterogeneous components.

Language Extension modules extend the domain-independent STATL core language to allow users to specify attack scenarios in particular application domains. The same Language Extension modules are compiled and used by the runtime core for recognizing events and types. Because it is the same Language Extension module for both, the user automatically gets an attack specification language along with his/her intrusion detection system. In addition, because the attack specification languages are an extension of the STATL core language, a user does not need to learn a new language style when setting up attack scenarios for a new intrusion detection application.

The STAT tools and the MetaSTAT infrastructure have been used in a number of evaluation efforts, such as the MIT/Lincoln Labs evaluations and the Air Force Rome Labs evaluations, in technology integration experiments, such as DARPA's Grand Challenge Problem (GCP) and the iDemo technology integration effort. In all of these very different settings, the STAT tools performed very well by detecting attacks in real-time with very limited overhead. In most cases, the STAT tools were run and compared with other tools from both the research and the commercial worlds. The positive feedback received from the organizers of these evaluation efforts provided a particularly significant comparison of the STAT toolset performance with respect to other state-of-the-art intrusion detection technologies.

The STAT Framework, the MetaSTAT infrastructure, and the STAT-based tools are open-source and publicly available at the STAT web site *http://www.-cs.ucsb.edu/∼rsg/STAT*.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0207, and by the National Security Agency's University Research Program, under agreement number MDA904-98-C-A891. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, the Defense Advanced Research Projects Agency (DARPA), the National Security Agency (NSA), or the U.S. Government.

Notes

1. An alternative would be to execute the transition codeblock before evaluating the state assertion. However, this would require backtracking to undo environment changes when the state assertion is not satisfied. Otherwise, the environment could be changed for "partially" fired transitions, which would be semantically unsatisfactory.

2. In the remainder of this chapter an instance of an intrusion detection system may be referred to as a sensor.

References

[1] *Apache 2.0 Documentation*, 2001. `http://www.apache.org/`.

[2] D. Curry and H. Debar. Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition. `draft-ietf-idwg-idmef-xml-06.txt`, December 2001.

[3] 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.

[4] 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.

[5] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[7] 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)*, pages 259–267, Scottsdale, AZ, December 1998.

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

[9] 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.

[10] 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):181–199, March 1995.

[11] Taligent Inc. Building Object-Oriented Frameworks. White Paper, 1994.

[12] Intersect Alliance. Snare: System Intrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`, August 2002.

[13] ISS. Realsecure 7.0. http://www.iss.net/, August 2002.

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

[15] R. Johnson and B. Foote. Designing Reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[16] K. Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. Master's thesis, MIT, June 1999.

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

[18] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[19] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating Intrustion Detection Systems: The 1998 DARPA Offline Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition, Volume 2*, Hilton Head, SC, January 2000.

[20] P.G. Neumann and P.A. Porras. Experience with EMERALD to Date. In *First USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, April 1999.

[21] NFR Security. *Overview of NFR Network Intrusion Detection System*, February 2001.

[22] D.L. Parnas. The Design and Development of Program Families. *IEEE Transactions on Software Engineering*, March 1976.

[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] C.E. Perkins and E.M. Royer. Ad hoc on-demand distance vector routing. In C. Perkins, editor, *Ad hoc Networking*. Addison-Wesley, 2000.

[25] 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.

[26] P.A. Porras and P.G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 1997 National Information Systems Security Conference*, October 1997.

[27] M.J. Ranum, K. Landfi eld, 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.

[28] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.

[29] G. F. Rogers. *Framework-Based Software Development in C++*. Prentice-Hall, 1997.

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

[31] Y. Vandoorselaere. Prelude, an Hybrid Open Source Intrusion Detection System. `http://www.prelude-ids.org/`, August 2002.

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

[33] G. Vigna, R.A. Kemmerer, and P. Blix. Designing a Web of Highly-Confi gurable Intrusion Detection Sensors. In W. Lee, L. Mè, and A. Wespi, editors, *Proceedings of the* 4^{th} *International Symposiun on Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 69–84, Davis, CA, October 2001. Springer-Verlag.

[34] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.