Designing a Web of Highly-Configurable Intrusion Detection Sensors

Giovanni Vigna, Richard A. Kemmerer, and Per Blix

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

Abstract. Intrusion detection relies on the information provided by a number of *sensors* deployed throughout the monitored network infrastructure. Sensors provide information at different abstraction levels and with different semantics. In addition, sensors range from lightweight probes and simple log parsers to complex software artifacts that perform sophisticated analysis. Managing a configuration of heterogeneous sensors can be a very time-consuming task. Management tasks include planning, deployment, initial configuration, and run-time modifications. This paper describes a new approach that leverages off the STAT model to support a highly configurable sensing infrastructure. The approach relies on a common sensor model, an explicit representation of sensor component characteristics and dependencies, and a shared communication and control infrastructure. The model allows an Intrusion Detection Administrator to express high-level configuration requirements that are mapped automatically to a detailed deployment and/or reconfiguration plan. This approach supports automation of the administrator tasks and better assurance of the effectiveness and consistency of the deployed sensing infrastructure.
Keywords: *Security, Software Engineering, Intrusion Detection, STAT.*

1 Introduction

Any monitoring and surveillance functionality builds on the analysis performed by *surveillance sensors*. The intrusion detection community has developed a number of different systems that perform intrusion detection in particular domains (e.g., hosts or networks) and in specific environments (e.g., Windows NT or Solaris).

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 specific case of signature-based intrusion detection systems [1–4] the sensors are equipped with a number of signatures that are matched against a stream of incoming events. Most systems (e.g., [1]) are initialized with a set of signatures at startup time. Updating the signature set requires stopping the sensor, updating the signature set, and then restarting

execution. Some of these tools 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. In addition, the *ad hoc* nature of existing tools does not allow one to dynamically configure a running sensor so that a new event stream can be used as input for the security analysis.

Another limit of existing tools is the relatively static configuration of responses. First of all, as for signatures, normally it is possible to choose only from a specific subset of possible responses. In addition, to our knowledge, no system allows for associating 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, the configuration of existing tools is mainly 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 configurations. The challenge is to determine if the current configuration of one or more sensors is valid or if a reconfiguration is meaningful.

In this paper, we describe a novel approach to distributed intrusion detection. The idea is that a protected network is instrumented with a "web of sensors" composed of distributed components integrated by means of a local 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. The outputs of the sensors, in the form of *alerts*, are collected by a number of "meta-sensor" components. Each meta-sensor is responsible for a subset of the deployed sensors, and may coordinate its activities with other meta-sensors. The meta-sensors are responsible for storing the alerts, for routing alerts to other sensors and meta-sensors (e.g., to perform correlation to identify composite attack scenarios), and for exerting control over the managed sensors.

Control is the most challenging (and most overlooked) functionality of distributed surveillance. Most existing approaches simply aggregate the outputs of distributed sensors and focus mainly on the intuitive presentation of alerts to the network security officer. This is just not enough. There is a need for fine-grained control of the deployed sensors in terms of scenarios to be detected, tailoring of the sensors with respect to the protected network, and dynamic control over the types of response. These are requirements that can be satisfied only if the surveillance sensors are *highly configurable* and configuration can be performed dynamically, without stopping and restarting sensors when a reconfiguration is needed.

We have designed a suite of highly configurable surveillance sensors and a command and control meta-sensor that allows the network security officer to exert a very fine-grained control over the deployed surveillance infrastructure. Meta-sensors can be organized hierarchically to achieve scalability and can be replicated to support fault-tolerance. This web of sensors is built around the

State Transition Analysis Technique (STAT) framework developed by the Reliable Software Group at UCSB. The STAT framework provides a platform for the development of highly configurable probes in different domains and environments. The STAT approach is centered around five key concepts: the STAT technique, the STATL language, the STAT Core, the CommSTAT communication infrastructure, and the MetaSTAT control system.

The approach 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, the approach models explicitly the dependencies among the modules composing a sensor so that it is possible to identify automatically the steps that are necessary to perform a reconfiguration of the deployed sensing infrastructure. In addition, the possibility of retrieving current configurations from remote sensors allows one to determine if a reconfiguration is valid or meaningful.

The remainder of the paper is structured as follows. Section 2 presents the fundamental elements of the STAT approach. Section 3 describes the structure of STAT-based sensors. Section 4 discusses the dependencies between modules and the concept of a valid and meaningful configuration. Section 5 describes how dependencies are used during the reconfiguration process. Section 6 draws some conclusions and outlines future work.

2 The STAT Framework

The STAT framework is the result of the evolution of the original STAT technique and its application to UNIX systems [5–7] into a general framework for the development of STAT-based intrusion detection sensors [8].

The STAT Technique. STAT is a technique for representing high-level descriptions of computer attacks. Attack scenarios are abstracted into *states*, which describe the security status of a system, and *transitions*, which model the evolution between states. By abstracting from the details of particular exploits and by modeling only the key events involved in an attack scenario STAT is able to model entire classes of attacks with a single scenario, overcoming some of the limitations of plain signature-based misuse detection systems [9].

The STATL Language. STATL is an extendible language [10] that is used to represent STAT attack scenarios. The language defines the domain-independent features of the STAT technique. The STATL language can be extended to express the characteristics of a particular domain and environment. The extension process includes the definition of the set of *events* that are specific to the particular domain or environment being addressed and the definition of new *predicates* on those events. For example, to extend STATL to deal with events produced by the Apache Web browser one would define one or more events that represent entries in the application logs. In this case an event would have the fields `host`,

`ident`, `authuser`, `date`, `request`, `status`, and `bytes` as defined by Apache's Common Log Format (CLF) [11]. After having defined new events it may be necessary to specify specific predicates on those events. For example, the predicate `isCGIrequest()` would return true if an event is a request for a CGI script. Event and predicate definitions are grouped in a *language extension*. Once the event set and associated predicates for a language extension are defined, it is possible to use them in a STATL scenario description by including them with the STATL `use` keyword. A number of extensions for TCP/IP networks, Sun BSM audit records [12], and Windows NT event logs have been developed.

STATL scenarios are matched against a stream of events by the STAT core (described below). In order to have a scenario processed by the STAT core it is necessary to compile it into a *scenario plugin*, which is a shared library (e.g., a "`.so`" library in UNIX or a DLL library in Windows). In addition, each language extension used by the scenario must be compiled into an *extension module*, which is a shared library too. Both STATL scenarios and language extension are translated into C++ code and compiled into libraries by the STAT development tools.

The STAT Core. The STAT core represents the runtime of the STATL language. The STAT core implements the domain-independent characteristics of STATL, such as the concepts of state, transition, timer, matching of events, etc. At runtime the STAT core performs the actual intrusion detection analysis process by matching an incoming stream of events against a number of scenario plugins. A running instance of the STAT core is dynamically extended to build a STAT-based sensor, as described in Section 3.

The CommSTAT communication infrastructure. STAT-based sensors are connected by a communication infrastructure that allows the sensors to exchange alert messages and control directives in a secure way. CommSTAT messages follow the standard Intrusion Detection Message Exchange Format (IDMEF) [13]. The original IDMEF definition includes the two events `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 configuration 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 (`x-stat-scenario-activate`), as well as messages to manage language extensions and other modules (the message names are all prefixed with `x-stat-`, following the extension guidelines of the IDMEF format). Participation in the CommSTAT communication infrastructure is mediated by a *CommSTAT proxy* that performs preprocessing of messages and control directives and that supports the integration of third-party tools that are not based on the STAT framework.

The MetaSTAT control infrastructure. The CommSTAT communication infrastructure is used by the MetaSTAT component to exert control over a set of sensors. The MetaSTAT component is responsible for the following tasks:

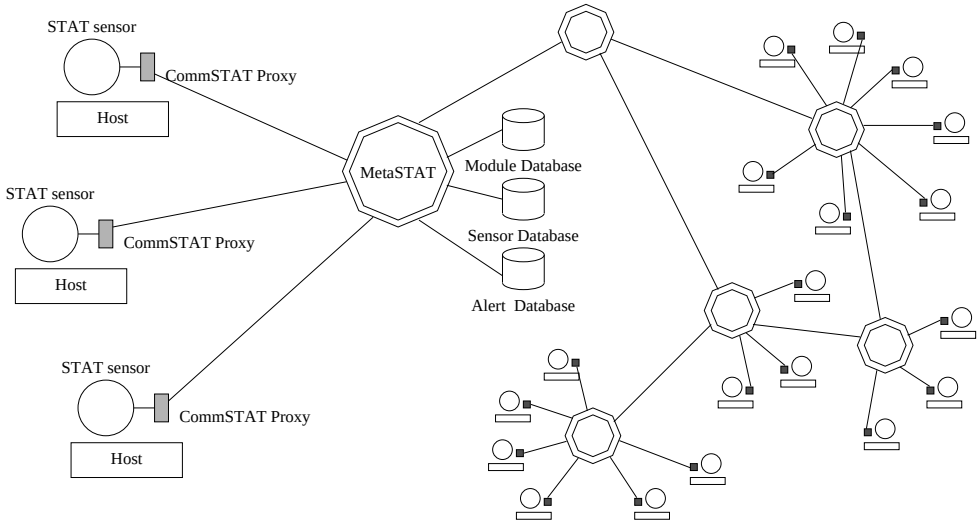

Fig. 1. Architecture of a web of sensors.

- **Collect and store the alerts produced by the managed sensors.** IDMEF alerts are stored in a MySQL relational database. A schema to efficiently store and retrieve IDMEF alerts has been developed, and a GUI for the querying and display of stored alerts has been implemented.
- **Route alerts to STAT sensors and other MetaSTAT instances.** MetaSTAT components and STAT-based sensors can subscribe for specific alerts. Alerts matching a subscription are routed through the appropriate CommSTAT communication channels.
- **Maintain a database of available modules and relative dependencies.** Each MetaSTAT component is associated with a *Module Database* of compiled scenario plugins, language extension modules, and other modules that will be discussed later. For each module, the database stores the dependencies with respect to both other modules and the operational environment where the module may need to be deployed. These dependencies are a novel aspect of the STAT approach and are described in more detail in Section 4.
- **Maintain a database of current sensor configurations.** MetaSTAT manages a *Sensor Database* containing the current components that are active or installed at each STAT-based sensor. This "privileged" view of the deployed web of sensors is the basis for controlling the sensors and planning reconfigurations of the surveillance infrastructure. The structure of the database is described in detail in Section 4.

The high-level view of the architecture of the STAT-based web of sensor is given in Figure 1. The following sections discuss the structure of a single STAT-based sensor and how its reconfiguration is performed through a MetaSTAT component.

3 STAT Sensors

STAT sensors are intrusion detection systems that perform localized security analysis of one or more event streams (operating system audit records, network traffic, application logs, system calls, etc.).

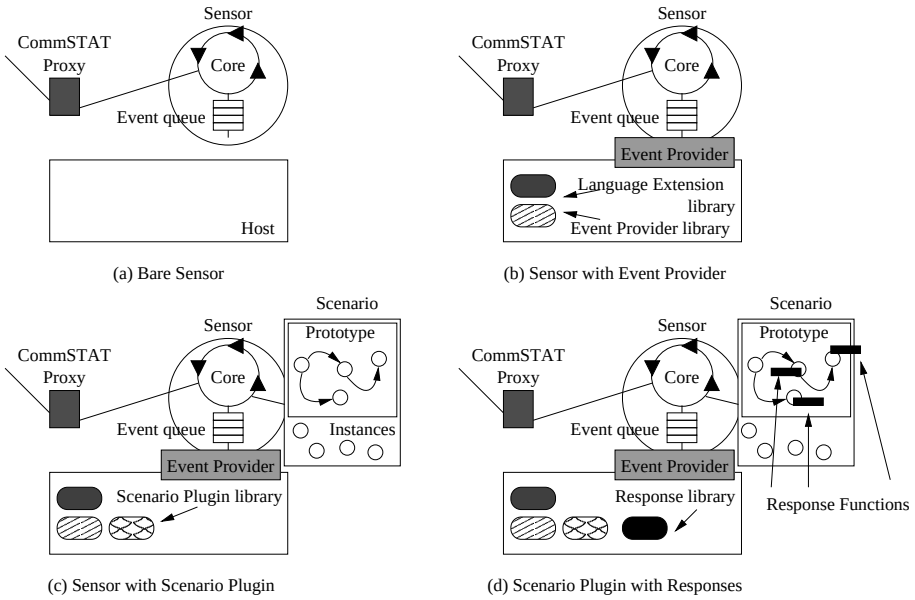

Fig. 2. Evolution of a STAT-based sensor.

The architecture of a STAT-based sensor is centered around the STAT core (see Figure 2). The STAT core is extended with a number of modules that, together, determine the sensor's intrusion detection capabilities and behavior. The configuration of a STAT sensor can be changed at run-time through control directives sent by the MetaSTAT 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, 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 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 CommSTAT proxy, which, in turn, is connected to a MetaSTAT instance. This initial "bare" configuration, which is presented in Figure 2 (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. An event provider collects events from the external environment (e.g., by parsing the Apache server logs, or by obtaining packets from the network driver), creates events as defined in one or more STAT language extensions (e.g., the Apache language extension), encapsulates these events into generic STAT events, and inserts these events into the input queue of the STAT core. Event providers can be dynamically added to and removed from a STAT core, and more than one event provider can be active at one time. For example, both an event provider for Apache events and a Solaris BSM audit record provider may feed their event streams to the same core. An event provider is implemented as a shared library. The activation of an event provider is done through MetaSTAT by requesting the shipping of the event provider shared library to the sensor and then requesting its 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, these have to be shipped there as well. Once both the event provider and the language extensions are available at the remote host, 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 the external source, filters out those that are not of interest, transforms the remaining events into event objects (as defined by the language extension), encapsulates them into generic STAT events, and then inserts them 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, the core is empty, and therefore no actual processing is carried out. This configuration is described in Figure 2 (b).

To start doing something useful, it is necessary to load one or more scenario plugins into the core. To do this, first a scenario plugin module, in the form of a shared library, is transferred to the sensor's host. A plugin may need the functions of one or more language extension modules. If these are not already available at the destination host then they are shipped there as well. Once all the necessary libraries have been transferred to the sensor's host, the plugin is loaded into the core, specifying a set of initial parameters. When a plugin is loaded into the core an initial *prototype* for the scenario 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. The events obtained by the provider are matched against the subscriptions of the initial instance. If an event matches a subscription, then the corresponding transition assertion is evaluated. If the assertion is satisfied then the destination state assertion is evaluated. If this assertion is also satisfied then the transition is fired. As a consequence of transition firing the instance may change state or a new in-

stance may be created. Each scenario instance represents an attack in progress. The details of scenario processing are described elsewhere [8]. This situation is presented in Figure 2 (c), where a scenario plugin has been loaded and there are currently four active instances of the scenario.

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*. A synthetic event 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 specified 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, reconfigure a firewall, or shutdown a connection). Response modules are implemented as shared libraries. To activate a response function it is necessary to transfer the shared library containing the desired response functionality to the sensor's host, load the library into the core, and then request the association of a function with a specific state in a scenario definition. This allows one to specify responses for any intermediate state in an attack scenario. Each time the specified state is reached by any of the instances of the scenario, the corresponding response is executed. Responses can be shipped, loaded, activated, and removed remotely through the MetaSTAT component. Figure 2 (d) shows a response library and some response functions associated with particular states in the scenario definition.

At this point, the sensor is configured 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 reconfigurations are subject to a number of dependencies that must be satisfied 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.

4 Module Dependencies and Sensor Configurations

The flexibility and extendibility supported by the STAT-based approach is a major advantage: the configuration 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 configurability requires careful planning of module installation and activation. 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 configurations. These databases provide the support for consistent modifications of the managed web of sensors. In the following, the term

module is used to denote event providers, scenario plugins, response modules, and language extensions. 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. These 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 the core with BSM events. 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.

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

Table 1. Input and output, and dependencies of STAT sensor modules.

Information about dependencies between modules is stored in MetaSTAT's *Module Database*. The schema of the Module Database is shown in Figure 3.

The functional dependencies for a module are partly modeled implicitly by matching the inputs required by the module with the outputs provided by some other module. 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 from the `Module Input` table. The second

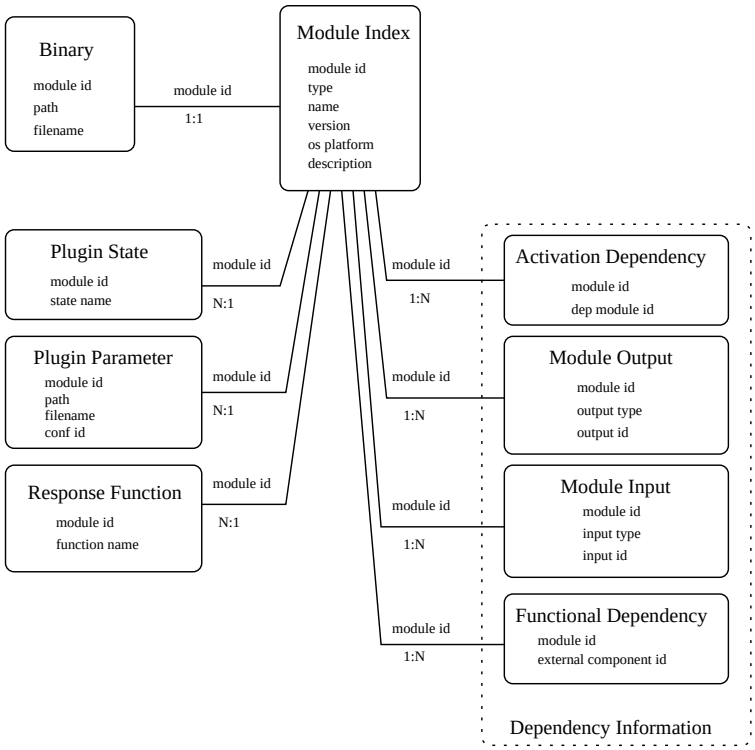

Fig. 3. Schema for the Module Database.

query examines the `Module Output` table to determine 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 `Functional Dependency` table. In addition to dependencies, the Module Database also stores information such as version, OS/architecture compatibility information, etc.

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 2 (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. A visualization of the Sensor Database schema is given in Figure 4.

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

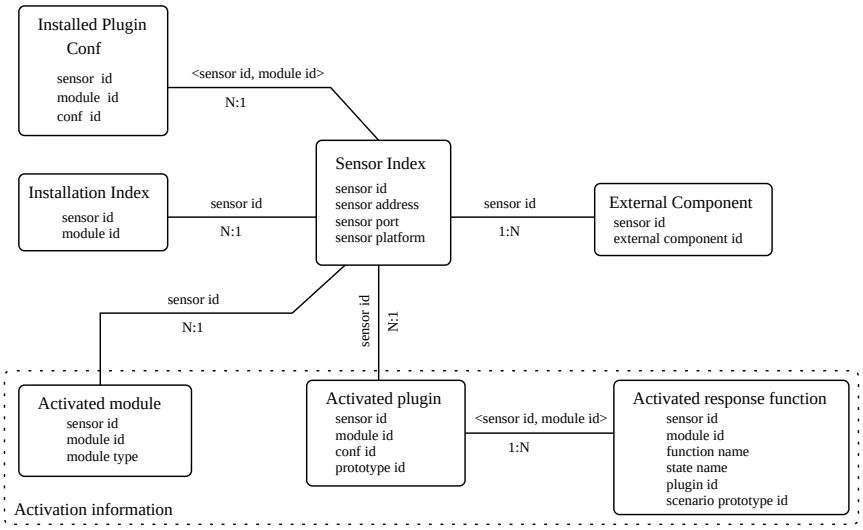

Fig. 4. Schema for the Sensor Database.

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.

5 Automated Support for Sensor Reconfigurations

MetaSTAT uses the databases described in the previous section to assist the Intrusion Detection Administrator (IDA) in reconfiguring a web of sensors. To better describe the operations involved in a reconfiguration and the support provided by MetaSTAT, an example will be used.

Suppose that the IDA noted or was notified of some suspicious FTP activity in a subnetwork inside his[1] organization. Usually, the IDA would contact the responsible network administrator and would ask him[2] 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 a considerable setup time.

[1] By "his" we mean "his or her".
[2] By "him" we mean "her".

MetaSTAT supports a different process in which the IDA interacts with a centralized control application (i.e., the MetaSTAT console) and expresses his 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 indicates 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 5). 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 exam have active STAT-based sensors. The query identifies two suitable hosts. Host `lucas`, a Linux machine, has a bare sensor installed. Host `spielberg`, 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 help 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 file 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>
```

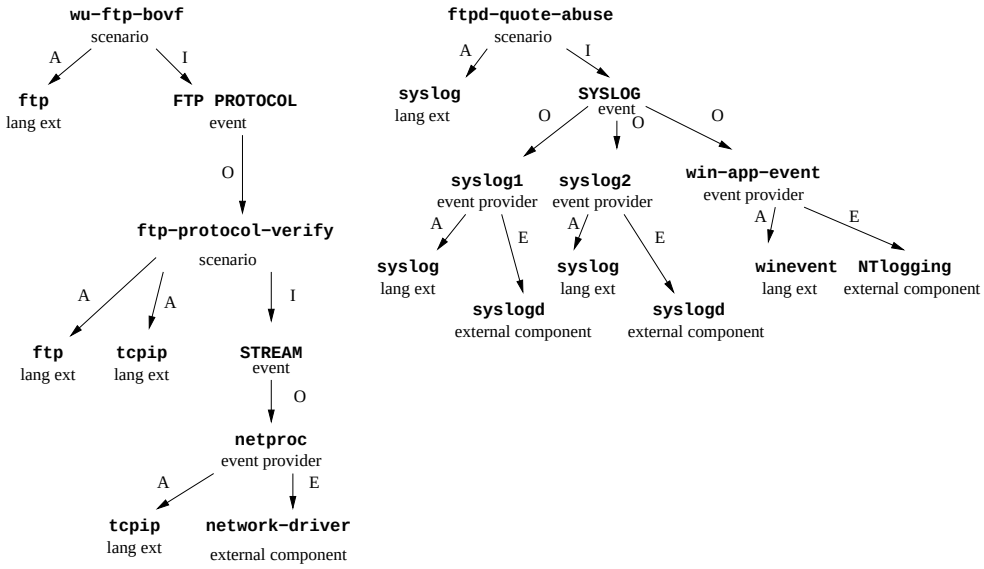

Fig. 5. Dependency graph for scenarios `wu-ftp-bovf` and `ftpd-quote-abuse`. In the figure, arrows marked with the letter "A" are used to represent activation dependencies. Arrows marked with "I" represent the relationship between a module and the input events required. Arrows marked with an "O" represent 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.

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 shipped to 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.

6 Conclusions and future work

Many research and commercial intrusion detection systems implement their intrusion detection functionality using a distributed set of sensors. The advantages of this approach are obvious, but these systems suffer from a number of limitations mainly related to their configurability.

For example, it is not possible to add new event sources to existing sensors; it is difficult, if not impossible, to create, ship, and load new signatures at run-time; and responses are usually predefined or chosen from a predefined set. We have implemented a set of components and a control infrastructure that overcome these limits. The STAT-based framework has been leveraged to realize a highly-configurable "web of sensors" controlled by a meta-sensor component, called MetaSTAT. The flexibility of the framework allows the Intrusion Detection Administrator to perform complex reconfiguration tasks. In addition, by explicitly modeling the dependencies between modules it is possible to automatically generate a valid deployment plan from high-level specifications.

The "web of sensors" is based on the STAT approach but it has been designed to be open. Third party IDS modules can easily be integrated through CommSTAT proxies. Integration of external components is limited to the exchange of alerts if primitives for the dynamic configuration of sensors are not available.

The STAT framework and the core component have been designed and implemented. The STAT framework has been used to build a number of IDSs, including two systems for host-based intrusion detection in UNIX and Windows NT environments, called USTAT and WinSTAT, respectively [5–7], a network-based intrusion detection system called NetSTAT [14, 15], and a distributed event analyzer called NSTAT [16]. Two of the systems, namely USTAT and NetSTAT, have been used in four different DARPA-sponsored evaluations [17, 18]. The CommSTAT communication infrastructure has been completed and distributed to the intrusion detection community through the IETF idwg mailing list. A first prototype of the MetaSTAT component that collects alerts from multiple sensors concurrently, stores them in a MySQL alert database and provides the IDA with a graphical viewer has been developed. In addition, database schemas for the Module Database and the Sensor Database have been implemented. Most of the control primitives have been defined and partially implemented. The MetaSTAT component is also lacking the alert routing functionalities. These will be the focus of future work.

Acknowledgments

Thanks to Steve Eckmann for providing helpful comments on this paper.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF,

under agreement number F30602-97-1-0207, by the National Security Agency's University Research Program, under agreement number MDA904-98-C-A891, and by the Army Research Office, under agreement DAAD19-01-1-0484. 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 author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, the National Security Agency, the Army Research Office, or the U.S. Government.

References

1. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Proceedings of the USENIX LISA '99 Conference. (1999)
2. Neumann, P., Porras, P.: Experience with EMERALD to Date. In: First USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California (1999) 73–80
3. NFR Security: Overview of NFR Network Intrusion Detection System. (2001)
4. Internet Security Systems: Introduction to RealSecure Version 3.0. (1999)
5. Ilgun, K.: USTAT: A Real-time Intrusion Detection System for UNIX. Master's thesis, Computer Science Department, University of California, Santa Barbara (1992)
6. Ilgun, K.: USTAT: A Real-time Intrusion Detection System for UNIX. In: Proceedings of the IEEE Symposium on Research on Security and Privacy, Oakland, CA (1993)
7. Porras, P.: STAT – A State Transition Analysis Tool for Intrusion Detection. Master's thesis, Computer Science Department, University of California, Santa Barbara (1992)
8. Vigna, G., Eckmann, S., Kemmerer, R.: The STAT Tool Suite. In: Proceedings of DISCEX 2000, Hilton Head, South Carolina, IEEE Computer Society Press (2000)
9. Ilgun, K., Kemmerer, R., Porras, P.: State Transition Analysis: A Rule-Based Intrusion Detection System. IEEE Transactions on Software Engineering **21** (1995)
10. Eckmann, S., Vigna, G., Kemmerer, R.: STATL: An Attack Language for State-based Intrusion Detection. In: Proceedings of the ACM Workshop on Intrusion Detection Systems, Athens, Greece (2000)
11. : Apache 2.0 Documentation. (2001) *http://www.apache.org/*.
12. Sun Microsystems, Inc.: Installing, Administering, and Using the Basic Security Module, 2550 Garcia Ave., Mountain View, CA 94043. (1991)
13. Curry, D., Debar, H.: Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition. `draft-ietf-idwg-idmef-xml-03.txt` (2001)
14. Vigna, G., Kemmerer, R.: NetSTAT: A Network-based Intrusion Detection Approach. In: Proceedings of the 14^{th} Annual Computer Security Application Conference, Scottsdale, Arizona (1998)
15. Vigna, G., Kemmerer, R.: NetSTAT: A Network-based Intrusion Detection System. Journal of Computer Security **7** (1999) 37–71

16. Kemmerer, R.: NSTAT: A Model-based Real-time Network Intrusion Detection System. Technical Report TRCS-97-18, Department of Computer Science, UC Santa Barbara (1997)
17. Durst, R., Champion, T., Witten, B., Miller, E., Spagnuolo, L.: Addendum to "Testing and Evaluating Computer Intrusion Detection Systems". CACM **42** (1999) 15
18. Durst, R., Champion, T., Witten, B., Miller, E., Spagnuolo, L.: Testing and Evaluating Computer Intrusion Detection Systems. CACM **42** (1999) 53–61