The STAT Tool Suite

Giovanni Vigna Steve T. Eckmann Richard A. Kemmerer

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

Abstract

This paper describes a suite of intrusion detection tools developed by the Reliable Software Group at UCSB. The tool suite is based on the State Transition Analysis Technique *(STAT), in which computer penetrations are specified as sequences of actions that cause transitions in the security state of a system. This general approach has been extended and tailored to perform intrusion detection in different domains and environments. The most recent STAT-based intrusion detection systems were developed following a framework-based approach, and the resulting design uses a "core" module that embodies the domain-independent characteristics of the STAT approach. This generic core is extended in a well-defined way to implement intrusion detection systems for different domains and environments. The approach supports reuse, portability, and extendibility, and it allows for the optimization of critical functionalities.*

1. Introduction

The evolution of computer networks fostered a deep change in the way computer systems are structured and information is processed. The centralized approach, characterized by powerful expensive computing nodes accessible from dumb terminals connected through low-bandwidth dedicated lines evolved towards a decentralized pattern where processing is distributed among many small nodes (possibly millions) connected by high-speed networks. This evolution broadened the focus of computer security accordingly. Centralized authentication and access control is no longer the main focus of security research and solutions. The Internet and the use of enterprise-wide TCP/IP networks shifted the focus of security to the protection of communication through untrusted networks, remote authentication, control of the access to network services, support for mobile users, and protection against dynamic download and installation of code coming from untrusted sources.

In world-wide networked settings the mainstream security solution is represented by firewalls and domain-level security [1, 2]. With this approach networks are divided into smaller subnetworks that are under the control of a single authority. These security domains use internal mechanisms and policies to authenticate and authorize users (e.g., Kerberos [11]). Domains are protected against access from outer domains by means of firewalls, which are network filters that regulate the access to an internal network from the outside.

Even though domain-level security mechanisms and the use of firewalls have improved the overall security of networks, attacks and intrusions are still possible and common. This inability to prevent all attacks demands complementary solutions. As a consequence, intrusion detection has received an increasing amount of attention from academic, commercial, and government organizations.

Intrusion detection systems analyze information about the activities performed in a computer system or network, looking for evidence of malicious behavior. The information may come in the form of audit records produced by the operating system auditing facilities, log messages produced by different types of sensors and network devices, and in the form of raw network traffic obtained by eavesdropping a network segment. These data sources are used by intrusion detection systems in two different ways, according to two different approaches: *anomaly detection* and *misuse detection*. Anomaly detection systems are based on models of the "normal" behavior of a computer system. The model may focus on the users, on the applications, or on the network. Behavior profiles may be built by performing statistical analysis on historical data [8] or by using rule-based approaches to specify behavior patterns (e.g., [10]). Anomaly detection compares actual usage patterns against the established profiles to identify abnormal patterns of activity. Misuse detection systems take a complementary ap-

proach. The detection tools are equipped with a number of attack descriptions. These descriptions (or "signatures") are matched against the stream of audit data looking for evidence that the modeled attack is occurring. Misuse and anomaly detection both have advantages and disadvantages. Misuse detection systems can perform focused analysis of the audit data and usually they produce only a few false positives. At the same time, misuse detection systems can detect only those attacks that have been modeled. Anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for in terms of the large number of false positives and the difficulty of training a system with respect to a very dynamic environment.

The *State Transition Analysis Technique* (STAT) [7] was conceived as a misuse detection method to describe computer penetrations as sequences of actions that an attacker performs to compromise the security of a computer system. The STAT approach mitigates the disadvantages of plain signature-based approaches by abstracting from the details of the modeled attacks. Actions are abstracted from their native form (either plain audit records or network packets) to a higher-level representation so that similar actions in a system that may have different low-level representations are mapped to a single action type. In addition, the STAT methodology supports a modeling approach that represents only those steps in an intrusion that are critical for the effectiveness of the attack. By abstracting away from the details of a particular attack, it is possible to detect previously unknown variations of an attack or attacks that exploit similar mechanisms.

The STAT approach has been successfully applied to host-based and network-based intrusion detection and a tool suite based on the approach has been developed. This paper describes the STAT-based intrusion detection tool suite. The current tool suite was developed by leveraging off of an extensible framework for developing STAT-based intrusion detection systems. Section 2 presents the initial experience with STAT and intrusion detection. Section 3 introduces the new design framework for the toolset. Section 4 describes how the framework has been used in developing the new STAT-based tool suite. Section 5 draws some conclusions and outlines future work.

2. STAT and intrusion detection

The *State Transition Analysis Technique* is a method 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. This characterization of attack scenarios allows for an intuitive graphic representation by means of *state transition diagrams* (See for example Figure 1.).

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 state transition analysis technique has been applied to host-based intrusion detection, and a tool, called USTAT [5, 6, 13], has been developed. USTAT uses 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 are monitored by leveraging off of the auditing facilities provided by security-enhanced operating systems, such as Sun Microsystems' Solaris equipped with the Basic Security Module [14]. The first implementation of the USTAT tool 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).

The original USTAT tool interprets the audit trail produced within a single operating system. The USTAT design has been extended to detect attacks that involve multiple hosts sharing network file systems. The resulting tool, called NSTAT [9], uses a client-server architecture to collect audit records from different sources (hosts), merge them into a single audit trail, manage synchronization and correlation among the different trails, and then perform state transition analysis on the resulting trail. Even though NSTAT's components are distributed over a network, NSTAT does not perform monitoring at the network level; its event sources are the auditing facilities available on the monitored hosts.

The natural evolution of state transition analysis was its direct application to networks. NetSTAT was the result of this evolution [15]. NetSTAT is a tool aimed at real-time network-based intrusion detection. NetSTAT takes advantage of the peculiar characteristics of intrusion detection based on the analysis of network traffic. Networks provide detailed information about computer system activity, and they can provide this information regardless of the installed operating systems or the auditing modules available on the

hosts. The NetSTAT tool proved that the STAT approach could be extended to new domains, but the first NetSTAT prototype was also developed ad hoc, building a completely new tool that would fit the new domain.

In the second half of 1998, the NetSTAT and USTAT systems were evaluated as part of both the MIT Lincoln Laboratory's off-line intrusion detection system evaluation [12] and the Air Force Research Laboratory (AFRL) real time evaluation [3, 4]. In the first case, USTAT and NetSTAT were used to analyze BSM logs and network traffic dumps of several weeks of traffic looking for attack signatures. In the second case, NetSTAT and USTAT were installed on a testbed network at AFRL. In both efforts the STAT-based tools performed very well and their combined results scored at the highest level in the evaluation.

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 evidenced 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 tools to match attack scenarios against a stream of events suggested that the STAT-based toolset could be redesigned as a family of systems.

The resulting design is based on a "core" module that embodies the domain-independent characteristics of the STAT approach. The core module provides support for the representation of the domain-independent parts of attack scenarios and implements the domain independent mechanisms used at runtime to match attack scenarios against a stream of events. The core module alone would be useless because intrusion detection is performed in particular domains (e.g., hosts or networks) and in specific environments (e.g., Windows NT or Solaris). Therefore, the new design provides a well-defined way to extend the core and obtain a complete intrusion detection system tailored to the characteristics of a specific domain and environment. The core-based approach supports reuse, portability, extendibility, and customization, and it allows for the optimization of critical functionalities.

3. The STAT core

The STAT core-based architecture is a framework supporting both the development of STAT-based intrusion detection systems and the description of attack scenarios. The framework relies on four elements:

- A language, called *STATL*, that is used to represent attack scenarios using states and transitions. The language defines the domain-independent features of attack scenarios. The STATL language is extended by the intrusion detection system developer to express the characteristics of a particular domain and environment.

- An off-line architecture for the translation of attack scenarios into executable modules, called *scenario plugins*. The architecture provides hooks to extend and customize the translation process to match the peculiarities of the target environment.

- A runtime architecture for the creation of intrusion detection monitors based on the STAT approach.

- A module – the *STAT core* – that implements the domain-independent semantics of scenario execution. The STAT core allows an external application to load scenario plugins and to match an event stream collected from an external source against the attack scenarios.

The remainder of this section discusses these elements in detail.

3.1. A language for scenario representation

The STAT framework provides a language, called STATL, for the specification of attack scenarios. STATL provides constructs to define the domain-independent entities of an attack scenarios and must be extended by the intrusion detection system developer to match a particular domain or environment.

A STATL specification is the description of a complete scenario, which is translated into an executable representation that is loaded into the intrusion detection system at run-time. The execution of a STATL specification generates a runtime *prototype* that contains the data structures representing the scenario's definition and global environment, and a set of *instances* organized in a parent/children structure. Each scenario instance represents an attack in progress. The details of the runtime execution of a scenario are given in Section 3.4. The primitives and constructs used to describe a scenario are introduced in this section.

STATL is primarily a text-based language but the STAT development environment includes a graphic editor that allows for direct visualization of the state transition diagram representing an attack scenario. In the graphical form, states are represented by circles and transitions by arrows. Figure 1 presents a sample state transition diagram with four states and three transitions. Figure 2 shows the textual representation of the scenario.

A scenario has a name, may have parameters, may contain some front matter such as constant and variable declarations, and also contains the states and transitions that define the steps of the attack. Scenario parameters are specified as a list of comma-separated typed identifiers, and they

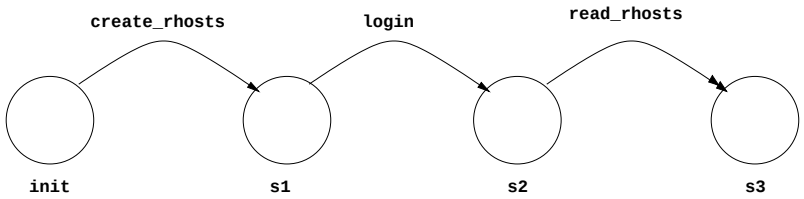

Figure 1. A sample state transition diagram of a STATL scenario.

may specify default values. In addition, a variable number of parameters may be specified by using ellipses (...).

Variables in a scenario may be global or local. Global variables are shared by all instances of the scenario (i.e., by all the attacks of that type). Local variables are instantiated privately in each instance of the scenario. In other words, when a global variable is updated, all instances of the scenario see the update, but when a local variable is updated only the instance in which the update occurs is affected.

A scenario may have *annotations*, which are application-specific directives used by an application-specific analyzer component to customize the translation process of a STATL scenario (see Section 3.2).

The main components of a STATL scenario are *states* and *transitions*. A state has a name, an assertion, a code block, and an annotation. The name is a unique identifier for the state. All the other elements are optional. The state assertion, if present, is a C logical-or-expression, or an expression using the STATL infix operator `in`: `a in b` is True if the value of variable `a` is in the set denoted by variable `b`, otherwise it is False. A state assertion is tested before entry to the state. A state's assertion is implicitly True if none is specified. The state code block is executed after entering the state. A code block is a C compound statement containing declarations followed by statements. For example, in Figure 2 the code block for state `s3` declares a local variable `username` and has two statements, calling the USTAT extension procedure `userid2name` and the built-in procedure `stat_log`. The assertion and the code block have access to the global environment (the set of global variables), the local environment (the set of variables local to that instance), and, in the case of a code block, to the variables declared in the compound statement. A state annotation is a set of application-specific directives to the Analyzer to be used in the translation process.

A transition has a name, a type, an action, an assertion, a code block, and an annotation. The type of the transition can be *consuming*, *nonconsuming*, or *unwinding*. Different types of arrows are used to denote different types of transitions: a solid arc with a single arrowhead denotes a nonconsuming transition, a solid arc with a double arrowhead denotes a consuming transition, and a dashed arc denotes an unwinding transition. A nonconsuming transition is used to represent an evolution of the state of an occurring attack that does not prevent further occurrences of attacks that spawn from the original state. Therefore, when a nonconsuming transition fires both the source and destination attack states become valid. For example, if an attack has two steps that are the creation of a link named `-i` to a SUID shell script and the execution of the script through the created link, then the second step does not invalidate the previous state because another execution of the script through the link may occur. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. For example, an action that deletes a file may invalidate the source state. The file cannot be deleted twice and therefore the original state is invalid. Unwinding transitions represent a form of "rollback" and they are used to describe events and conditions that may invalidate the progress of an attack scenario and require the return to an earlier state. For example, a logout event may make a scenario that involves the associated user uninteresting, and, therefore, the scenario may be brought to a previous state, for instance before user login.

The transition's action is the essential element of a transition. An action specifies an event type that may cause the firing of the associated transition and a name used to refer to the event. An action may be simple or composite. A simple action identifies a unique event type and has a unique name. For example, in Figure 2 transition `create_rhosts` has a simple action of type `USTAT_WRITE` named `w`. A composite action may be the conjunction or the disjunction of actions. In addition, actions may be nested by using square bracket notation. The transition's assertion, code block, and annotation have a syntax and semantics that is similar to the corresponding state elements. The main difference is that the name space includes the action name, and, therefore, the transition's assertion, code block, and annotation may access the fields of the corresponding action.

The STATL language provides the built-in concept of timers. There are three aspects to using timers in STATL: declaration of a timer, starting a declared timer, and using a declared timer as a "timer action" in a transition. A timer declaration specifies the name of the timer and its scope, which can be local or global. Global timers are shared among the instances of a scenario, while local timers are private to each instance. A timer can be started in a code block by an invocation of the built-in procedure

```
// USTAT: create .rhosts by non-owner then rlogin using it.
scenario ftp_write
{
  int user;
  int pid;
  int inode;

  initial state init { }

  transition create_rhosts (init -> s1) nonconsuming
  {
    USTAT_WRITE w :
      (w.euid != 0) &&
      (w.owner != w.ruid)
    {
      inode = w.inode;
    }
  }

  state s1 {}

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

  state s2 { }

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

  state s3
  {
    {
       String username;

       userid2name(user, username);
       stat_log("ftp-write: remote user %s gained local access", username);
    }
  }
}
```

Figure 2. A sample STATL scenario.

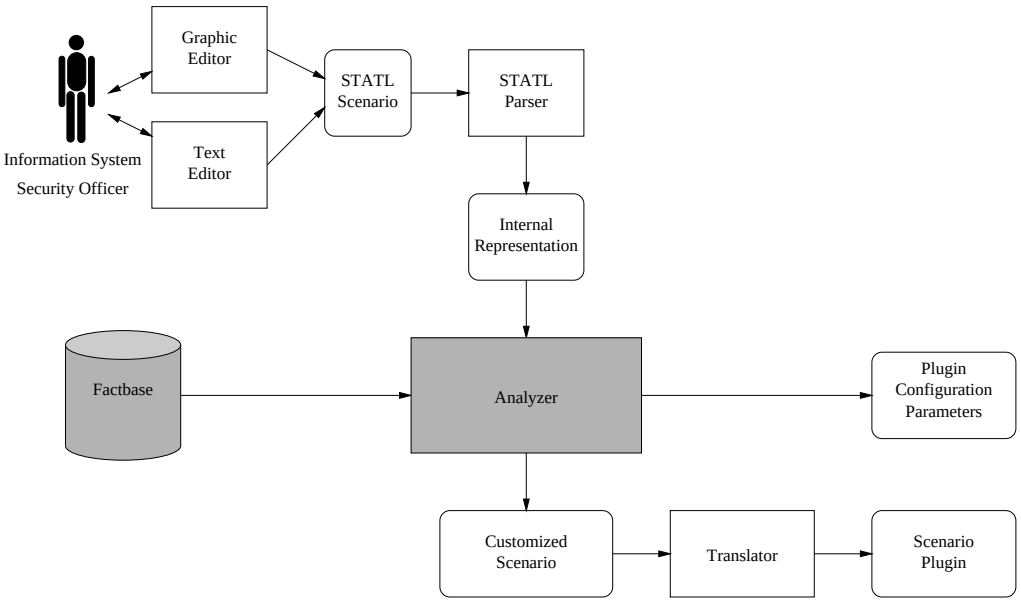

Figure 3. STAT offline process.

`start_timer(t,N)` where `t` is a timer name and `N` is a positive integer number of seconds. After `N` seconds from the start of the timer an event of type `timer` will be generated and active transitions marked with the action `timer t` will fire. Starting a timer that is already running resets the timer.

3.2. Scenario translation process

The STAT core includes four components that support the preparation of the attack scenarios to be loaded into an intrusion detection system (IDS). Figure 3 shows how these four components interact with each other and with a user, such as an organization's Information Systems Security Officer (ISSO), to generate scenario plugins. The Analyzer and Factbase are shaded to indicate that they must be customized for each STAT application (e.g., NetSTAT and USTAT have different Analyzers and Factbases).

The *Scenario Editor* provides a GUI with which a user constructs scenarios in graphical form. The editor produces as output STATL scenarios. The user may also write the scenario directly in STATL. The *Parser* reads a STATL scenario and transforms it into an internal format used by both the *Analyzer* and the *Translator*. Every STAT-based tool uses the same Parser, because STATL is the common scenario specification language. The *Analyzer* reads a scenario in the common internal format and produces a possibly modified copy of the scenario, which may be based on the information contained in the Factbase. The modifications to the scenario that the Analyzer applies are application-specific. A typical modification is to extract *annotations* – parts of the scenario that refer to the Factbase – and use them to generate plugin configuration parameters. The *Translator* reads a scenario in the common internal format, with no annotations, and produces C code. The code produced by the Translator is compiled into a dynamically linked library (e.g., "`.so`" files on Unix, "`.dll`" files on Windows NT), which are linked into the runtime architecture as needed.

3.3. Runtime architecture

Intrusion detection applications based on the STAT-framework share a common high-level architecture. Figure 4 shows the runtime architecture of a complete STAT-based intrusion detection system, with arrows indicating dependencies between components.

The IDS application is composed of an *audit stream provider*, a *preprocessor*, a *STAT core*, a number of *scenario plugins*, and an *extension component*. The audit stream provider component supplies raw event records. For example, in a host-based IDS for the Windows NT environment, the audit record provider may be a module that reads audit records from the operating system's event logging facilities. In a network-based IDS, the audit stream provider could be a network sniffer that reads the network packets broadcast within a subnetwork. The preprocessor component is responsible for filtering out uninteresting audit records, translating interesting audit records to an abstract — but still IDS-specific — format, and then encapsulating the audit records in STAT events. The produced STAT events are sent to the STAT core component for processing.

The STAT core is created and customized by the appli-

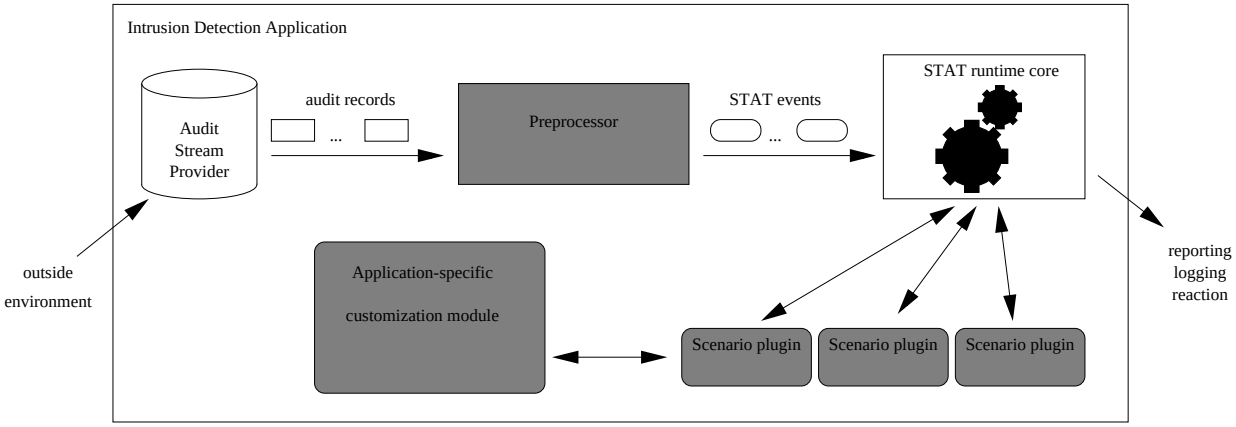

Figure 4. Runtime architecture of a STAT-based intrusion detection system.

cation through an API. The STAT core component is connected to a number of scenario plugin components. Each scenario plugin component contains the executable representation of an attack scenario. The scenario plugins use the types and functions provided by the extension component. These types and functions allow a scenario plugin to access the application-specific characteristics of the processed events and may provide functions to be used in reacting to an attack. For example in the USTAT scenario of Figure 2, `match_name` is a USTAT-specific predicate that checks if the name of an object matches a given string.

3.4. Scenario execution in the STAT core

The main task of a scenario-based intrusion detection system is to match a set of attack scenarios against an event stream, looking for sequences of events and changes in a system's state that are evidence of malicious activity.

At run-time, the core-based IDS loads a number of scenario plugins into the STAT core. Scenario plugins can be parameterized, and the same attack scenario plugin can be loaded into the STAT core with different parameters. For example, a scenario that describes an attack against a WWW server can be loaded providing different web server IP addresses as parameters.

When a plugin is loaded into the core, a *scenario prototype* is created. The scenario prototype is a data structure containing the representation of the attack scenario and the parameters used when loading the plugin. More precisely, the scenario prototype contains the scenario representation in terms of states and transitions and the functions to be invoked to verify assertions, to execute code fragments, and to manage both the local and global environments. When the representation of a scenario is loaded into the core, the action types associated with the signature actions of the scenario's transitions are encapsulated into *event specs* and are inserted in the *event spec database*, which is a global repository that contains a description of all the events types that may be of interest for the scenarios currently loaded into the core.

Scenario prototypes are static representations of attack scenarios. During event processing a scenario prototype is instantiated into *scenario instances* to keep track of particular occurrences of the attack. A scenario instance is characterized by its current state, a set of *event subscriptions*, and its local and global environment. The current state of an instance is one of the states in the scenario representation, as stored in the scenario prototype. An event subscription is the statement of interest of a scenario instance for a certain type of event. An event subscription specifies the scenario instance that is interested in the event and the transition that should be fired if the actual event matches the transition assertion and the destination state assertion. A subscription for a particular event type is stored in the corresponding event spec in the event spec database. The local environment of a scenario instance is a private collection of variables bound to particular values. In addition to the local environment, scenario instances of the same prototype share a global environment that is stored in the scenario prototype. Note that if a plugin is loaded with different parameters then different prototypes are created and each associated set of scenario instances has a separate global environment.

The first scenario instance – the *root* instance – is created at plugin loading time. The root instance is in the scenario's initial state. This first instance acts as the ancestor of new instances that are generated as a consequence of event processing. Should the root instance terminate for any reason, a new root instance would be created automatically.

Once the scenario plugins have been loaded into the core, the core-based IDS reads events from the application-specific event stream and hands the events to the core, where the actual event processing takes place. As a consequence

of event processing, new instances may be created and existing instances may evolve.

Application events cannot be directly processed by the core, because the core does not know about the application-specific characteristics of an event, i.e., the number and types of the application event fields. Therefore, the intrusion detection system encapsulates application-specific events in *STAT events*. A STAT event has a type, a timestamp, and a reference to the application-specific event data, which is considered an "opaque" data structure. An event may be *simple* or *composite*. A simple event has a single type code and a single "opaque" event. A composite event is an ordered list of sets of simple events. The creation of composite events is the responsibility of the external application. The core provides a set of APIs for the creation and composition of events.

When a STAT core event is passed to the core for processing, the core determines the set of instances that may be interested in the event. This is done by matching the event against the event specs in the core's event spec database. For each matching event spec the associated event subscriptions are considered. For each subscription the assertion of the referenced transition is checked. If the assertion is verified then the assertion associated with the transition's destination state is checked. If the destination state assertion is verified the tuple $\langle$*scenario*, *transition*, *event*$\rangle$ is inserted in the set of transitions to be fired. There are three separate sets depending on the type of transition: nonconsuming, consuming, and unwinding.

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

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

After both consuming and nonconsuming transitions have been fired, the core proceeds by firing unwinding transitions. The firing of an unwinding transition with respect to a scenario instance has the effect of undoing the steps that brought the scenario instance to its current state. This means that other scenario instances may be affected by the unwinding procedure. More precisely, if we consider an unwinding transition from state S_x to state S_y we have to remove all the instances that were created by the series of events that brought the unwinding instance from state S_y to state S_x. In the core, this is achieved by traversing back the parent/child chain until an instance in state S_y is found. Then the instance subtree rooted in the last visited instance is removed.

4. Experience with the core

The STAT core-based framework has been used to generate a new tool suite based on the STAT approach. The tool suite includes USTAT, WinSTAT, and NetSTAT. The STAT tool suite was recently used in the 1999 DARPA Intrusion Detection Evaluation effort. For each tool in the suite an extension module containing the application-specific abstract event types and predicates was developed. In addition, an application-specific preprocessor was developed to translate the native audit records into the abstract event representation. For each tool a Factbase and Analyzer were designed to translate STATL scenarios into executable plugins. Their implementation is currently incomplete, and, therefore, most of the translation process must be performed manually.

USTAT is the core-based implementation of the original Unix host-based IDS. USTAT uses Sun Microsystems Basic Security Module (BSM) as a source of audit data.

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

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

NetSTAT is a network-based intrusion detection system. The core-based approach has been used to develop the NetSTAT probe component. A NetSTAT probe uses as inputs the network traffic sniffed from a network segment or may work offline by reading a tcpdump file. The NetSTAT extension of the STAT core includes 13 event types and more than 30 predicates and functions. The NetSTAT preprocessor is responsible for the filtering of relevant network packets and for the parsing and abstraction of network events. The NetSTAT preprocessor tasks include the reassembling of fragmented IP datagrams, the reassembling of TCP streams, the parsing of DNS and RPC events and the maintenance of relevant information about the state of the network (e.g., active connections). The architecture for the off-line development of scenario representation follows the one described in Figure 3. The Factbase component is a repository that stores and manages the security relevant information about a network, such as the network topology and the network services deployed. The Analyzer uses the annotations in STATL scenarios to determine where the probes must be placed in the protected network and how the probes must be configured [16].

5. Conclusions and future work

The core-based framework for the development of STAT-based intrusion detection systems provides a number of advantages. The framework supports efficient development of new IDSs because the main mechanisms and the semantics of the approach are coded in a domain-independent model. Therefore, the IDS developer has to implement only the domain/environment-specific characteristics of the new IDS. In addition, extensions to the core module are developed following a well-defined process.

The core module and its extensions were developed using the GNU build system, which allows for improved portability. In particular, it was possible to port the STAT core module to Solaris, SunOS, Linux, and Windows NT without any modifications.

The core module is a critical component of the tool suite. Therefore, the core has been carefully coded for improved performance. Experience with the ongoing DARPA IDS evaluation showed that the use of the core module brought an increase of an order of magnitude in the speed of the tools in the STAT suite.

Future work will be focused on the translation process and in the continuous improvement of the features of the STATL language.

Acknowledgments

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. 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 Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

References

[1] B. Chapman and E. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, 1995.

[2] W. Cheswick and S. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

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

[6] 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.

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

[8] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[9] R. Kemmerer. NSTAT: A Model-based Real-time Network Intrusion Detection System. Technical Report TRCS-97-18, Department of Computer Science, UC Santa Barbara, November 1997.

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

[11] J. Kohl and C. Neuman. The Kerberos Authentication Service (V5). RFC 1510, September 1993.

[12] M. L. Lab. The 1998 DARPA Intrusion Detection Evaluation. http://ideval.ll.mit.edu/1998_index.html, 1998.

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

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

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

[16] G. Vigna and R. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 1999.