Designing and Implementing a Family of Intrusion Detection Systems

Giovanni Vigna Fredrik Valeur Richard A. Kemmerer

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

ABSTRACT

Intrusion detection systems are distributed applications that analyze the events in a networked system to identify malicious behavior. The analysis is performed using a number of attack models (or signatures) that are matched against a specific event stream. Intrusion detection systems may operate in heterogeneous environments, analyzing different types of event streams. Currently, intrusion detection systems and the corresponding attack modeling languages are developed following an *ad hoc* approach to match the characteristics of specific target environments. As the number of systems that have to be protected increases, this approach results in increased development effort. To overcome this limitation, we developed a framework, called STAT, that supports the development of new intrusion detection functionality in a modular fashion. The STAT framework can be extended following a well-defined process to implement intrusion detection systems tailored to specific environments, platforms, and event streams. The STAT framework is novel in the fact that the extension process also includes the extension of the attack modeling language. The resulting intrusion detection systems represent a software family whose members share common attack modeling features and the ability to reconfigure their behavior dynamically.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Security, Design

Keywords

Security, Intrusion Detection, Object-Oriented Frameworks, Program Families

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...$5.00.

1. INTRODUCTION

Intrusion detection systems (IDSs) analyze information about the activities performed in a computer system or network, 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.

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 specific case of signature-based intrusion detection systems [25, 18, 19, 11], 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-specific language (e.g., N-code [24], which is the language used by the Network Flight Recorder intrusion detection system). 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 network are not only heterogeneous, but also dynamic. Therefore, intrusion detection systems need to support mechanisms to dynamically change their configuration as the security state of the protected system evolves. Most existing intrusion detection systems (e.g., [25]) 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. In addition, the *ad hoc* nature of existing IDSs 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 limitation of existing IDSs is the relatively static

configuration of responses. Normally it is possible to choose only from a specific 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, the configuration of existing IDSs is mostly performed manually and at a very low level. This task is particularly error-prone, especially if the intrusion detection systems are deployed across a very heterogeneous environment and with very different configurations.

This paper describes a framework for the development of intrusion detection systems, called STAT, that overcomes these limitations. The STAT framework includes a domain-independent attack modeling language and a domain-independent event processing analysis engine. The framework can be extended in a well-defined way to match new domains, new event sources, and new responses. The resulting set of applications is a software family whose members share a number of features, including dynamic reconfigurability and a fine-grained control over a wide range of characteristics [30]. The main advantage of this approach is the limited development effort and the increased reuse that result from using an object-oriented framework and a component-based approach.

STAT is both unique and novel. First, STAT is the only known framework-based approach to the development of intrusion detection systems. Second, even though the use of frameworks to develop families of systems is a well-known approach, the STAT framework is novel in the fact that the framework extension process includes, as a by-product, the generation of an attack modeling language closely tailored to the target environment. This paper focuses primarily on the STAT framework. A detailed analysis of the STATL language and intrusion detection language issues can be found in [5].

This paper is structured as follows. Section 2 introduces the STAT framework. Section 3 presents a family of intrusion detection systems developed using the framework. Section 4 describes a control and communication infrastructure shared by all the IDSs in the family. Section 5 discusses the advantages of the approach in terms of reuse. Section 6 presents relevant related work. Finally, Section 7 draws conclusions and outlines future work.

2. THE STAT FRAMEWORK

The *State Transition Analysis Technique* [9] is a methodology to describe computer penetrations as *attack scenarios*. Attack scenarios are 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 state and transitions allows for an intuitive graphic representation by means of *state transition diagrams* (For an example see Figure 1.).

In the early 1990s, the State Transition Analysis Technique was applied to host-based intrusion detection, and a system, called USTAT [7, 8, 22], 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, such as Sun Microsystems' Solaris equipped with the Basic Security Module (BSM) [27]. 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 [29]. 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 the second half of 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 [17] 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 [15]. 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 very 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*.

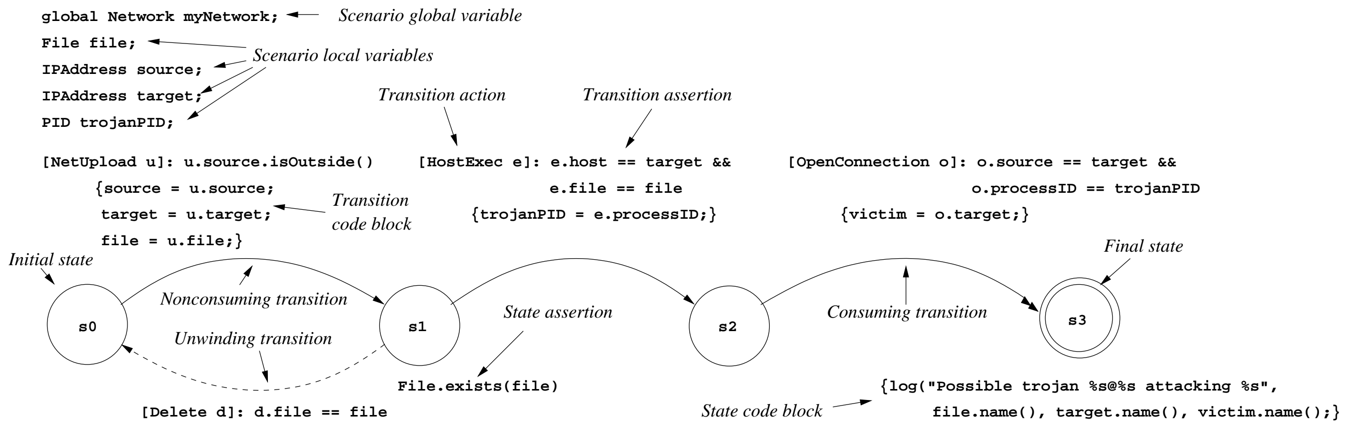

Figure 1: A sample state transition diagram of an attack scenario. The attack is a very simplified version of a Trojan horse installation attack. The first transition is fired when the upload of a file from a host outside the local network is detected. The second transition fires when the same file is executed. The final transition fires when the program being executed opens a network connection to another host.

2.1 STATL

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 may cause the scenario to move to a new state. For example, an action may be the opening of a TCP connection or the execution of an application. The space of possible relevant actions is constrained by a *transition assertion*, which is a filter condition on the events that may possibly match the action. For example, an assertion may require that a TCP connection be opened with a specific destination port or that an application being executed be part of a predefined set of security-critical applications.

Figure 1 shows a pedagogical example of a STATL attack scenario specification[1]. 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 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.

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 *specification*. The scenario specification represents the scenario's definition and global environment, and a scenario instance represents a particular attack that is currently in progress. The details of the STATL syntax and semantics can be found in [5].

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 *nonconsuming*, *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. 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 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 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.

2.2 STAT Core

The STAT Core module is the runtime for the STATL language. The Core implements the concepts of state, transition, instance, timer, etc. In addition, the STAT Core

[1] Although the state-transition graphic representation of attack scenarios is more intuitive, STATL scenarios can also be specified in pure textual format.

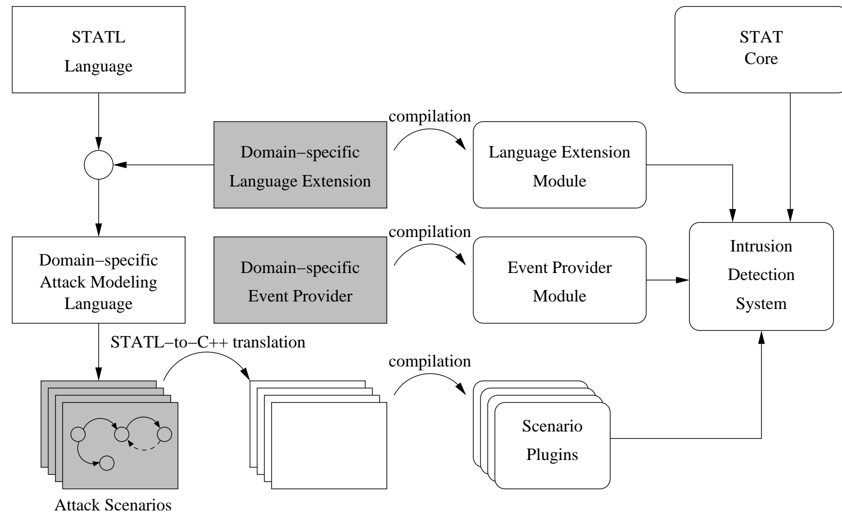

Figure 2: The STAT Framework extension process. The grayed boxes are the modules that need to be developed to extend the framework. The other components are generated automatically through either translation or compilation.

is responsible for obtaining events from the target environment, and matching this event stream against the actions and assertions corresponding to transitions in the active attack scenarios.

2.3 STAT Extensions

The STATL language and the Core runtime are domain-independent. They do not support any domain-specific 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 specific 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 `STAT_Response`. In the following paragraphs, the extension process is presented in detail. A graphic description of the extension process is given in Figure 2.

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 express encapsulation (e.g., Ethernet frames that encapsulate IP datagrams, which, in turn, contain TCP segments) using a tree of events.

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. Then, STATL attack scenarios are 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.

A runtime equipped with Language Extensions, Scenario Plugins, and Event Providers represents a functional intrusion detection system. However, the STAT Framework also provides classes that help define *Response Modules*. A Re-

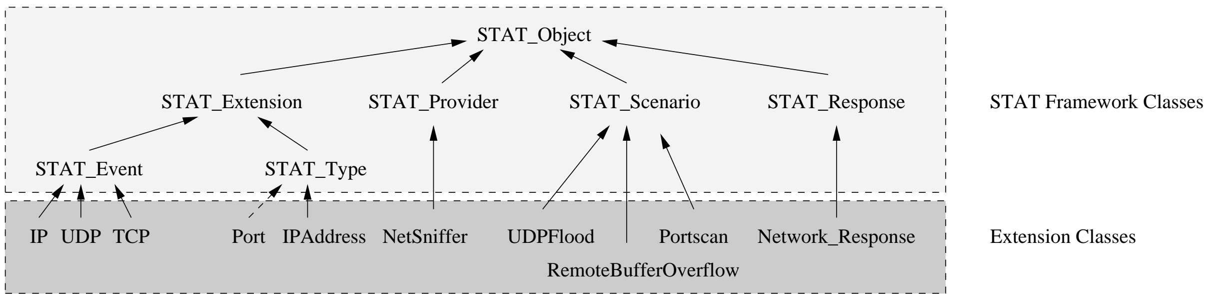

Figure 3: The STAT Framework class hierarchy.

sponse 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 3 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 an instance 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.

3. 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 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 2), a network-based Event Provider, and some network attack scenarios. The resulting system would be a network-based IDS, similar to Snort [25] 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.

3.1 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 [22]. USTAT performs intrusion detection using BSM audit records [27] 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, processes, and files. 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.

3.2 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.

3.3 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.

3.4 AlertSTAT

AlertSTAT is a STAT-based intrusion detection system whose task is to fuse, aggregate, and correlate alerts from other intrusion detection systems (sensors). 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.

3.5 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). The details concerning the amount of reuse for these three systems is presented in Section 5.

3.6 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 [21] 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* [16], and detects attacks that exploit mobile agents.

3.7 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 4. Third, by factoring-out the commonalities between members of the family, it is possible to reuse substantial portions of the code. A quantitative analysis of reuse is presented in Section 5. 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.

4. METASTAT

MetaSTAT is an infrastructure that enables dynamic reconfiguration and management of the deployed STAT-based sensors[2]. MetaSTAT is responsible for the following tasks:

- **Collect and store the alerts produced by the managed sensors.** Alerts about ongoing attacks are collected in a centralized relational database. A schema to efficiently store and retrieve these alerts has been developed, and a GUI for querying and displaying 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 to the appropriate MetaSTAT endpoint.
- **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.
- **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.

MetaSTAT uses a communication infrastructure, called *CommSTAT*, to route messages and alerts between the different MetaSTAT endpoints in a secure and efficient way. CommSTAT messages are based on the IDMEF format, which defines 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 configuration of STAT sensors. For example,

[2]In the remainder of the paper an instance of an intrusion detection system may be referred to as a sensor.

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 *CommSTAT 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:

- **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 CommSTAT 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 CommSTAT proxy, which, in turn, is connected to a MetaSTAT instance. This initial “bare” configuration, which is presented in Figure 4 (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 the external source, filters out those that are not of interest, transforms the remaining events into event objects (as defined by a Language Extension), 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, there are no scenarios, and, therefore, there are no events of interest to be processed. This configuration is described in Figure 4 (b).

System name	Code size	Percent reuse
AlertSTAT	13090	76%
LinSTAT	6107	87%
logSTAT	3589	92%
NetSTAT	22989	65%
USTAT	7755	85%
WebSTAT	5024	89%
WinSTAT	7793	85%

Table 1: Code reuse for each system

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 transferred to 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 specification is created. The scenario specification 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 specification is used as a template to create 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 4 (c).

Finally, if it is desirable to have response functions other than those provided by default, then it is necessary to transfer the appropriate Response Modules to the sensor’s host and load them. The functions in the module can be associated with any state of any active scenario. This is shown in Figure 4 (d).

The Sensor Database and the Module Database are used during the configuration of the sensor to automatically generate a deployment plan from a high-level configuration specification. For instance, a Network Security Officer might want to install a new scenario to catch a recently published attack. The scenario might be dependent on other STAT components, such as Language Extensions and Event Providers, or even external third-party libraries. The information in the two databases is used to generate a list of MetaSTAT control messages that has to be sent to the sensors in order to install all of the components needed to activate the new scenario. These databases and their use are discussed in detail in [30].

5. EVALUATION

The size of the STAT code base is 152,905 lines of code. This includes 42,604 lines of domain-independent components and 66,443 lines of domain-dependent Language Extensions, Event Providers, and Scenario Plugins (see Table 1 for a further breakdown of these numbers.) The remaining 43,858 lines make up the MetaSTAT components.

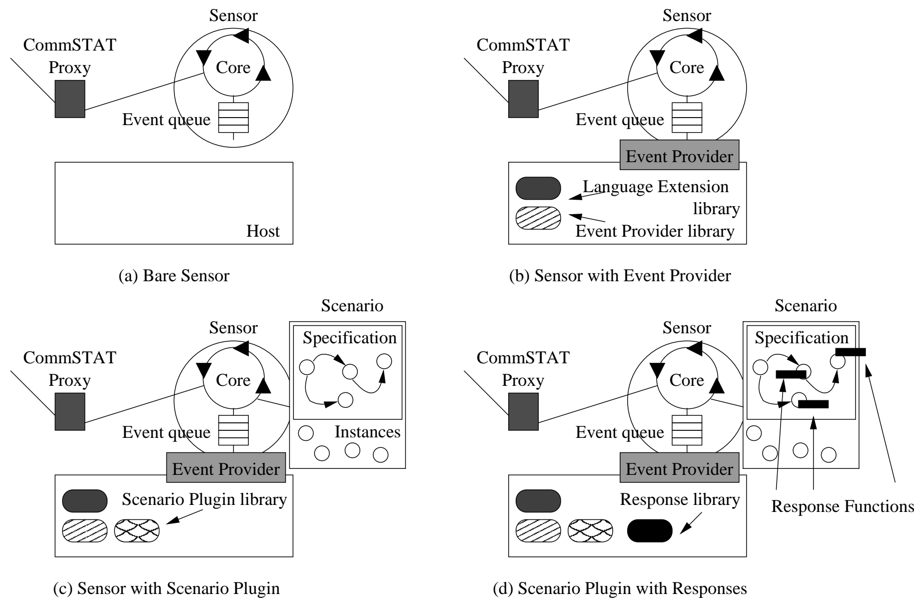

Figure 4: Evolution of a STAT-based sensor.

The code reuse metric used in this paper is the ratio between the domain-dependent code and the domain-independent code measured in lines of code. The results for each of the systems in the STAT software family can be seen in Table 1. When determining the size of the domain-independent code, non-essential components, like MetaSTAT and an editor for the STATL language, have been excluded. Third-party libraries used for tasks like encryption and XML parsing were also excluded from the line count.

It can be argued that the code reuse metric used is inaccurate if the domain-independent code contains large portions of unreachable code. This would cause the reuse percentage to be artificially high. However, this is not the case for the STAT framework. An earlier version of NetSTAT was implemented before the framework existed. NetSTAT was later reimplemented using the framework. By comparing these two versions one can estimate how much code overhead the framework imposes compared to implementing an ad-hoc sensor for a specific domain. The code size of the original NetSTAT sensor was 52,821 lines. The new framework based sensor has a code size of 65,593 lines including the framework code. It follows that the overhead caused by implementing NetSTAT using the framework is 19%. This is not a large overhead given that the framework implementation offers numerous advantages.

Some of the systems implemented have a similar problem domain. When this is the case, it is natural to share some of the domain-specific components between systems, and the percentage of reused code becomes more than just the domain-independent code. As an example, USTAT was the first host-based sensor of the STAT family. When WinSTAT and LinSTAT were developed subsequently, components specific to host-based systems like the UNIX Language Extension and several of the scenarios were reused. Table 2 presents an overview of the different components used by USTAT, LinSTAT, and WinSTAT. The table shows, for each system, which components had to be developed and which components were reused from previous development. The reuse percentage is calculated as the ratio between the number of reused lines of code and the total lines of code.

Even though the use of a framework-based approach may support code reuse in the development of new systems, additional computational overhead might be introduced if the framework is not designed carefully. This is due to the fact that frameworks are designed to provide the functionality shared by possibly very diverse applications, and, therefore, sometimes computational efficiency is sacrificed for the sake of generality. On the other hand, the components that are part of the framework and are often reused may undergo a more thorough testing. In addition, any improvement in their performance positively impacts all the applications that are built by extending the framework or by reusing the optimized components.

We evaluated the performance impact of our framework-based approach by comparing the performance of the original, *ad hoc* version of NetSTAT to the one developed by extending the STAT framework. We ran the two systems on a file containing two days worth of network data from the 1999 MIT Lincoln Laboratory evaluation. The total CPU time was collected for both sensors during multiple runs. The average processing time was 3,220 seconds for the original NetSTAT and 2,862 seconds for the framework-based sensor. The speedup of 13.8% is attributed to careful optimization of the framework source code.

6. 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 [14]. 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 [10, 26], 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, de-

Component/System	USTAT	LinSTAT	WinSTAT
Domain Independent (42604 lines)	r	r	r
BSM provider (4236 lines)	n		
BSM extension (981 lines)	n		
UNIX extension (1240 lines)	n	r	r
UNIX scenarios (397 lines)	n	r	r
BSM scenarios (901 lines)	n		
LinSTAT extension (1111 lines)		n	
LinSTAT provider (2521 lines)		n	
LinSTAT scenarios (838 lines)		n	
WinSTAT extension (1579 lines)			n
WinSTAT provider (3494 lines)			n
WinSTAT scenarios (1083 lines)			n
Total reused code	42604	44241	44241
Total new code	7755	4470	6156
Percentage reused	85%	91%	88%

Table 2: Reuse between host based systems. An `r` symbolizes a reused component, `n` means a new component had to be developed for the specific system.

pending 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 [20] 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 toolsets are becoming more common, their development is still characterized by an *ad hoc* approach. Notable examples are SRI's Emerald [23, 18], ISS RealSecure [13], and Prelude [28]. 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.

7. CONCLUSIONS

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

The work reported in this paper 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. Practitioners in the field of intrusion detection software can certainly gain from the lessons learned. Hopefully, they will use the STAT framework or adapt a component-based software family approach for their development.

We have quantitatively evaluated the effectiveness of the STAT Framework from the reuse point of view. Using a generic, portable runtime that is extended with domain-specific components enabled substantial reuse of resources and reduced development time. These results are not surprising to researchers in the area of framework technologies, but again they may encourage Intrusion Detection software developers to adopt these practices.

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 configuration and control infrastructure and in having the attack specification language tightly coupled with the application development. 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 configuration and control infrastructure can be used to manage a large number of heterogeneous components.

The Language Extension module (STAT_Event and STAT_Type classes) extends the domain-independent STATL core language to allow users to specify attack scenarios in a particular application domain. 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.

Future work will focus on better modeling of the dependencies between components and on management techniques for large sensor networks. In addition, we plan to provide support for extending the analysis engine that is shared by the different tools in the family. By adding new domain-independent analysis techniques to the framework it should be possible to create new IDSs that perform statistical analysis and anomaly detection.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the Defense

Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0207. 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), or the U.S. Government.

8. 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. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

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

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

[8] 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.

[9] K. Ilgun, R. Kemmerer, and P. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[10] T. Inc. Building Object-Oriented Frameworks. White Paper, 1994.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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