Modeling Network Intrusion Detection Alerts for Correlation

JINGMIN ZHOU
University of California, Davis
MARK HECKMAN and BRENNEN REYNOLDS
Promia, Inc.
and
ADAM CARLSON and MATT BISHOP
University of California, Davis

Signature-based network intrusion-detection systems (NIDSs) often report a massive number of simple alerts of low-level security-related events. Many of these alerts are logically involved in a single multi-stage intrusion incident and a security officer often wants to analyze the complete incident instead of each individual simple alert. This paper proposes a well-structured model that abstracts the logical relation between the alerts in order to support automatic correlation of those alerts involved in the same intrusion. The basic building block of the model is a logical formula called a *capability*. We use *capability* to abstract consistently and precisely all levels of accesses obtained by the attacker in each step of a multistage intrusion. We then derive inference rules to define logical relations between different capabilities. Based on the model and the inference rules, we have developed several novel alert correlation algorithms and implemented a prototype alert correlator. The experimental results of the correlator using several intrusion datasets demonstrate that the approach is effective in both alert fusion and alert correlation and has the ability to correlate alerts of complex multistage intrusions. In several instances, the alert correlator successfully correlated more than two thousand Snort alerts involved in massive scanning incidents. It also helped us find two multistage intrusions that were missed in auditing by the security officers.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.10 [**Software Engineering**]: Design—*Data abstraction*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms: Algorithms, Design, Security

Additional Key Words and Phrases: Alert correlation, alert fusion, capability, intrusion detection

Authors' addresses: Jingmin Zhou, Adam Carlson, and Matt Bishop, Department of Computer Science, University of California at Davis, One Shields Ave, Davis, CA 95616. Mark Heckman and Brennen Reynolds, Promia, Inc., 1490 Drew Ave. Suite 18, Davis, CA 95616.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1094-9224/2007/02-ART4 $5.00 DOI 10.1145/1210263.1210267 http://doi.acm.org/10.1145/1210263.1210267

ACM Reference Format:
Zhou, J., Heckman, M., Reynolds, B., Carlson, A., and Bishop, M. 2007. Modeling network intrusion detection alerts for correlation. ACM Trans. Info. Syst. Sec. 10, 1, Article 4 (February 2007), 31 pages. DOI = 10.1145/1210263.1210267 http://doi.acm.org/ 10.1145/1210263.1210267

1. INTRODUCTION

The wide deployment of signature based NIDSs has created a problem: security officers are unable to handle the massive number of simple alerts generated by the NIDSs. It is well known that real-world intrusions often consist of multiple stages, but the NIDSs only detect each individual stage. There is a demand for tools that can automatically identify the multistage intrusions from a large number of alerts. These tools find alerts involved in the same intrusion incident and correlate them into a single alert. This new alert is helpful for intrusion analysis, since it records a rich set of data of the entire intrusion incident. It also reduces the number of raw alerts, making alert auditing more efficient.

To correlate alerts, we must know their logical relations. Since alerts represent attacks, we are, indeed, interested in the relations between attacks. In this paper, we focus on the *requires*/*provides* model [Templeton and Levitt 2000]. It states that, in a multistage intrusion comprised of a sequence of attacks, the early attacks acquire certain advantages, e.g., information about the system under attack and the ability to perform actions on the system under attack, and use them to support the later attacks that require them. These advantages usually represent the accesses the attacker possesses in intrusions. We call them *capabilities*.[1] We chose the *requires*/*provides* model because it closely fits our purpose to correlate alerts in the same intrusion.

To correlate attacks using the *requires*/*provides* model, we must show the connection between the capabilities provided by the early attacks and required by the later attacks. There are several challenges to this task. First, we have to handle thousands of IDS signatures related to many different components and layers of abstraction in our systems. It requires a systematic and consistent approach to model these attacks unambiguously in different contexts. Secondly, it is impractical if not infeasible for NIDSs to monitor every single step in a complex intrusion because some steps can be *legitimate* activities. Attack correlation needs to handle missing data and still be able to correlate detectable attacks. Third, different attacks can create equivalent effects from the view of an intruder, even though they result in different system states. These attacks generate variants of an intrusion. A good correlator should capture the equivalence between results of different attacks. Previous approaches, however, have difficulties solving these problems. They lack a systematic method of clearly defining the capabilities delivered between attacks. Often based on connecting system state between attacks, they are prone to missing steps in intrusions and cannot handle variants of attacks.

[1]There is a superficial similarity between the definition of a capability in this paper and the usual definition of a capability in the literature. However, they are actually different concepts.

In this paper, we propose a systematic approach that first abstracts the basic building blocks of the capabilities delivered between attacks and then uses them to define the capabilities. All capabilities in different layers of a system abstraction are defined by a single formula. The definition is more expressive and precise than the predicates used by other alert correlation approaches. We then abstract IDS alerts in terms of capability sets and derive logical relations between different capabilities in terms of inference rules. This approach is able to handle missing attacks and capture equivalent effects of different attacks. Several algorithms are developed to correlate alerts based on alert abstractions and inference rules. Our experience in modeling hundreds of signatures of three popular NIDSs shows that it facilitates the task of developing the capability sets based on our model. The experimental results of some well-known and real-world intrusion detection datasets show that the approach is promising at alert fusion and correlation.

2. TERMINOLOGY

A term used in intrusion detection often has different meanings in different contexts. For example, an intrusion is traditionally defined as an action that *successfully* violates certain security policy [Anderson 1980; Denning 1987]. However, today's IDSs often try to detect not only intrusions, but also unsuccessful intrusion attempts or even legitimate activities. Thus, for the sake of clarity, we informally define several terms that will be used throughout this paper.

Definition 2.1 (*Malicious Event*). An event generated by a single attempt to violate certain security policies, regardless of whether the attempt achieves its goal.

By definition 2.1, even if an attempt fails to violate a security policy, the events it generates are still malicious. This conforms to the common understanding of a malicious event [Howard 1997; Allen et al. 1999]. For example, an attempt to overflow a buffer on a nonvulnerable web server is still malicious, even though it fails.

Definition 2.2 (*Suspicious Event*). A nonmalicious event generated by an attempt that has strong logical connections with the malicious events.

For example, some Snort [Roesch 1999] signatures detect IP sweep attempts that do not violate the security policies of many sites. However, these events often have a strong connection to intrusion attempts because the attackers are trying to identify active computer systems.

Definition 2.3 (*Attack* (*Noun*)). A malicious or suspicious event detected by the IDSs.

We shall concentrate on the events that IDSs detect, because usually attacks are only discernable in terms of IDS alerts. Moreover, alert correlation only works on the alerts, and not on the events that the IDSs do not detect. In addition, this definition of attack makes it interchangeable with the IDS alert in the following. Thus, we will not always explicitly state that an attack is represented by the alerts.

Definition 2.4 (*Alert*). A message reported by the IDSs as the result of an attack.

We do not explicitly distinguish false positive (false alerts) from true positives (true alerts) for now. In practice, false alerts generate some noise, but do not cause significant difficulties to our approach.

Definition 2.5 (*Intrusion Incident (Incident)*). A sequence of related attacks within a time frame against a single computer system by an attacker to achieve some goals.

We define many constraints for an incident, such as a single target system, a time frame, and a single attacker, in order to avoid arbitrarily expanding the incident. For example, the attacker may launch attacks on the first day to break into victim system *A*, and a few days later she breaks into system *B* from system *A*. Without any constraints, all attacks in this scenario can be grouped into a single incident. Often correlating all these attacks is unnecessary, considering overhead and complexity.

Definition 2.6 (*Alert Fusion (Aggregation)*). Grouping alerts by their common characteristics; typically, grouping alerts of the same signature and network addresses.

Definition 2.7 (*Requires/Provides (Prerequisite) Relation*). If an early attack provides logical support, e.g., information of or access to the system under attack, for a later attack that requires it, there is a requires/provides relation between the two attacks and the corresponding alerts of the attacks.

Definition 2.8 (*Alert Correlation*). Grouping alerts by their requires/provides relation.

3. RELATED WORK

In order to reduce the number of IDS alerts for better intrusion analysis, many researchers [Bass 1999, 2000; Debar and Wespi 2001; Morin et al. 2002; Valdes and Skinner 2001] have studied methods to fuse the alerts sharing similar attributes. These approaches do not reveal the strong logical relations between different types of alerts. Debar and Wespi [2001] and Morin et al. [2002] proposed to correlate different types of alerts according to known attack patterns in an intrusion. This approach does not capture the diversity of the intrusions.

Templeton and Levitt [2000] propose the *requires/provides* model that defines the logical relations between different attacks in the same intrusion incident. That is, the early attacks acquire accesses and use them to provide logical support to the later attacks that require them. The accesses are defined in terms of system state. The JIGSAW language uses simple predicates to define system state. However, they do not provide a systematic approach to develop the predicates used in the model. In addition, complex logical combinations of the simple predicates are difficult to maintain and are error prone. We propose first to create a systematic model to consistently and precisely abstract the accesses obtained by the attacker as *capabilities*. The definitions of

the capabilities are clear in their meanings and are easy to develop because of their consistency. We then model IDS alerts using the capabilities and develop several novel correlation algorithms based on the alert abstraction.

Cheung et al. [2003] propose the attack correlation language CAML for modeling multistage intrusions. Their approach uses predicates to abstract the attack scenarios in terms of system state. Cuppens et al. [2002]; Cuppens and Miège [2002] present a similar approach using the LAMBDA language. One problem with these approaches is that NIDSs generally are unable to determine host-level system state. Worse, NIDSs are unable to track every step of the intrusions that can change system state. Furthermore, often an attack is to *discover* an existing system state, but not to *create* or *alter* it. All these cause problems with alert correlation based on state transitions. A special predicate *know*, which is out of the domain of the system state, is required. Finally, the simple predicates used in these approaches are often ambiguous, because they do not contain adequate information. To the contrary, our approach abstracts the effect of the attack as capabilities from the viewpoint of the attacker, e.g., the resources and the ability the attacker has obtained after the attacks. This approach brings several benefits. First, variants of an attack that obtain the same result can be abstracted to the same capability regardless of system state in different attack scenarios. Secondly, multiple attack steps do not necessarily change the capabilities from the view of the attacker, though the system state may have changed. Thus, even though IDSs miss some intermediate attack steps, the effect of the attacks do not change in our model. This facilitates correlating alerts. Third, the capability of *knowing* a system state is not special compared to any other capabilities. It is defined as consistently as the others. Finally, the definitions of the capabilities contain enough information to make their meaning unambiguous.

Ning et al. [2004] present an alert correlation approach that is similar to the *requires*/*provides* model. They also define predicates for alert correlation. However, many concepts that can affect the correlation results are not clearly defined in this approach, e.g., true and false alerts. Like other approaches, predicates are often policy-related, e.g., compromise and access. For different policies, the same term often has different meanings. Without context, the meanings of the predicates are ambiguous. Moreover, the authors do not give a systematic and consistent way to develop the predicates. We define capabilities that are all in a consistent form. Each field of a capability is chosen from a well-defined domain with a clear meaning. Thus, our approach provides a systematic way to develop the capabilities precisely and quickly. Furthermore, the capabilities are policy independent so as to simplify the definition and avoid ambiguity. We also provide a finer abstraction of IDS alerts considering different results of the attacks. Our approach is effective in both alert fusion and correlation.

Some researchers [Eckmann et al. 2002; Lin et al. 1998; Pouzol and Ducassé 2002] have studied state-based misuse intrusion detection and improvements using a high-lavel abstraction of signatures. An important difference between these approaches and alert correlation is that the latter does not focus on detecting low-level simple attacks. Instead, it focuses on the attack results and

discovers the logical relations between the attacks. We believe that having IDSs track sophisticated multistage intrusions can be overwhelming, and make it difficult to develop and maintain the IDSs and their signatures.

Sheyner et al. [2002] study algorithms to generate and analyze attack graphs automatically. They use vulnerability information instead of IDS alerts to construct high-level attack graphs. The attack graphs represent the possible path the attackers may take in intrusions. This method focuses on known vulnerabilities of the systems, but the attacker may penetrate the system through legitimate paths or unknown vulnerabilities.

4. CAPABILITY MODEL

An intrusion incident is often composed of multiple steps in attacks. For example, a typical network intrusion incident consists of an IP sweep, a port scan, and a buffer overflow attack. In this incident, from the view of the intruder, the early attacks provide certain capabilities to the later attacks that require them. Intuitively, we want to abstract the capabilities with a well-defined model. Such a model can provide a *standardized* way to develop the capabilities for thousands of IDS signatures, help understand the equivalence between the attacks, and avoid ambiguity. The model should be expressive and flexible because of the diversity of the systems and the intrusions. In addition, the model should be similar to other known models so that it is easy to understand and integrate within the framework of the existing systems. Inspired by the access control matrix model and the object-oriented model, we build up our model: the *capability model*.

4.1 Capability

A *capability* is the basic unit that describes the access that an attacker has obtained during an intrusion. It answers the common questions asked by a security officer when he assesses the damage of an intrusion, e.g., what advantages has the attacker obtained, or what can the attacker do to the system? Intuitively, the access to the system consists of four aspects: the subject (intruder) that owns the access, the object be accessed, the method, and privilege to access the object. Thus, we define a capability as follows:

Definition 4.1 (*Capability*). Let D be a set of network addresses, A be a set of actions, C be a set of credentials, S be a set of services, and P be a set of properties. R is a set of abilities, and $\delta : A \times C \times S \times P \rightarrow R$ is a relation function. A *capability* is a 6-tuple: (*source*, *destination*, *credential*, *action*, *service*, *property*), where $source \in D$, $destination \in D$, $credential \in C$, $action \in A$, $service \in S$, $property \in P$, $\delta(action, credential, service, property) = r$ and $r \in R$. The capability means that the attacker can access the *destination* from the *source* and the access is to perform the *action* on the *property* of the *service* as *credential*.

Example 4.1. The capability (mariner, spurr, smith, exec, /bin/sh, code) means that the attacker accesses host spurr from host mariner, and the access is to execute the code of the program "/bin/sh" as the account (user) smith.

It is inconvenient to write a set of capabilities that share some common fields, such as the set of two capabilities: {(mariner, spurr, smith, exec, /bin/sh, code), (mariner, spurr, smith, exec, /bin/csh, code)}. Therefore, we introduce an extended form of capability: (mariner, spurr, smith, exec, {/bin/sh, /bin/csh}, code), where each field of a capability can be a set of values. Readers should notice two things: first, the sample does not mean to execute program /bin/sh using /bin/csh's code, or vise versa; second, if one picks a single value from the set of each field to create a standard capability, it is an element of the capability set represented by the extended form. In other words, a capability set in an extended form represents a product of the set of all fields. For convenience, we will refer to a capability set in an extended form as a capability in the following unless the difference is important.

Sometimes, it is inconvenient to list all applicable values of a set. We introduce a helper function *ALL* to denote the set. This function acts as a universal quantifier and accepts two parameters: a type identifier specifying the type of values in the set, and an optional qualifier specifying some extra constraints for the set. For example, in function *ALL (file, /)*, the first parameter specifies that the type of values in the set is "file," and the second parameter gives a constraint for the set, i.e., the files in the "/" file system.

Similar to the function *ALL*, we introduce a helper function *ANY* as an existential quantifier. It accepts an explicit set of values as a single parameter, or two parameters similar to function *ALL*. For example, suppose a running IIS web server can be any version prior to 3.0. It is denoted as *ANY* ({*1.0, 2.0, 3.0*}) or *ANY* (*version,* $\leq$ *3.0*).

Below, we explain each field of a capability in detail. In order to avoid ambiguity and facilitate the computation of different values, we introduce types as the constraints to the values of each field.

4.1.1 *Source and Destination.* Ideally, we want to specify the subject of a capability, i.e., the attacker. However this is often unavailable or is hard to model in the computer world. We often only know the source network address of the attacks. Thus, the *source* defines the source of the attacker, i.e., from where the attacker is able to access the system under attack. Similarly, the *destination* specifies the network address of the system accessed by the attacker.

The domain of the network address can be categorized by several subsets, e.g., IP address, Ethernet address, etc. Each subset is defined by a type identifier, e.g., *IP* for an IP address and *Ethernet* for an Ethernet address. Therefore, a network address is written in the form *type:value*, e.g., *IP:192.168.0.1* and *Ethernet:00-08-74-4C-04-95*. For convenience, we will omit the type prefix *IP* for an IP address and replace a real IP address with its corresponding DNS or host name in the following.

An attack that has an impact on a network may result in a set of capabilities whose destinations include all the addresses in the network. For example, an attack crashing a router gives the attacker the ability to disconnect the network behind the router from the Internet. In this case, the destination of the capability set is the network behind the router.

A local attack generates capabilities that have the same source and destination. To correlate only the attacks on a local host, the source and destination are implicitly ignored.

The introduction of source and destination is useful to model remote-to-local and local-to-local attacks. By connecting the capabilities through the address relations, we are able to construct the scenarios of complicated intrusion incidents. For example, an attacker may penetrate a victim host through several remote-to-local attacks and obtain nonroot accesses. She then obtains root accesses on the victim host using some local attacks, and finally uses the victim host to launch remote-to-local attacks against other systems. All phases of the intrusion could be correlated by the relations between the addresses in the capabilities. In this paper, however, we will focus on correlating the remote-to-local attacks.

4.1.2 *Action.* Action abstracts the access method of a capability. Previously a classification of the access method was often ignored when analyzing the network intrusions. This, however, is inadequate for precisely analyzing the impact of the attacks in intrusion detection, which can be critical for correctly correlating the attacks. For example, an attacker may exploit a vulnerability to obtain a capability of reading arbitrary files, but not writing any files. Therefore, a later attack that requires the capability of overwriting files should not be correlated to the earlier attack. A single concept of access is insufficient to describe the difference between these capabilities. Though a classification of the access methods seems complicated, it is often needed. For example, in host-based IDSs (HIDSs), attack scenarios are often described by specific actions, e.g., read, write, or execute files. Thus, we introduce a simple classification for the access method called *action* into the capability model.

The model defines five types of actions: *Read*, *Write*, *Exec*, *Communicate*, and *Block*. The first three types, i.e., *Read*, *Write*, and *Exec* can be easily mapped into relevant concepts in file system access-control mechanism. Thus, the model can describe the accesses applied to file system objects. Furthermore, since similar concepts are widely used in many different layers of abstraction in the computer systems, e.g., from the basic operations of memory blocks to network file sharing, this makes the model quite extensible. However, there are two major limitations. First, it is not natural to model communications using these actions, e.g., to connect to a network process or to send a signal to a process. Second, it is hard to define the effect of denial of service (DoS) attacks, e.g., crashing a network process. Therefore, two extra types, *Communicate* and *Block*, are introduced into the model.

Each type defines multiple values, which are meaningful for appropriate type of objects and their properties in the systems. For example, in type *Read*, **read** can denote the action to read the content of a file, and **list** can denote the action to enumerate the file names in a directory or the account names in a system. Similar to network address, each action is written in form of *type:value*, and we often omit the type prefix for convenience. Table I shows the actions in our current implementation.

Table I. Actions

Action Type	Action Value
Read	read, list, know
Write	create, modify, append, delete
Communicate	send, recv, connect
Exec	invoke, exec
Block	block, delay, spoof

The action **know** of type *Read* defines the state that certain knowledge or information is obtained after performing attacks or other actions, e.g., action **read**. For example, after reading the file "/etc/passwd" in a UNIX like system, the attacker knows the account names. The difference between **read** and **know** is that **read** is detectable by monitoring the system activities, but **know** is undetectable and can only be inferred from detectable activities like **read**.

4.1.3 *Credential.* Intuitively, every action is performed with certain privileges. This is defined as the *credential*.

The capability model defines five types of credentials: *Root*, *User*, *Daemon*, *System*, and *None*, which denote the privileges of an administrative account, a nonadministrative interactive account, a noninteractive account for executing background processes (e.g., daemons in UNIX like systems), the system kernel, and no specific privilege, respectively. Currently, these types are domain specific with respect to operating systems, but they can be extended to other domains, such as a relational database management system (RDBMS). Each of type *Root*, *System*, and *None* contains only a single value, specifically **root**, **system**, and **none**, respectively. Each of type *User* and *Daemon* contains multiple values as defined by the systems. Similar to network address, we often omit the type prefix for a value when it does not cause confusion.

Since there are multiple values in type *User* and *Daemon*, a value of these types should be instantiated at runtime, if possible. For example, an attacker may be able to execute a program as account **smith** or **nobody**. If this information is available at runtime, the credential of corresponding capabilities should be instantiated accordingly. Otherwise, it is instantiated as *ANY (account)*.

Notice that **none** differs from the **nobody** account in many UNIX like systems. The latter is a special noninteractive account for executing processes and has a type *Daemon*, while **none** means the attacker does not need any explicit privilege. For example, to know the running status of the web server on a network host, the attacker may try to connect to port 80 of the host, which does not require the attacker to obtain any explicit privilege on the target system.

Certain actions may be performed with group privilege. In this case, the credential is a set of multiple values denoting the accounts in the group, or a group name prefixed by an identifier *group*. For example, *group:workgroup* specifies that "workgroup" is a group name.

4.1.4 *Service and Property.* Every access is applied to certain objects, which are specified by the *service* field of a capability.

There are a variety of objects in different layers of the system. For example, at the system level there are hardware devices and the operating system (OS);

Table II. Service Types

Type	Sub Type
Hardware system	Network interface, CPU
Software system	OS, DBMS
Process	Daemons, user processes
Account	Users, daemons, root
OS kernel	Kernel modules, network stacks, system calls
File system	Directories, files (programs, scripts)

Table III. Types of Service Property

Service Types	Property Types
CPU	Architecture, processing capacity
Net interface	Ethernet address, bandwidth
OS	Version, patch level, running status
Network stack	Running status, address, sending, replying
System calls	Code
Processes	Running status, version, port, protocol, option
Accounts	Name, password, id, activity, shell, home directory
Directories	Path, existing, permission, owner, timestamp
Files	Link, path, existing, permission, owner, timestamp, content

at the OS level there are kernel modules, processes, accounts, directories, and files; at the application level, like an RDBMS, there are tasks, stored procedures, and accounts. Some objects are passive, such as files, and some are active, such as processes. All of them provide certain functionality or information. Because of this, they are called *services*.

Only some services are relevant to intrusion detection. Through analysis of signatures of three popular NIDSs, i.e., Snort [Roesch 1999], RealSecure [Internet Security Systems (ISS)], and Cisco Secure IDS [Cisco Systems Inc.], we find six types of security-relevant services. Each type may include subtypes, as shown in Table II. It is by no means complete and can be extended, if needed. For example, the services of an RDBMS are not included, but can be added in the future. The services listed in the table are sufficient for the current work.

Each type/subtype of services consists of one or more instances of object. For example, the type of *OS* contains only one object, while the type of *file* contains all file objects in the system. For convenience, we will use the same name for a type and its object, if there is only one object in the type.

Similar to the object-oriented model, each service is associated with one or more properties. An action usually does not operate on a service as a whole. Instead, it is only applied to specific properties of a service. For example, an action like *chmod* only changes the permission of a file as a certain user, leaving the size and other properties of the file untouched. For each property of a service, there can be one or more actions operating on it. For example, one can *read* or *write* the content of a file. Table III defines the types of property for each service type.[2] Again, this is by no means to be a complete set, and can be extended, if needed.

Each type of property can be instantiated by one or more values. For example, there are multiple versions of a program. Sometimes it is inconvenient to

[2]For a program or a script, its property *content* is also named *code*.

instantiate the value of a property, e.g., the actual code of a program. In this case, we use the same name for the type and its value, such as *code*. As a general rule, if the value of a type is not enumerable, we use the same name for the type and its value, e.g., code; otherwise, it is instantiated, for example by version numbers.

Readers should notice that objects of the type *daemons* are instantiated by their functionality, e.g., ftpd or httpd, instead of their names, e.g., *wu-ftpd* or *Apache*. In fact, *wu-ftpd* and *Apache* appear as a part of the version property of a process instead of the process itself. It is meaningless to compare an Apache process to an IIS process in the context of process, but it is meaningful to compare them in the context of the version of the web server. Thus, the version property of an OS or a process usually consists of two parts: the product name and a release number.

In the capability model, a set of relations are defined over the types of action, credential, service, and property. Each relation has an unambiguous meaning. To define a capability, a relation is chosen from the relation set by the type of service, credential, property and action fields of the capability. Currently, the relation set is stored in a database.

4.2 Capability Inference

We now introduce an important component of the capability model, namely the *inference rule*. An inference rule defines the logical relation between two capabilities (or capability sets). It is similar to the ontological rule proposed by Cuppens [Cuppens and Miège 2002]. The separation of capability definition and inference rule grants us great flexibility. In particular, we can define and apply customized inference rules to correlate alerts differently without altering the capabilities we have already developed for alerts. Moreover, a correlation engine is free to pick and apply the rules in any order if needed.

We have developed several inference rules and they are discussed in the following.

4.2.1 *Comparable Inference.* Comparable inference is the simplest and most widely used inference rule in our approach. Before giving its definition, let us look at an example.

Example 4.2. Let C_1 and C_2 be two capabilities: C_1 = (mariner, spurr, root, read, /etc/passwd, content), and C_2 = (mariner, spurr, root, read, ALL(file, /), content). C_1 can be inferred from C_2, since, if one has the capability of reading arbitrary files, one certainly can read the file "/etc/passwd".

Recall that C_2 in example 4.2 represents a set of capabilities. If we expand it into regular form, it will include C_1. Intuitively, we want to develop a method to determine this type of relation without expanding each extended capability. Notice that in example 4.2, the set {/etc/passwd} is a subset of the set of all files and the two capabilities share the same value of source, destination, action, and credential, and the same type of service (file) and property (content). Thus, we derive the following relation:

Definition 4.2 (*Comparable*). Let C_1 and C_2 be two capabilities. C_1 and C_2 are *comparable* if they have the same *value* of source, destination, action, and credential, and have the same *type* of service and property.

By definition 4.2, C_1 and C_2 in example 4.2 are *comparable*.

Definition 4.3 (*Comparable Inference*). Let C_1 and C_2 be two capabilities. C_1 can be inferred from C_2 if C_1 and C_2 are *comparable*, $C_1.service \subseteq C_2.service$, and $C_1.property \subseteq C_2.property$.

Example 4.3. Let C_3 and C_4 be two capabilities: C_3 = (mariner, spurr, none, know, httpd, IIS:4.0), and C_4 = (mariner, spurr, none, know, httpd, Apache:1.3.29). C_3 and C_4 are comparable, but C_3 *cannot* be inferred from C_4 or vice versa.

Comparable inference is implemented by the inclusion relation between sets.

One may wonder whether comparable inference indicates that this type of inference is monotonic. Unfortunately, it is not. For example, if an attacker has a capability to execute the shell program "/bin/sh," she can execute arbitrary programs from that shell program. Thus, the capability of executing arbitrary programs is inferred from the capability of executing a single shell program. This example shows the importance of the semantics of the capabilities.

4.2.2 *Resulting Inference.* Comparable inference only handles the cases where two capabilities (sets) have the same action. However, sometimes two capabilities (sets) have a logical relation, even if they have different actions, as shown by the following example:

Example 4.4. Let C_1 and C_2 be two capabilities: C_1 = (mariner, spurr, root, know, ALL(account), name), and C_2 = (mariner, spurr, root, read, /etc/passwd, content). C_1 can be logically inferred from C_2, because, by reading the file "/etc/passwd," one can obtain the names of all accounts.

Definition 4.4 (*Resulting Inference*). Let C_1 and C_2 be two capabilities. If the exercise of C_2 logically results in C_1, then C_1 can be inferred from C_2.

Unlike comparable inference, this type of inference rule depends on the semantics of each capability. One has to manually derive a rule for each instance of inference, like example 4.4.

Among all the resulting inference rules, there are several special cases. In particular, if an attacker has the capability of executing arbitrary programs on a computer system, then any capabilities with the same credential can be inferred from it. Besides, the attacker may launch attacks against other computer systems from the victim system. The rules for these cases are defined as follows:

Definition 4.5 (*Compromise Inference*). Let C_1 and C_2 be two capabilities. C_1 and C_2 have the same source and destination value, and C_2 is a capability to execute arbitrary programs on the destination i.e., (src, dst, account, exec, ALL(program, /), code). C_1 can be inferred from C_2 if C_1 has the same credential as C_2.

Definition 4.6 (*External Inference*). Let C_1 and C_2 be two capabilities. C_1's source is the same as C_2's destination, and C_2 is a capability to execute arbitrary programs on C_2's destination i.e., (src, dst, account, exec, ALL(program, /), code). Then C_1 can be inferred from C_2.

For the sake of clarity, we define a term about relations between capability sets that will be used in the following.

Definition 4.7 (*Satisfied*). Let S_1 and S_2 be two capability sets, S_1 is *satisfied* by S_2 if S_1 is an empty set or each capability in S_1 can be inferred from a capability in S_2.

If S_1 is satisfied by S_2, or vice versa, then S_1 and S_2 have the *requires*/*provides* relation.

4.2.3 *Inference for Attack Correlation.* The inference rule provides a method to reason about the logical relation between capabilities, and this facilitates correlation of attacks. As discussed earlier, the IDSs often miss intermediate attack steps. Thus, the capabilities provided by an early attack may not be identical to the capabilities required by a later attack. Consider the "Illegal NFS Mount" scenario in Cuppens' paper [Cuppens and Miège 2002]. After a successful NFS mount attack, the attacker has obtained a capability to read and write arbitrary files, including the ".rhosts" file, in the NFS exports. Since the next rlogin attack requires a capability to write to the ".rhosts" file, by applying the comparable inference rule, we can connect the two attacks together, regardless of whether an intermediate attack of modifying ".rhosts" is detected. On the other hand, if correlation is based on the attack scenarios like Cuppens' approach, an explicit intermediate attack scenario is needed. A problem to hypothesize the intermediate attack is that the attacker often can use different attacks to achieve the same goal.

5. CORRELATING IDS ALERTS USING CAPABILITY MODEL

We now use capabilities to model and correlate IDS alerts. Capabilities are the basic primitives for modeling the requires/provides relation among multiple attacks involved in an intrusion. Since attacks are reported by the IDSs in terms of alerts, we model alerts using capabilities, and then construct multistage intrusion scenarios using the requires/provides relations between the alerts.

5.1 Modeling IDS Alerts

Intuitively, each alert requires certain capabilities as preconditions, and provides certain capabilities as postconditions, if it succeeds. Thus, we abstract an alert as follows:

Definition 5.1 (*h-Alert*). An *h-alert H* is the abstraction of an IDS alert in terms of capabilities. It is a 4-tuple (*requires*, *provides*, *failure*, *raw*) where the *requires*, *provides* and *failure* are three sets of *capabilities*: the *requires* denotes a set of capabilities preparing for the attack of the alert, the *provides* denotes a set of capabilities that are *possibly* possessed after the attack successfully achieves its goal, and the *failure* denotes a set of capabilities that are *possibly*

possessed after the attack fails to achieve its goal. The *raw* is a data structure that stores facts about the IDS alert, such as the network address, network protocol, TCP/UDP port number (if applicable), timestamp, and network packet direction.

Next, we explain the fields of an h-alert in detail.

5.1.1 *Requires.* The *requires* field defines a set of capabilities that prepare for the attack of the alert. However, it is *not* a complete set of all required capabilities for the attack, because it is impractical to include all of them. For example, a successful buffer overflow attack may depend on many subtle runtime environment settings, which are often impossible for an attacker to know in advance. In addition, the attacker may launch the attacks even though she does not have a complete set of capabilities. Therefore, we usually choose the capabilities that are explicitly stated in the documentation of the IDS signatures.

5.1.2 *Provides and Failure.* The *provides* field denotes the *possible* consequences of a successful attack. It usually contains the *requires* of the attack. This is because the *requires* represents the preconditions of the attack, which usually still hold after the attack. However, this is not always true. For example, a DoS attack crashing a running system can make previously available services inaccessible after the attack.

Previously in alert correlation one often assumed that the attacks always succeeded [Ning et al. 2004], because many IDSs do not verify the attack result in alerts. Hence, the *provides* of an alert is the capabilities the attacker will possess. However this assumption is not always correct. In fact, it is well-known that NIDSs often report a large number of alerts for unsuccessful attacks. This complicates intrusion analysis. For example, suppose an attack would give an attacker a new capability to read and write arbitrary files if it were successful, but will give no new capabilities if it were unsuccessful. If we know this attack, indeed, is unsuccessful, a future attack that requires the read or write capabilities probably should not be connected to it.

On the other hand, even if an attack is unsuccessful, the attacker sometimes still obtains new capabilities from the attack. For example, a CGI attack to read and write arbitrary files on a system may fail because access to the CGI script is forbidden. However, by analyzing the reply, the attacker can identify the version, and then infer other possible vulnerabilities of the web server. In this case, a new capability giving the version of the web server is stored in the *failure* field of the h-alert for the attack. If IDSs can determine the result of this attack, instead of the *provides*, the *failure* of the h-alert will be augmented to the attacker's capabilities. Thus, the *failure* provides a refined means to model the capabilities obtained in the attack.

It is not trivial to use *failure* because of the limits of the IDS alerts. Sometimes there are two separate alerts for an attack: one records the attack activity itself and the other records the reply message from the system under attack.[3] We are

[3]Zhou et al. [2005] discusses a stateful method to verify the result of NIDS alerts using server responses and, thus, eliminates the second alert.

only able to determine the attack result from the second alert. When we receive the second alert H_2 that tells us the unsuccessful result of the first alert H_1, H_1 was already assumed to be successful and $H_1.provides$ has already been added to the attacker's capabilities. We must *roll back* to the point before $H_1.provides$ was added to the attacker's capabilities, and add to $H_1.failure$ instead. We will discuss a solution for this problem shortly.

5.1.3 *Raw.* The *raw* data structure stores the data of the raw IDS alert and some extra information, such as packet direction and result.

A NIDS alert differs from a HIDS alert in that it is generated from some network packets that have a direction property. For example, Snort can report two alerts for a web CGI attack: a CGI attack alert and an attack response alert. The first alert is triggered by the network packets flowing to the victim web server. It is called a *request alert* in the following. The second alert is trigged by the network packets sent in response by the web server. It is called a *response alert* in the following. Furthermore, if the *response alert* indicates that the attack is unsuccessful, its result is a *failure*; otherwise, its result is a *success*.

Below are several examples of h-alerts.[4]

Example 5.1. The h-alert for an IP sweep attack from host mariner to host spurr is captured as: $requires = \emptyset$, $failure = \emptyset$, and $provides$ = {(mariner, spurr, none, know, IP, run)}.

Example 5.2. The h-alert for an ftp port scanning attack from host mariner to host spurr is captured as: $requires$ = {(mariner, spurr, none, know, IP, run)}, $provides$ = {(mariner, spurr, none, know, IP, run), (mariner, spurr, none, know, ftpd, run)}, and $failure$ = {(mariner, spurr, none, know, IP, run)}.

Example 5.3. The h-alert for an ftpd buffer overflow attack from host mariner to host spurr is captured as:[5] $requires$ = {(mariner, spurr, none, know, IP, run), (mariner, spurr, none, know, ftpd, run)}, $provides$ = {(mariner, spurr, none, know, IP, run), (mariner, spurr, none, know, ftpd, run), (mariner, spurr, root, exec, ALL(program, /), code)}, and $failure$ = {(mariner, spurr, none, know, IP, run), (mariner, spurr, none, know, ftpd, run)}.

5.2 Modeling Correlated Alerts

Like Ning et al.'s [2004] *correlated hyper-alerts*, we introduce a convenient data structure, namely an *m-attack*, to denote a set of correlated alerts.

Definition 5.2 (*M-Attack*). An *m-attack* M is a triple (haset, capset, tmstmp) where *haset* is a set of h-alerts, *capset* is a set of capabilities, and *tmstmp* is the timestamp of M. The *haset* denotes the set of h-alerts that have been correlated. The *capset* denotes a unioned set of capabilities obtained from each h-alert in the *haset*: each h-alert contributes its *provides* or *failure* to the *haset*,

[4]*Raw* is omitted here as it does not affect our discussion.

[5]This example is simplified. In a real attack, the *requires* usually contains an optional capability: knowing version of the ftpd.

depending on its result. Each capability in *haset* is associated with a reference array to record the h-alerts that supply it. The *tmstmp* is the timestamp of the newest *h-alert* in *haset*.

5.3 Capability Qualifier

Ideally, the attacker would launch each step of an intrusion exactly as described by example 5.1, 5.2, and 5.3: first an IP sweep, next a port scan, finally a buffer overflow attack. Alert correlation thus becomes as simple as matching the *requires* of early alerts to the *provides* of the later alerts. However, currently this is rarely true. The attacker often ignores some steps, or the IDSs do not detect some steps. For example, the IDSs may report only two alerts: IP sweep and buffer overflow. Thus, in the example, the *requires* of the buffer overflow alert cannot be satisfied by the *provides* of the IP sweep alert. The condition for correlation is inadequate.

To solve this problem, a *qualifier* is associated to each capability of the *requires* in an h-alert. For example, a qualifier *optional* for a capability states that the capability can be ignored under certain conditions, e.g., if there exists no comparable capabilities in the capability set that it is compared to. An implicit qualifier *mandatory* for a capability means that it must be satisfied by the capability set that it is compared to. In this way, we can tag the *optional* qualifier to the capability (mariner, spurr, none, know, ftpd, run) for example 5.3. Correlation of the IP sweep and buffer overflow alerts can be done even if the port scan alert is missing. Usually, we tag the capabilities with the *optional* qualifier if they are often omitted by the intruder.

The *optional* qualifier requires a change to the *satisfied* relation between capability sets. In particular, assuming there are two capability sets S_1 and S_2, an optional capability in S_1 cannot be ignored unless there exists no *comparable* capability of it in S_2. Thus, the new *satisfied* relation is as follows:

Definition 5.3 (*Satisfied* (*With the Optional Qualifier*)). Let S_1 and S_2 be two nonempty capability sets, let S_1' be a subset of S_1 such that all non-optional capabilities in S_1 belong to S_1', and let S_1'' be a subset of S_1 such that every optional capability that has a comparable capability in S_2 is in S_1''. Then, S_1 is satisfied by S_2 if $S_1' \cup S_1''$ is satisfied by S_2.

At the first glance, the *optional* qualifier for the *requires* is similar to Ning et al.'s [2004] partial satisfaction of prerequisites. However, there are several differences. First, an *optional* capability is not unconditionally ignored as we have discussed. Secondly, a *mandatory* qualifier states that a capability must not be ignored. We believe that this finer control can help avoid potential false correlation.

The qualifiers can be used for capability sets other than the *requires*, as well. For example, in on-line correlation, one has to decide when to report an m-attack, e.g., upon creation of a new m-attack, upon adding new capabilities to an existing m-attack, or when the risk represented by an m-attack exceeds a predefined threshold. For the last case, one can define the risk level for each capability in the *provides* using a qualifier. Thus, if an attacker obtains a high-risk

capability, such as executing the shell program, the corresponding m-attack is reported.

5.4 Correlation Algorithms

Alert correlation is a procedure that combines a new h-alert with one or more existing m-attacks into a new m-attack. For convenience, let us assume the timestamp of the new h-alert to be correlated is newer than the timestamps of all m-attacks to be examined.

Definition 5.4 (*Alert Correlation*). Given a new h-alert H and a set of n existing m-attacks $S = \{M_1, M_2, \ldots, M_n\}$, *alert correlation* is finding an appropriate subset S' of S and combining H with S'. S' is a set of m-attacks $S' = \{M_{i_1}, \ldots, M_{i_k}\}$ $(0 \leq k \leq n)$ such that if S' is a nonempty set, H and all m-attacks in S' are *possibly* involved in the same intrusion incident, because $H.requires$ is satisfied by $\cup_{j=1}^{k} M_{i_j}.capset$. In short, we say H is satisfied by S'. After S' is identified, S' and H are combined into a new m-attack M': $M'.haset = \cup_{j=1}^{k} M_{i_j}.haset \cup \{H\}$, $M'.tmstmp = H.tmstmp$, and $M'.capset = \cup_{j=1}^{k} M_{i_j}.capset \cup H.provides$, if H is successful or $M'.capset = \cup_{j=1}^{k} M_{i_j}.capset \cup H.failure$ if H is unsuccessful. M' is added to the m-attack set S, and all m-attacks in S' may be removed from S afterward.

Therefore, an alert correlator is a program that accepts as input a sequence of h-alerts ordered by their timestamps, correlates each h-alert, and outputs m-attacks. The correlator maintains a set of m-attacks in its memory during runtime, and the set initially is empty.

5.4.1 *M-Attack Set Searching Algorithm.* The essential part of alert correlation is the algorithm to find a subset S' of S. We want to find the subset S' instead of a single m-attack because an h-alert H may require multiple capabilities, each of which is satisfied by a different m-attack. These m-attacks may not be satisfied by each other.

The algorithm to find the appropriate set S' of m-attacks is nontrivial. Finding a minimal set is a set-covering problem that is known to be NP-hard [Cormen et al. 2001]. Moreover, a minimal set may not be optimal for our problem. Thus, we introduce several heuristics to find S'. (1) We assume that two alerts having a closer time distance have a higher probability of being related to each other, and our algorithm assigns a higher priority to them. Thus, for a new alert, the second newest alert among the existing alerts has the highest priority to be correlated to it. (2) The algorithm only looks through a minimal number of m-attacks to find a set that can support the *requires* of a new h-alert, even though there may be more alerts involved in the intrusion incidents. (3) If an m-attack under consideration does not provide a *new* contribution to support the new h-alert, it is not correlated. This means the algorithm will find a set of m-attacks in sequence such that each m-attack adds some new capabilities to support the new h-alert. Ning et al. [2004] proposed a graph connection algorithm that finds all possible requires/provides relations between the alerts. These graphs are hard to understand before graph reduction. We believe that a

```
SEARCH-SET (H, A, n)
INPUT: H is an h-alert, A is an array of n m-attacks
OUTPUT: S', a set of m-attacks
1   Sort m-attacks in A such that A[1].tmstmp ≥ ... ≥ A[n].tmstmp
2   m-attack set S' = ∅
3   capability set C = H.requires
4   for i = 1 to n do
5       if C is satisfied by (S'.capset ∪ A[i].capset) then
6           S' = S' ∪ {A[i]}; C = ∅; break
7       elif C' is satisfied by (S'.capset ∪ A[i].capset) and C' ⊂ C then
8           S' = S' ∪ {A[i]}; C = C − C'
9   if C ≠ ∅ then
10      S' = ∅
11  return S'
```

Fig. 1. M-attack set searching algorithm.

complete graph is unnecessary to discover multistage intrusions. Experiments show that our algorithm can generate the same correlation results as their approach.

Using these heuristics, we have developed an algorithm (Figure 1). For convenience, S'.*capset* denotes the union of *capset* of all m-attacks in S'. The algorithm first sorts n existing m-attacks in an array A by their timestamps into decreasing order (line 1), creates an empty m-attack set S' (line 2), and duplicates the capability set H.*requires* as C (line 3). It then scans from the newest m-attack to the oldest m-attack (line 4): for each m-attack $A[i]$, if C is satisfied by the union of $A[i]$.*capset* and S'.*capset* (line 5), $A[i]$ is added to S' and we have found the set. The loop stops here (line 6) and S' is returned (line 11). Otherwise, if $A[i]$.*capset* provides a nonempty subset of C (line 7), meaning $A[i]$ provides some new contributions to S', but C is not completely satisfied, $A[i]$ is added to S', the contribution is subtracted from C (line 8), and the loop continues on next iteration. If $A[i]$.*capset* does not provide any contributions to S', $A[i]$ is ignored and the loop continues on next iteration. If after the last loop iteration, C is not satisfied by S' (line 9), S' is reset to an empty set (line 10). When the algorithm stops, S' either is an empty set, meaning that no suitable m-attack set is found, or it is a set of the newest m-attacks by which H.*requires* is satisfied.

5.4.2 *Provides Replacement Algorithm.* The algorithm in Figure 1 only provides a generic method to correlate alerts. However, as discussed earlier, IDSs can report two alerts for a single unsuccessful attack: a request alert H_1 followed by a response alert H_2, where H_2 states that the attack of alert H_1 is unsuccessful. However, before the correlator sees alert H_2, it has already processed H_1: the attack is assumed to be successful and H_1.*provides* has been added to the attacker's capabilities. We must roll back the capabilities obtained from H_1.*provides*, and replace them with H_1.*failure*. Thus, we introduce a *provides replacement algorithm* to handle this case. This algorithm is invoked after the *m-attack set searching algorithm* finds a nonempty m-attack set S' for H_2, but before combining H_2, with S'. In this case, H_2.*provides* is always an empty set so that the combination can be done later as usual without any sideeffect.

The algorithm works as follows:

1. Determine that H_2 is a response alert and denotes an unsuccessful attack: in $H_2.raw$, the packet direction is *response* and the result is *failure*. If not, stop here.
2. Find H_1 by walking through the *haset* of each m-attack in S' and looking for a request alert H_1 such that $H_1.raw$ matches the network address, port number, and network protocol, with $H_2.raw$ within a pre-defined time frame. To maximize the accuracy, currently only alerts of the TCP and UDP protocols are handled. If a corresponding request alert cannot be found, stop here. Otherwise, let M denote the m-attack to which H_1 belongs.
3. Subtract $H_1.provides$ from $M.capset$: for each capability C in $H_1.provides$, remove H_1 from the reference array for C in $M.capset$. If the reference array for C in $M.capset$ becomes empty, C is removed from $M.capset$.
4. Add $H_1.failure$. Add the capabilities of $H_1.failure$ to the $M.capset$. The reference array of each updated capability in $M.capset$ is altered accordingly.

5.4.3 *External Correlation Algorithm.* The algorithms we have developed so far focus on correlating alerts targeting the same system. Often after a system is compromised,[6] the attacker uses it to launch new attacks against other systems. This kind of attack can be correlated using the *external inference* rule, and so is called an *external correlation*. We have developed a simple *external correlation algorithm*, which is invoked after the *m-attack set searching algorithm* returns an empty set. This algorithm works as follows: for all m-attacks, find one that contains the capability of executing arbitrary programs on the victim system specified by the new h-alert $H.source$. If such an m-attack is found, it is combined with H.

We must be careful when using external correlation in certain circumstances. For example, two different attackers may compromise the same target host simultaneously. It can be difficult to apply the *external correlation algorithm* if there is not sufficient evidence to determine which attacker launched the external attacks. In this case, our algorithm will find all candidate m-attacks and correlate the new h-alert to each of them. The correctness of this result must be examined by the security officer.

6. IMPLEMENTATION

We have developed a prototype alert correlator based on the capability model discussed in Section 4, and the alert model and alert correlation algorithms discussed in Section 5. This section explains the implementation of the tool.

The tool, as shown in Figure 2, is comprised of three major components:[7] a database to store the capability sets of IDS signatures, a set of preprocessors to convert the raw alerts of different IDSs into a uniform format, and a correlation engine implementing the correlation algorithms.

[6]Compromise here means the attacker can execute arbitrary programs on the victim system.
[7]In the current prototype, inference rules are hard-coded in the algorithms.

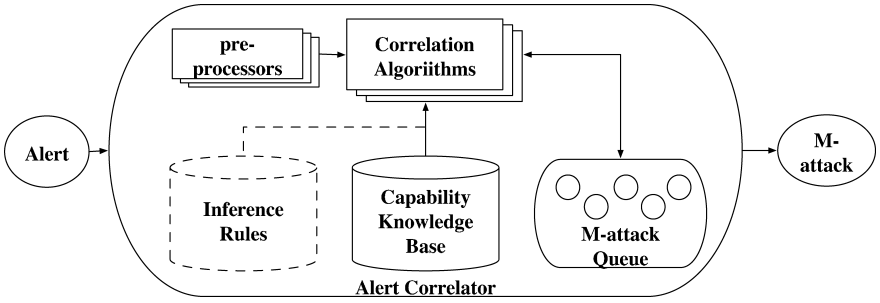

Fig. 2. Prototype alert correlator.

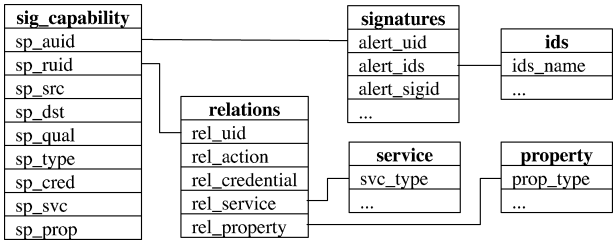

Fig. 3. Capability database schema.

6.1 IDS Signature Capability Database.

The database is a knowledge base storing the capability sets of every IDS signature. Currently three popular NIDSs (specifically, Snort, RealSecure, and Cisco Secure IDS) are supported. A web-based user interface has been implemented to maintain the database.

The schema of the database is built on top of a RDBMS and consists of 15 tables. Part of the schema is shown in Figure 3. Table *sig_capability* is the core of this schema. It defines the capability sets for every NIDS signature. Each capability is a record in this table. The definition of the table refers to many other tables. The fields in the table are:

- **sp_auid** represents the unique identity of an IDS signature. It is defined by tables *ids* and *signatures*. To support the signatures of a new IDS, one needs to create a record for the IDS in table *ids* and one record for each signature in table *signatures*. Each record in table *signature* has a unique identifier and is referenced by table *sig_capability* as **sp_auid**.
- **sp_ruid** refers to the unique identity of a relation defined by table *relations*. A relation is defined as a 4-tuple: (action, credential, service, property), where *credential*, *service*, and *property* specify the data type saved in field sp_cred, sp_svc, and sp_prop. It means to perform *action* on the *property* of *service* as *credential*. For example, it can be (know, user, process, version), which means that the data type of sp_cred, sp_svc, and sp_prop are account name, process name, and version, respectively.

- **sp_src** and **sp_dst** specify the type and constraints of source and destination addresses of the capability. For example, if it is *src.host*, its real value will be instantiated to the source IP address stored in the *raw* data structure of the h-alert. If it is *dst.net*, its real value will be instantiated to the subnet of the destination IP address saved in the *raw* data structure of the h-alert. These values are instantiated once when an h-alert is created.
- **sp_qual** specifies the qualifier of the capability. It must be *optional* or *mandatory* for a capability of the *requires* capability set.
- **sp_type** specifies the capability set that this capability belongs to. Its value can be one of "R," "P," "F", and "C," which represent the *requires*, *provides*, *failure*, and *context* capability sets, respectively. The *context* is an extension that we will explain it shortly.
- **sp_svc**, **sp_prop** and **sp_cred** specify the instantiated values whose type is defined by the sp_uid field of this record.

6.1.1 *Context Set and Config Set.* It is possible for NIDSs to determine the attack results given enough context information [Lippmann et al. 2002]. Thus, we add a table *host_config* into our database. Table *host_config* stores a set of capabilities, each of which is a *know* capability that records the system configuration of the known hosts, e.g., the OS version and the running network services. We use the term *config set* to denote this capability set. Recall that in table *sig_capability* we introduced an extra capability set *context*. Each capability of the *context set* is also a *know* capability, which records the information of the vulnerable OS and network services under the attack specified by the signature. At runtime, by comparing the *context set* of an alert to the *config set* of the victim host specified by the alert, we can tell the possible result of an attack. For example, a CodeRed worm attack [CERT 2001] against an Apache web server should fail because the capability in the *context set* (knowing the vulnerable *httpd* is IIS) cannot be inferred from the capability in the *config set* (knowing the running *httpd* is Apache).

By Definition 5.4, if an attack fails, we should add its *failure* set to the attacker's capability. In practice we could simplify our approach by filtering out the unsuccessful alerts so that they do not go through the correlation process. This is because people tend to ignore those failed intrusion attempts. Though it reduces the precision of the correlation result, we found that it had little impact in our experiments.

6.2 Alert Preprocessor and Correlation Engine

We have created preprocessors to convert the alerts of two NIDSs, Snort and RealSecure, into a uniform format. Thus, the correlation engine does not need to handle different formats of NIDS alerts. Support for the Cisco Secure IDS is under development. The uniform alert is comprised of eight fields: unique signature id, signature name, timestamp, source IP address, source port, destination IP address, destination port, and network protocol.

We assume that the timestamps of NIDS alerts from different sources are correctly synchronized.[8] Approaches to synchronize NIDS alert timestamps are covered by another paper [Ristenpart et al. 2004].

The correlation engine works as follows:

1. **Initialization of H-Alert Template Table** The tool first initializes an h-alert template table from the capability database. One alert template is created for each IDS signature. Each template is an object that consists of four capability sets. The templates are stored in a table T_{alert} indexed by the unique signature id. The entire table T_{alert} is stored in memory. We chose this approach so that an h-alert object can be instantiated from the template object, saving the cost of repeated database queries.
2. **Initialization of Config Set Table** The correlator initializes a *config set* for each known host from tables *host* and *host_config*. These capability sets are stored in a table T_{config}, indexed by the network address of each host.
3. **Instantiation of Alert** The correlator waits for the input of uniform IDS alerts. After a new alert arrives, an h-alert template is found from table T_{alert} by the signature id in the alert. A new h-alert object H is then instantiated from the template as follows: i) The *raw* data structure of the h-alert object is instantiated from the uniform alert; ii) The source and destination address of each capability in the template are computed from the *raw* data structure as discussed in Section 6.1; iii) The capabilities in the *provides* or the *failure* may be instantiated from the *config set*. For example, an *ftp* alert may provide a capability: knowing the *ftp* server is *any* version of wu-ftpd prior to 2.6.2. The real *ftp* server under attack, however, is wu-ftpd 2.6.0 and this information is stored in the *config set* of the victim system. Then, the capability in the *provides* of the *ftp* alert is instantiated to the correct version number, i.e., 2.6.0 in this case.
4. **M-Attack Queue** The correlator maintains a m-attack queue $Q_{mattack}$. A runtime parameter timeout value is used to determine whether an m-attack in $Q_{mattack}$ is too old: if the distance between the timestamp of an m-attack and a new h-alert H is larger than the timeout value, the m-attack is removed from $Q_{mattack}$. This prevents the size of $Q_{mattack}$ from increasing infinitely. It also helps on-line correlation that requires fast correlation: if the size of $Q_{mattack}$ is too large, it may take a long time to search the entire queue. One can set a large timeout value in off-line correlation for correlating slow intrusions.
5. **Context Set Verification** For every new h-alert H, the correlator looks up the *config set* of the victim host in table T_{config}. If found, it is compared with the *context* set of the h-alert. Only if it is satisfied by the *context* set, will the h-alert be considered for correlation. This step is optional and can be turned off by a runtime parameter.

[8] In fact, our approach is often tolerant of incorrectly synchronized alerts. For example, if we switch the order of Example 5.1 (IP Sweep) and 5.2 (Port Scan), we still can correlate the three alerts of Example 5.1, 5.2, and 5.3.

6. **Alert Correlation** The correlator runs the *m-attack set searching algorithm* to look for a suitable set S' of m-attacks. If S' is empty, an optional *external correlation* may be invoked or a new m-attack is created from H and is added into queue $Q_{mattack}$. If S' is not empty, the *provides replacement algorithm* is invoked, and a new m-attack is created by combining H with S'. This new m-attack is added into queue $Q_{mattack}$ and all m-attacks, included in S', are removed from $Q_{mattack}$.

7. EXPERIMENTAL RESULTS

We have tested our correlator with the DARPA 2000 intrusion detection evaluation (IDEVAL) dataset [MIT Lincoln Lab 2000] that is often used to evaluate IDSs, and a real-world intrusion dataset collected at our site.

In all the experiments, we set the timeout value for queue $Q_{mattack}$ to 7200 s, meaning we only correlate alerts less than 2 hours apart.

7.1 Experiments with DARPA 2000 Datasets

The DARPA 2000 dataset includes several five-phase attacks. Ning's approach generated some multi-stage hyper-alert correlation graphs, which are slightly different from the description of the original dataset. (Refer to MIT Lincoln Lab [2000] and Ning et al. [2004] for details).

We used Ning's version of the RealSecure Network Sensor 6.0 alerts corresponding to the DARPA 2000 dataset in order to compare our approach to his. We developed the capabilities for all 28 different alert signatures in this dataset and used the same fields of each signature as Ning.[9] The dataset includes four sets of alerts: two sets from the DMZ and two sets from the inside network. We performed experiments for each set of alerts. The *context set* is not used in the experiments.

When manually examining the correlation results, we first noticed an unexpected interesting effect: the correlator clustered multiple alerts that share the same signature and network addresses. This is understandable. If we assume that the first attack (represented by the first alert) succeeds, its *requires* (preconditions) should hold, and are usually carried over to its *provides* (postconditions). The successive attacks with the same signature and network addresses share the same *requires*, which are satisfied immediately. Thus, they are correlated. This shows that our approach natively supports alert fusion. Thus, unlike Ning's, a separate fusion step is not needed.

We also found many m-attacks that only contain *ftp*, *telnet*, and *sendmail* alerts. They may represent legitimate activities, but RealSecure could not distinguish them from malicious use of these services. Sessions using these services, like the ftp authentication procedure, were also captured in our results, because each session triggered several logically related alerts in sequence. We used the Unix *grep* command to quickly filter them out as well as to filter out the isolated alerts and the m-attacks containing alerts of a single signature. After

[9]The six fields we used are: timestamp, source IP, source port, destination IP, destination port and signature id. A minor difference is that Ning used the signature name instead of id.

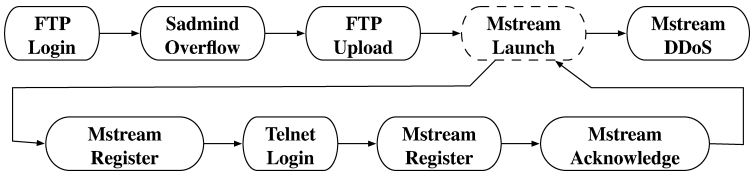

Fig. 4. INT2 intrusion correlation result.

these efforts, we found several multistage intrusion incidents in the remaining m-attacks. After analyzing these m-attacks and reexamining the description of the dataset, we realized that each of the 12 documented multistage intrusions was captured by an m-attack. A further study showed that all alerts involved in these intrusions were correlated in the corresponding m-attacks, and no alerts were missing. This is the same as Cui's [2002] result.

Figure 4 presents an intrusion example in the dataset INT2, where the intruder penetrated host pascal (172.16.112.50) from mill (172.16.115.20) by exploiting a buffer overflow vulnerability in the Sadmind daemon on pascal. The intruder then uploaded an Mstream DDoS program to pascal via ftp and launched the program on pascal to attack other systems. This attack consists of five major phases, in which the fourth phase includes four intermediate steps. Our tool successfully correlated all 16 alerts in this complex incident into a single m-attack. Below, we briefly describe the requires/provides relations between the alerts. Detailed capability definitions are omitted because of space limitations, but can be easily deduced from the description.

1. Alert 1 (FTP_User), assuming it is successful, provides three capabilities required by Alert 2 (FTP_Pass): know the running status of host pascal, know the running status of the ftp server, and know an account name. A new capability, know the account password, is obtained after Alert 2.
2. Alert 3 (Sadmind) requires two capabilities: know the running status of pascal and know pascal's OS version. The first is satisfied by previous alerts. The second is optional and is thus ignored. This alert, in turn, provides a new capability: know the running status of sadmind daemon. Alert 4 (Sadmind_Amslverify_Overflow) requires three capabilities: know the running status of pascal, know the running status of the sadmind daemon, and know pascal's OS version. It provides a new capability: execute arbitrary programs. Alerts 5 and 6 repeat Alerts 3 and 4.
3. Alerts 7 to 9 (FTP_User, FTP_Pass, FTP_Put) represent an ftp session and, for them, the capabilities obtained from the first phase are reused.
4. Alert 10 (Mstream_Zombie) requires three capabilities: know the running status of pascal, know pascal's OS version, and execute the mstream agent program. These are satisfied by the first and second phases. This provides a new capability: know the running status of the mstream program on pascal.
5. Alerts 11 to 13 (TelnetEnvAll, TelnetTerminaltype, and TelnetXdisplay) each require the same three capabilities: know the running status of pascal, know the running status of the telnet server, and know pascal's OS version, in which the second is optional and is only ignored for Alert 11.

6. Alert 14 (Mstream_Zombie) is correlated like Alert 10, and Alert 15 (Mstream_Zombie) is correlated as the reply of Alert 14.[10]
7. Alert 15 (Stream_DoS) requires two capabilities: know OS version of an arbitrary host and know the running status of the mstream program on the host. They are satisfied by the previous alerts when an arbitrary host is instantiated to pascal.

In Cui's [2002] work, the telnet alerts are not correlated. He regarded these alerts as *false alerts* and ignored them. However, his definition of *true* and *false alerts* is unclear, making it hard to decide which alerts should be correlated or ignored. We believe that every alert generated from the attacker's activities relating to the complete intrusion incident should be correlated. In fact, the description of dataset INT2 states that "the attacker notices that the (mstream) server on pascal is not registered with the master and he/she telnets to pascal to re-start that server" [MIT Lincoln Lab 2000]. These scenarios were precisely captured in our results. It confirms our understanding and makes the correlation results more complete.

In addition, the Mstream_Zombie registration alert (e.g., Alert 10) often has a broadcast destination address, since the sender does not know the address of the receiver. To correlate these alerts, we define their *requires* as executing the mstream program on the sender's system, and the OS of the sender's system being Linux or Solaris. Cui [2002] defines the precondition as compromising both the sender and receiver of the alert, which we believe is too strong.

7.2 Experiments with Honeypot Datasets

We have set up four honeynet machines [The Honeypot Project 2001] since June 2003. All the network traffic of these machines have been recorded using Tcpdump [Tcpdump and Libpcap]. We also have monitored the network traffic using Snort inline [Snort Inline]. Three honeynet machines, a Windows NT 4.0 server a Windows 2000 server, and a RedHat Linux 7.2 were repeatedly compromised. Further, some scans to the honeynet generated several thousands Snort alerts per incident.

In all the experiments, we used the *context set* to filter out alerts of unsuccessful attacks, because the configuration of all tested systems was available.

7.2.1 *Intrusions on RedHat Linux 7.2.* There were two major intrusions on the RedHat Linux 7.2 system. On July 31, 2003, a remote attacker exploited a vulnerability in the OpenSSL library [The OpenSSL Project 2002] used by the Apache web server to obtain a nonroot shell. On September 13, 2003, a remote attacker exploited a vulnerability in the wu-ftpd server [Purczynski and Niewiadomski 2003] to obtain a root shell.

The initial experimental result did not show anything about the first intrusion. After examining the data, we found that some signatures of Snort inline

[10]Alerts 10 and 14 differ from 15 that the latter is the reply of the former. For details, please refer to Dittrich et al. [2000]. We can distinguish them because their port numbers differ.

Table IV. The wu-ftpd Intrusion Alerts

Seq. #	Alert Name
1–73	FTP RNFR ./. attempt
74	FTP CWD overflow attempt
75	FTP wu-ftp bad file completion attempt {
76–77	FTP RNFR ./. attempt
78	ATTACK-RESPONSES id check returned root

were turned off by default, so not a single alert about this incident was generated. We turned on all signatures, executed Snort on the Tcpdump data of July 31, 2003, and fed the generated alerts to the correlator. Five Snort alerts about the incident were reported and were successfully correlated. The first alert was a bad HTTP/1.1 request, which provided four capabilities to the attacker: know the running status and OS version of the host and know the versions of the web server and the SSL module. The second alert was an OpenSSL buffer overflow attack, which required all four capabilities obtained from the first alert, and provided a new capability to execute arbitrary programs. The third alert was an attack reply alert that detected the output of a Unix *id* command after a successful buffer overflow attack. It required three capabilities: know the running status and OS version of the victim host, and execute the *id* program. The fourth and fifth alerts repeated the second and third alerts.

The initial experimental result of the second intrusion incident was not satisfactory either. The intrusion resulted in 78 Snort alerts as shown in Table IV, and Alerts 74 and 75 were not correlated. Alert 74 was missed because the *context set* mechanism filtered it out. The desired target of this attack is an ftp server running on Windows, according to the documentation of the signature, but the actual system under attack was a Linux system.[11] Alert 75 was missed for a similar reason: the correlator regarded the wu-ftpd 2.6.2 running on the victim system as nonvulnerable. We found this is because of an obvious error in the Bugtraq database [SecurityFocus 2004], which claims that only wu-ftpd 2.6.1 and earlier are vulnerable. After we fixed the entry in the *context set* for this alert and repeated the experiment, all alerts of this incident (except Alert 74) were correlated. In this incident, the first alert provides four capabilities: know the running status of the host and the ftp server and know the version of the OS and the ftp server. Alerts 2–73 and 76–77 were fused to it. Alert 75 requires all four capabilities obtained by the first alert. Alert 78 requires the three capabilities discussed in the OpenSSL intrusion.

7.2.2 *Intrusions of Windows NT 4.0 and 2000.* The Windows NT 4.0 server was penetrated several times by different attackers through various vulnerabilities in IIS 4.0 and the anonymous ftp server. Our administrators manually audited the observed intrusions and documented them in detail.

We first executed our tool on all Snort alerts for the Windows NT 4.0 server to generate a list of correlated alerts in the form of m-attacks and manually found the m-attacks corresponding to the intrusions identified by the administrators'

[11]This is possibly a documentation error, but we did not fix it, since we did not find evidence to support this hypothesis.

audit. We then clustered the Snort alerts involved in the intrusions based on the attacker's IP addresse that were recorded in the audit report. By comparing the two results, we found they were the same. It meant our tool had successfully correlated the alerts of all observed intrusions. In addition, no alert that should have been correlated was missed.

The tool generated a large number of alert clusters, e.g., there were 1342 m-attacks that contained at least five Snort alerts.[12] However, this number was much smaller than the total number of Snort alerts, i.e., 63,963. We manually examined the 1342 m-attacks and found many consisted of multiple alerts of the same signature and network addresses. After removing them by *grep*, we found many CodeRed v2 worm [CERT 2001] scanning incidents. They share the same pattern of alerts. For each incident, usually 30–80 alerts were reported, and were correlated as an m-attack. Furthermore, we observed six extensive scanning incidents, each of which generated more than 1000 alerts. The largest included 2321 alerts for a single scan! A comparison between these large m-attacks and simple alert clusters, based on IP addresses, shows that the two methods produce the same result.

After filtering out the worm-scanning incidents, we found two intrusions that happened on June 26, 2003 and August 12, 2003. The intruders exploited the Unicode directory traversal vulnerability in the IIS server to copy files into the system. These were missed in the audit report created by the administrators. This result demonstrated that our tool can help reduce the workload of security auditing while keeping important information from being missed.

The experimental results for the Windows 2000 server are similar to the Windows NT 4.0 server. We found a lot of alert clusters that correlated up to thousands of alerts from scans. In addition, based on previous experience, we quickly found an intrusion that utilized a buffer overflow vulnerability in the WebDAV search component of the IIS web server. This intrusion was documented by the audit report.

7.3 Performance

We tested the performance of our correlator on an Intel Pentium 4 3.2-GHz computer with 1-GB RAM running the version of Slackware Linux dated August 9, 2004, and with the Linux kernel version 2.6.7. MySQL 4.0.17 stores the capability database. The current version of the correlator is written in Perl.

We used a file of 63,963 uniform alerts collected from the Windows NT server to test the correlator. It consists of data collected from June 25, 2003 to January 9, 2004 and covers 4,276 distinct IP addresses. The correlator handled approximately 53 alerts/s. It is a CPU-intensive process and can be improved by a faster processor. In addition, we made few attempts to optimize the implementation. We believe that there is great potential to improve the performance.

The correlator process's memory usage stabilized at 18 to 24 MB, the peak usage being 24 MB. We attribute it to the timeout mechanism controlling the

[12]Five is an arbitrary number we chose, but seems adequate for our analysis.

size of queue $Q_{mattack}$. The largest $Q_{mattack}$ we found in the performance experiments contained 165 m-attacks.

8. DISCUSSION

In the current prototype, we used only the comparable and external inference rules. Since comparable inference rules require strictly matching network addresses among the alerts, intuitively we want to compare correlation results to the approach that simply clusters alerts by the same address. Our comparison shows that the two approaches generate the same result for the observed intrusions and scans. This suggests that there are strong logical connections among the alerts involved in the same intrusion and, if the connections are from the same source to the same target, our approach can perform as well as simple alert clustering. Of course, simple alert clustering enjoys higher performance. However, alert correlation reasons about the logical connections among the alerts and helps auditors quickly understand the intrusions. We believe they are complementary approaches.

After we applied the inference rules other than the comparable rules, alert correlation generated better results than simple clustering. As shown in Figure 4, without reasoning using the requires/provides relation, it is hard to connect Alert 16 to the earlier alerts because none of its source and destination addresses are related to earlier ones. Furthermore, many Mstream_Zombie alerts in the DARPA 2000 dataset contain broadcasting addresses. They were all correctly correlated by our tool. This shows the benefit of our approach.

The relaxation of address checking in our experiments did not introduce any false positives (alerts not involved in the intrusion are correlated) or false negatives (alerts involved in the intrusion are not correlated). We attribute this to the strong requires/provides relations between the alerts and careful definition of capabilities. Real life, however, can add complexity. For example, if an attacker penetrates system A from system B using the telnet service, and that session is mixed with legitimate telnet sessions from B to A, it is difficult to avoid false positives, even if we enforce the strict address checking.

We also did some preliminary experiments to further relax checking of the source of the ordinary alerts in order to capture potential intrusions that attack a target from multiple sources. Unfortunately, the result is not encouraging. For example, there are many similar worm scans in our honeynet dataset. If they attacked the same host, and we did not check the source, alerts from all scans would be connected together. We are investigating solutions to determine the appropriate cases for which address checking can be relaxed.

We introduced the *m-attack set searching algorithm* in order to capture the case that multiple earlier attacks together support a new attack, but they do not support each other. Interestingly, we noticed that the correlator always found only one m-attack in the experiments. This suggests that the attacks we have observed are still naive. We expect that more sophisticated attacks, in particular the attacks crossing multiple computer systems, may generate a different result. This is for our future work.

The development of the IDS signature capability database shows the capability model is generic and creates a more systematic development procedure. The model was first designed, based on hundreds of Cisco Secure IDS signatures. We then applied it to develop the capability sets for more than 600 Snort alert signatures (about 1/4 of all Snort alerts as of the writing of this paper). In this procedure, our major task is to choose appropriate values from the well-defined domains of each field of the capabilities, but not to modify the form of capability or to categorize the value of each field. Though it is still a manual task, our experience shows that it is feasible to develop capabilities for large number of IDS signatures, and the development procedure is straightforward. Moreover, we believe that with our model, the experienced IDS signature developers only need a little extra effort to define the capability sets when developing the signatures. For example, in our current capability database, on average, each signature has five capabilities, of which about 1.4 are used by the *context* set and can be quickly identified from the signature document. The capabilities in the *context* set usually have a duplicate copy in the *requires* set, and the capabilities in the *requires* set are often carried over to the *provides* set. Thus, on average, signature developers need to carefully define only two to three capabilities for each signature. In addition, many signatures are similar. For example, there are many signatures for ping scans that use different tools. Their capability sets are almost identical. This significantly reduces the workload.

Our experiences also reveal many problems with existing IDSs. One major problem is the lack of clear and consistent documentation for alert signatures. Good capability sets for the IDS signatures depend on accurate information about the affected systems, applications, potential damage impact, network traffic direction, and so forth. However, this information is often unavailable or incomplete. The capability model thus provides a new standard and framework for better documenting the IDS signatures. Since the form of the capabilities is uniform, documentation using this model can be consistent. Moreover, the capabilities are defined similarly to many popular system mechanisms. This makes documentation based on the model easy to understand. Finally, the model provides a framework for quickly and unambiguously developing the documentation.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an approach to model the capabilities obtained by an attacker from her point of view. We showed that the model abstracts the capabilities in a consistent and systematic manner, and the abstraction is precise without the ambiguity seen in previous work. We then applied the model to abstract IDS alerts, where we considered the capabilities obtained not only from successful attacks, but also from unsuccessful attacks. We also developed several alert correlation algorithms, including a generic algorithm, and some special algorithms to handle specific cases.

Our framework is comprised of three major components: the capability model and capability sets, the inference rules, and the correlation algorithms. This separation of correlation policy (inference rules and correlation algorithms)

from modeling IDS alerts brings great flexibility to our approach. For example, one can define certain inference rules that connect different capabilities via only the relations of the address fields, e.g., the same source and destination. This will turn alert correlation into simple alert clustering, based on network addresses. In addition, we also showed that different algorithms can work together, where some act like *plugins* to a generic algorithm in order to handle special cases. Thus, the approach can handle very different situations without altering the capability model and capability sets of alerts.

We then presented a prototype alert correlator and promising experimental results in alert fusion and correlation on different IDS datasets.

Several research problems remain unsolved. Currently the model works well for NIDS alerts, but its effectiveness on host-based IDS alerts needs to be evaluated. We only used the comparable and external inference rules in correlations and are planning to develop and apply other inference rules in the near future. Currently, the development of capability sets for each IDS signature is a manual task. Approaches to automate this procedure are planned to release this burden. In order to correlate complicated intrusions, e.g., attack steps from different sources, new algorithms need to be developed. Finally, we realize that the current implementation based on RDBMS, is not flexible in handling the relations between the capabilities. Therefore, we plan an approach based on formal languages.

ACKNOWLEDGMENTS

This project was supported by a grant from Promia Inc. to the University of California, Davis. We thank the anonymous reviewers for their valuable comments to improve this paper.

REFERENCES

ALLEN, J., CHRISTIE, A., FITHEN, W., MCHUGH, J., PICKEL, J., AND STONER, E. 1999. State of the Practice of Intrusion Detection Technologies. Tech. Rep. CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University. Jan.)

ANDERSON, J. P. 1980. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co.

BASS, T. 1999. Multisensor data fusion for next generation distributed intrusion detection systems. In *Proceedings of the IRIS National Symposium on Sensor and Data Fusion*.

BASS, T. 2000. Intrusion detection systems and multisensor data fusion. *Communications of the ACM 43*, 4, 99–105.

CERT. 2001. Advisory CA-2001-19 Code Red worm exploiting buffer overflow in IIS indexing service DLL.

CHEUNG, S., LINDQVIST, U., AND FONG, M. W. 2003. Modeling multistep cyber attacks for scenario recognition. In *Proceedings of the DARPA Information Survivability Conference and Exposition*. Washington, D.C.

CISCO SYSTEMS INC. Cisco intrusion prevention alert center, `http://www.cisco.com/pcgi-bin/front.x/ipsalerts/ipsalertsHome.pl`.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, 2nd ed*. The MIT Press. Cambridge, MA.

CUI, Y. 2002. A toolkit for intrusion alerts correlation based on prerequisites and consequences of attacks. M. S. thesis, North Carolina State University, Department of Computer Science.

CUPPENS, F. AND MIÈGE, A. 2002. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the IEEE Symposium of Security and Privacy*. 202.

CUPPENS, F., AUTREL, F., MIÈGE, A., AND BENHERFAT, S. 2002. Correlation in an intrusion detection process. In *Proceedings of the SECI02 Workshop*.

DEBAR, H. AND WESPI, A. 2001. Aggregation and correlation of intrusion-detection alerts. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*.

DENNING, D. E. 1987. An intrusion detection model. *IEEE Transaction of Software Engineering 13*, 2, 222–232.

DITTRICH, D., WEAVER, G., DIETRICH, S., AND LONG, N. 2000. The mstream distributed denial of service attack tool. `http://staff.washington.edu/dittrich/misc/mstream.analysis.txt`.

ECKMANN, S., VIGNA, G., AND KEMMERER, R. 2002. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security 10*, 1/2, 71–104.

HOWARD, J. D. 1997. An analysis of security incidents on the internet. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

INTERNET SECURITY SYSTEMS (ISS). X-force database, `http://xforce.iss.net/xforce/search.php`.

LIN, J.-L., WANG, X. S., AND JAJODIA, S. 1998. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the Computer Security Foundation Workshop*.

LIPPMANN, R. P., WEBSTER, S. E., AND STETSON, D. 2002. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*.

MIT LINCOLN LAB. 2000. DARPA 2000 intrusion detection evaluation datasets. `http://ideval.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html`.

MORIN, B., MÉ, L., DEBAR, H., AND DUCASSE, M. 2002. M2d2: a formal data model for ids alert correlation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection, Zurich, Switzerland*.

NING, P., CUI, Y., REEVES, D. S., AND XU, D. 2004. Techniques and Tools for Analyzing Intrusion Alerts. *ACM Transactions on Information and System Security 7*, 2 (May), 274–318.

POUZOL, J.-P. AND DUCASSÉ, M. 2002. Formal specifications of intrusion signatures and detection rules. In *Proceedings of the Computer Security Foundation Workshop*.

PURCZYNSKI, W. AND NIEWIADOMSKI, J. 2003. wu-ftpd fb_realpath() off-by-one bug. `http://isec.pl/vulnerabilities/isec-0011-wu-ftpd.txt`.

RISTENPART, T., TEMPLETON, S., AND BISHOP, M. 2004. Time synchronization of aggregated heterogeneous logs. In *Proceedings of the Student Workshop on Computing, Department of Computer Science, University of California, Davis*, CA.

ROESCH, M. 1999. Snort—lightweight intrusion detection for networks. In *Proceedings of the USENIX Lisa Conference, Berkeley, CA*.

SECURITYFOCUS. 2004. Vulnerability database. `http://www.securityfocus.com/bid`.

SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. 2002. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium of Security and Privacy*. Berkeley, CA.

SNORT INLINE. `http://snort-inline.sourceforge.net/`.

TCPDUMP AND LIBPCAP. `http://www.tcpdump.org/`.

TEMPLETON, S. J. AND LEVITT, K. 2000. A requires/provides model for computer attacks. In *Proceedings of the Workshop on New Security Paradigms*. 31–38.

THE HONEYPOT PROJECT. 2001. Know your enemy: Revealing the security tools, tactics, and motives of the blackhat community. `http://www.honeynet.org`.

THE OPENSSL PROJECT. 2002. OpenSSL security advisory [30 July 2002]. `http://www.openssl.org/news/secadv_20020730.txt`.

VALDES, A. AND SKINNER, K. 2001. Probabilistic alert correlation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*. Number 2212 in Lecture Notes in Computer Science. Springer-Verlag, New York.

ZHOU, J., CARLSON, A., AND BISHOP, M. 2005. Verify results of network intrusion alerts using lightweight protocol analysis. In *Proceedings of the Annual Computer Security Applications Conference, Tucson, AZ*.

Received November 2004; revised January 2006; accepted September 2006