M4D4: a Logical Framework to Support Alert Correlation in Intrusion Detection

Benjamin Morin [a] Ludovic Mé [a] Hervé Debar [b]
Mireille Ducassé [c]

[a] *Supélec, Rennes, France*
[b] *France Télécom R&D, Caen, France*
[c] *IRISA/INSA, Rennes, France*

Abstract

Managing and supervising security in large networks has become a challenging task, as new threats and flaws are being discovered on a daily basis. This requires an in depth and up-to-date knowledge of the context in which security-related events occur. Several tools have been proposed to support security operators in this task, each of which focuses on some specific aspects of the monitoring. Many alarm fusion and correlation approaches have also been investigated. However, most of these approaches suffer from two major drawbacks. First, they only take advantage of the information found in alerts, which is not sufficient to achieve the goals of alert correlation, that is to say to reduce the overall amount of alerts, while enhancing their semantics. Second, these techniques have been designed on an ad-hoc basis and lack a shared data model that would allow them to reason about events in a cooperative way. In this paper, we propose a federative data model for security systems to query and assert knowledge about security incidents and the context in which they occur. This model constitutes a consistent and formal ground to represent information that is required to reason about complementary evidences, in order to confirm or invalidate alerts raised by intrusion detection systems.

Key words: Intrusion Detection, Alert Correlation, Data Model

1 Introduction

Managing and supervising security in large networks has become a challenging task, as new threats and flaws are being discovered on a daily basis. This requires an in depth and up-to-date knowledge of the context in which security-related events occur. Several systems have been proposed to support security operators in this task. Some examples of these systems include:

- intrusion detection systems (IDS), which monitor the activity of the information system for the occurrence of malicious activities,
- firewalls, which filter inbound and outbound network traffic,
- vulnerability assessment scanners, which discover and report potential risks in computer systems,
- active and passive network mapping systems, which provide a picture of the network nodes and their interconnections (so called topology), as well as the software products running on them (so called cartography),
- honey-pots, which report current trends and threats in the wild and incident databases which inventory attack characteristics.

The scope of each of these systems is limited, both in terms of detection capabilities and in the part of the network they monitor. Therefore, several sensors need to be dispatched throughout the network in order to provide security operators with a comprehensive view of the events that occur. Since each system is likely to produce a large amount of observations, many of which are incomplete, irrelevant or not reliable, security operators are rapidly overwhelmed with events, the analysis of which is complex and time consuming. Thus, it is necessary to assist security operators in the diagnosis of the security incidents in order for them to focus on high priority incidents and take appropriate counter-measures.

Therefore, several event correlation and reasoning approaches have been proposed in the literature to fuse information available in alerts triggered by security mechanisms. However, information available in alerts is generally not sufficient; information about the monitored information system, the characteristics of attacks, and the configuration of security devices deployed throughout the network are also necessary. We argue that having a common data model to describe the relevant security-related information is a prerequisite for the security systems to share a common understanding of the situation at stake, and cooperate.

For this purpose, we propose a data model based on first order logic for security systems to query and assert information about security incidents and the context in which they occur. This model constitutes a consistent and formal ground to represent knowledge that is required to reason about complementary intrusion evidence.

This paper follows prior work on modeling knowledge in the intrusion detection field. In [1], we proposed a relational data model called M2D2, whose objective was to federate the information that is required to reason about alerts in intrusion detection. M2D2 was a first attempt to put together the concepts of alerts, events, vulnerabilities, sensors, network hosts, software products and their relationships.

Our contribution in this paper is twofold. Firstly, we have completely reformulated M2D2 in the first-order logic formalism. This new formalism allows one to translate almost straightforwardly the modelled concepts and relationships in an operational system, thanks to the existing prolog and datalog systems. This revised formalism also allows us to take advantage of logic as a uniform language to represent knowledge databases. Facts, rules and queries can be written in a single language. Moreover, the formalism supports definition of relations via recursive rules, something which is not allowed in traditional databases. Secondly, the new model includes new concepts that were missing in the original M2D2 model. In addition, the modeling of some remaining concepts has been refined. These changes include the description of attack instances and classes, a finer integration of security system capabilities, and the taking into account of routing in networks.

This article is structured as follows. First, we discuss the background and motivation of our work. The next four sections focus on each family of concepts our model is made of: the context (i.e. the characteristics of the monitored information system), the attacks and vulnerabilities, the security devices and the events and alerts that occur in the system. In Section 7, we show how the model can be used to reason about alerts by means of an attack scenario. Then, we briefly describe how a prototype implementation of the logical framework fits within an alert management platform. Before concluding and discussing future work, we present some related work on the subject.

2 Background and Motivations

2.1 Intrusion Detection

Intrusion detection is a field of computer security whose goal is to monitor the activity of an information system for the occurrence of malicious activities, i.e., actions intended to violate the security policy governing confidentiality, integrity and availability of services and data.

Intrusion detection has been a very active research area for the past twenty years, and several complementary solutions have been proposed to detect attacks of all forms and origins against hosts and networks. Despite these efforts, intrusion detection systems (IDS) still suffer from several drawbacks. Firstly, IDS trigger too many alerts, a large proportion of which turn out to be false or irrelevant alerts [2]. Security operators are consequently overwhelmed with alerts, the analysis of which is time consuming and incompatible with the alert rate. Secondly, the detection is still incomplete (i.e., attacks are still missed by IDS). Improving the detection rate requires the proliferation of heterogeneous

sensors, so as to enhance the monitoring coverage and benefit from complementary detection techniques. However, multiplying sensors also multiplies the number of alerts received by security operators. There is a need for intrusion detection sensors to collaborate and exchange information.

2.2 Alarm Correlation

Alarm correlation is a subfield of intrusion detection, whose goal is to make heterogeneous IDS sensors cooperate, in order to improve the attack detection rate, enrich the semantics of alerts and reduce the overall number of alerts.

Alarm correlation cannot be summarized to a single step in the analysis of alerts. Except Valeur et al. [3], who propose a correlation workflow intended to unify the various correlation steps, most correlation approaches proposed in the literature generally focus on specific aspects of the alert analysis.

These correlation approaches can basically be split in two categories, namely the *implicit* and *explicit* ones. Our objective here is not to review all alarm correlation approaches, but to briefly sketch some of them.

Implicit alarm correlation uses data-mining paradigms in order to fuse, aggregate and cluster large alert datasets. For example, the approaches of Valdes and Skinner [4], Dain and Cunnigham [5,6], as well as Debar and Wespi [7] are based on the similarity between alert features (*e.g.,* IP address of the victim and attacker). These processes are crucial to facilitate the analysis of the huge number of intrusion alerts, but generally fail to enhance the semantics of the alerts. Some extensions of these approaches have been proposed to extract relevant information from alert groups, for example by mining association rules between alerts [8]. In [2], Julisch proposes to apply attribute oriented induction techniques in order to generalize alert groups and support root cause analysis. In [9], we proposed an extension of this approach inspired by logical concept analysis, where alarm correlation is tackled as an information retrieval problem. Logical concept analysis unifies querying and navigation of information, which facilitates the investigation of large alert datasets.

Explicit alarm correlation approaches rely on a language which allows security experts to specify logical and temporal constraints between alert patterns in order to recognize complex attack scenarios, which generally require several steps to achieve their ultimate goal. When a complete or a partial intrusion scenario is detected, a higher level alert is generated. For example, in [10], we proposed an explicit correlation scheme based on the formalism of chronicles, and in [11,12] an imperative language to correlate sequences of alerts. Cuppens and Ortalo [13] also proposed a similar language.

An extension of explicit alarm correlation approaches, sometimes referred to as semi-explicit, uses the assumption that complex intrusion scenarios are likely to involve attacks whose prerequisites correspond to the consequences of some earlier ones [14–16]. Therefore, semi-explicit correlation consists in associating preconditions and postconditions, represented by first order formulas, with individual attacks or actions. The correlation process receives individual alerts and tries to build alert threads by matching the preconditions of some attacks with the postconditions of some prior ones.

2.3 Knowledge representation

Despite their differences, almost all of these correlation approaches share a common requirement: the availability of some knowledge about the characteristics of the attacks and the context in which they occur. However, the correlation approaches do not focus so much on how to *represent* the required knowledge, as to how to *use* this knowledge in their reasoning process. These alarm correlation paradigms have generally been implemented on an ad-hoc basis and validated in specific environments or using proprietary formats. We claim that having a consistent data model is a prerequisite for any alert fusion and correlation techniques to be applied. This is why we focus on such model in this paper.

The knowledge representation problem has partially been addressed in previous papers, which we present in Section 9.

As a summary, our intent in this paper is not to propose a new alarm reasoning technique; our objective is to join together the atomic concepts and relations that are required to correlate alerts in a complete and consistent model, called M4D4 ($= (M2D2)^2$). This model is to be used as a basis upon which existing and future correlation techniques can be designed. Nonetheless, we provide a sample alarm correlation scenario which takes advantage of M4D4 in Section 7.

Information modeled in M4D4 can be split in four categories that are described in the remainder of this paper: contextual information (i.e., topology and cartography), attacks and vulnerabilities, analyzers (i.e., IDS, vulnerability assessment scanners and firewalls), and events and alerts. A figure summarizing the main concepts and relationships of M4D4 is provided in appendix.

3 Context Information

In this section, we define our model of the context, i.e. the topological and cartographic data. Modeling information systems is a difficult task because they are increasingly complex and dynamic, but it is critical in order to assess the severity of security events that occur in the monitored system.

3.1 Topology

The network topology model defines network nodes and their interconnections, as well as some logical knowledge about nodes, such as their names. In this paper, we only focus on TCP/IP networks topology, but our model is abstract enough to take into account other network types.

We consider that the supervised network is split into several subnets, defined with predicate $network(N)$. A network address is assigned to them by means of predicate $netaddress(N, A_N)$, where N is a network and A_N is an address in CIDR [1] form, e.g., $netaddress(n,'\texttt{192.168.0.0/24}')$.

Subnets contain nodes, which represent any kind of machine connected to a network. A node H is modelled with predicate $node(H)$ and has an IP address assigned to it by means of predicate $nodeaddress(H, A_H)$, where H is a node and A_H is an IP address. Gateways are specific nodes, whose role is to connect networks together: $gateway(H)$ defines a node as a gateway.

Predicate $nodenet(H, N)$ models the membership of node H in network N. This predicate can be supplied as a fact, or deduced from node and network addressing as follows:

$$nodenet(H, N) \leftarrow \quad nodeaddress(H, A_H) \wedge netaddress(N, A_N) \wedge \\ matches(A_H, A_N)$$

where $matches(A_H, A_N)$ holds if a node address A_H is within the range of a network address A_N (e.g., it holds that $matches(\texttt{192.168.0.5}, \texttt{192.168.0.0/24})$). Of course, a node that is a gateway belongs to more than one network. We assume that there exists a network n_{EXT} that represents the *outside* of the network (e.g. internet), to which edge gateways are connected ($nodenet(G, n_{\text{EXT}})$).

The directly accessible gateways of a node are modelled with predicate

[1] Classless Inter-Domain Routing

$nodegw(H, H_G)$, where H is a node and H_G is a gateway. This predicate is defined as follows: $nodegw(H, H_G) \leftarrow nodenet(H, N) \wedge nodenet(H_G, N) \wedge gateway(H_G)$. We assume that edge gateways are directly accessible by external nodes (i.e., nodes whose address is outside the administration domain):

$$\begin{aligned} nodegw(H, H_G) \leftarrow \quad & \forall N. nodeaddress(H, A_H) \wedge netaddress(N, A_N) \wedge \\ & \neg matches(A_H, A_N) \wedge nodenet(H_G, n_{\text{EXT}}) \end{aligned}$$

Routing information is necessary for alarm correlation purpose in order to estimate the path followed by a packet in the supervised domain and evaluate the capability for a NIDS to detect an attack (see Section 5). Routing information is modelled by means of predicate $routes(H_{G_1}, H_{G_2}, N)$, which models the fact that packets to be delivered to network N are forwarded by gateway H_{G_1} to gateway H_{G_2}.

We now need to estimate the path followed by a packet between a source address A_S and a destination address A_D. We first define a predicate $path(H_{G_S}, H_{G_D}, L, N)$, where N is the final destination network of the packet, L a list of gateways by which a packet transits between a source gateway H_{G_S} and a destination gateway H_{G_D}:

$$\begin{aligned} path(H_{G_S}, H_{G_D}, L, N) \leftarrow \quad & travel(H_{G_S}, H_{G_D}, [H_{G_S}], L, N) \\ travel(H_{G_S}, H_{G_D}, R, [H_{G_D}|R], N) \leftarrow \quad & routes(H_{G_S}, H_{G_D}, N) \\ travel(H_{G_S}, H_{G_D}, R, L, N) \leftarrow \quad & routes(H_{G_S}, H_{G_I}, N) \wedge H_{G_I} \neq H_{G_D} \wedge \\ & H_{G_I} \notin R \wedge travel(H_{G_I}, H_{G_D}, [H_{G_I}|R], L, N) \end{aligned}$$

In this definition of *path*, L is to be instantiated with the possible lists of intermediary gateways.

Given two gateways H_{G_S} and H_{G_D}, $path(H_{G_S}, H_{G_D}, L, \_)$ tries to find a path between H_{G_S} and H_{G_D} in a graph, whose edges correspond to the *routes* predicate. The variable L is to be instantiated with as many lists [2] of gateways as there exists possible paths between H_{G_S} and H_{G_D}. *path* uses an internal predicate, *travel*, which recursively builds the path between H_{G_S} and H_{G_D}; *travel* takes an additional parameter R, which is a list used to remember the intermediary nodes in order to avoid cycles.

Given a source node H_S and a destination node H_D, obtaining the possible routes L (i.e., the lists of intermediary gateways) between H_S and H_D consists

[2] As a reminder, $[H|T]$ is the list comprehension notation, where H is the head of the list and T is the remainder of the list

in first obtaining the nearest gateways H_{G_S} and H_{G_D} of H_S and H_D, and then invoking *path*. This is achieved by the *route* predicate:

$$route(H_S, H_D, L) \leftarrow \quad nodegw(H_S, H_{G_S}) \wedge nodegw(H_D, H_{G_D}) \wedge \\ nodenet(H_D, N) \wedge path(H_{G_S}, H_{G_D}, L, N)$$

Obtaining the path between two IP addresses is simply achieved by getting the IP address associated to a node through predicate *nodeaddress*.

The *route* predicate allows one to calculate possible paths between nodes from the *routes* facts that have been gathered, e.g. by analyzing the routing tables of the gateways. There also exist tools that discover the topology of the network by using different algorithms or techniques. In this case, the above *route* predicate is not used to calculate the route, but its prototype remains the same. The underlying technique used to obtain the topology of the monitored network should be transparent to the processes which need topology information.

The topology model of M4D4 is simple. It does not take into account some techniques employed in networks, such as asymmetric routing for example. Our objective here is to model the basic information found in classical network infrastructures. Enhancement and refinements required to take into account exotic infrastructures is left for future work.

Static network address and port translation are modelled by means of predicate $nat(H_G, A_1, P_1, A_2, P_2)$, where H_G is the gateway that translates incoming connections to address A_1 on port P_1 to address A_2 on port P_2. This information is required to handle cases where two network IDS monitor traffic *before* and *after* a gateway that performs address translation because a target node will not be addressed with the same IP in alerts triggered by both IDS, thus making the aggregation of alerts more difficult.

In addition to the network topology model, we define the following logical information: predicates $nodename(H, SN)$ maps a node H with its system name SN, $resolve(A, DN)$ maps an IP address A with a name DN, as found in various naming resolution mechanisms (e.g., DNS). This is useful for alert correlation purposes because IDS name victims and attackers in various manners, depending on their data sources.

3.2 Cartography

Cartography denotes relationships between nodes and software products. Modeling these relationships is necessary because vulnerabilities affect software

products, and enable attacks against systems.

A software product (product for short) is modelled with a relation $software(S_N, S_V, S_T, S_A)$, where S_N is the name of the product (e.g. `Apache`), S_V its version (e.g. `1.3.29`), S_T its type (e.g. `webserver` or `operatingsystem`) and S_A the architecture the product has been compiled for (e.g. `i386`). The product type is the only mandatory attribute. The predicate $hosts(H, S)$ conveys the fact that a node H hosts a product S.

For alert correlation purposes, it is useful to compare the affected configuration of a vulnerability with the actual set of products of a given host. However, the version number is problematical because there is currently no canonical representation of software product versions, and version notations are heterogeneous (generally an alphanumeric string). Moreover, network mapping tools that rely on heuristics to discover services on hosts generally fail to spot the exact version of a product. We assume there is a partial order relation $\prec$ defined between version numbers, which allows one to compare two versions of the same product.

A process is a product being executed by a user, modelled with predicate $process(S, U)$, where S is a product and U is a username. This information allows one to reevaluate the severity of an attack according to the credentials of the user the process is executed by. Predicate $exec(H, P)$ models a host H executing a process P. The implication $hosts(H, S) \leftarrow exec(H, process(S, \_))$ holds, i.e., any product executed by a node is hosted by that node.

A service is a process listening on a port, modelled with predicate $service(P, Q)$, where P is a process term and Q is a port number. Predicate $listens(H, Ser)$ models a host H hosting a service Ser. As previously, the implication $exec(H, P) \leftarrow listens(H, service(P, \_))$ holds, i.e., any service listening on a node is a process executed by that node.

A node H hosting a web server on port 81 will thus be expressed with the following predicate:

$$listens(H, service(process(software(\texttt{Apache}, \_, \texttt{httpserver}, \_), \texttt{apache}), \texttt{81}))$$

We may distinguish three categories of tools that collect cartographic information. The first category relies on the installation of agents on the monitored hosts that periodically report configuration information to a central inventory database. Microsoft System Management Server is an example of such system. Unfortunately, these agents may not be deployed on all hosts, particularly in hosts that are not maintained by a central IS department. The second category of systems remotely analyze the configuration of devices, and include information related to the security vulnerabilities that may exist on the remote device. Nessus and nmap fall within this category of tools. These tools have

several drawbacks, the most significant of which is their notable side effects on the monitored information system. In [17], we investigate the third category of approaches, namely the passive network mapping, which basically consists in inferring the hosts characteristics by analyzing network traffic. This approach gave satisfactory results to improve the reliability of alerts triggered by IDS.

4 Attacks and Vulnerabilities

In this section, we describe how vulnerabilities and attack classes are modelled in M4D4.

4.1 Vulnerabilities

As defined by Shirey (cf. RFC 2828), a vulnerability is a flaw or weakness in a system (i.e. software product) design, implementation, or management that could be exploited to violate the system security policy.

A vulnerability $vulnerability(V)$ generally does not affect a single product, but a combination of products (e.g., a given web server version running on a specific operating system). We call such a set a *vulnerable configuration* (configuration for short). Predicate $affects(V, C)$ connects a vulnerability with a configuration. We may notice that a single vulnerability may affect several distinct configurations. Predicate $takespartin(S, C)$ models the fact that a product S takes part in a vulnerable configuration C.

From these predicates, we can express the fact that a node H is not vulnerable to a vulnerability V with the following rule:

$$\begin{aligned} not\_vulnerable(H, V) \leftarrow \quad & vulnerability(V) \wedge node(H) \wedge \\ & \forall C.affects(V, C) \wedge \\ & (\exists S_V.takespartin(S_V, C) \wedge \\ & (\forall S_N.hosts(H, S_N) \wedge \\ & S_V \not\approx S_N)) \end{aligned}$$

where $\not\approx$ is defined as follows:

$$S_1 \not\approx S_2 \leftarrow \quad S_1.type \neq S_2.type$$

$$S_1 \not\approx S_2 \leftarrow \quad S_1.type = S_2.type \wedge S_1.name \neq S_2.name$$

$$S_1 \not\approx S_2 \leftarrow \quad S_1.type = S_2.type \wedge S_1.name = S_2.name \wedge S_1.version \prec S_2.version$$

Here, $S_1.name$ is a shorthand to denote attribute *name* of a product. We assume relation $\prec$ is defined over the set of product versions.

This rule states that a host H is not vulnerable to a vulnerability V as long as every configuration affected by V involves at least one product that does not match ($\not\approx$) any of the products hosted by H. We define *not_vulnerable* instead of *vulnerable*, as the assessment of the former is generally more reliable than the latter. This is notably due to the aforementioned problem of precise product version identification.

In addition to the product configurations affected, M4D4 also takes into account the following vulnerability characteristics:

- $severity(V, Sev)$ models the severity of a vulnerability V, where Sev is a string which represents the estimated danger associated to the vulnerability (e.g. "high", "medium", "low");
- $requires(V, W)$ models the access level W ($W \in \{$`remote`, `local`, `user`$\}$) required to exploit vulnerability V;
- $losstype(V, Con)$ models the consequence Con of a successful exploitation, $Con \in \{$`confidentiality`, `availability`, `integrity`, `privilege_escalation`$\}$;
- $published(V, Date)$ provides the age of a vulnerability by means of its publication date $Date$.

Several organizations aim at providing standardized names for vulnerabilities. Mitre's Common Vulnerabilities and Exposures (CVE) [3] list [18] and Bugtraq IDs are examples of such initiatives. Modeling these references is important because many security tools use them to describe their observations. Vulnerability references thus constitute a *de facto* common attack naming convention between heterogeneous security tools (vulnerability scanners and intrusion detection systems, for instance). Predicate $refersto(V, O, V_N)$ provides the unique name or serial number V_N affected by organization O to vulnerability V.

The Mitre also provides a list of name equivalences between CVE and other vulnerability names, which we model with a predicate $equiv(O_1, V_{N_1}, O_2, V_{N_2})$, where O_i is an organization name (e.g. bugtraq) and V_{N_i} is the unique name of a vulnerability for this organization. This relation is used to cluster vulner-

[3] `http://mitre.nist.gov/`

ability names which refer to the same vulnerability. Ideally, *equiv* should be an equivalence relation, but it is not in reality. As Mann notices in [18], the mapping between vulnerability names and CVE names is seldom one-to-one, so a non-CVE name may be equivalent to more than one CVE names. As a result, from one non-CVE vulnerability name, it is possible to get many CVE vulnerabilities.

The vulnerability characteristics modelled above can be extracted from several available resources. For example, the National Vulnerability Database [4] (NVD, formerly known as ICAT) and the Open Source Vulnerability Database [5] (OSVDB) are two independent initiatives which aim at structuring information about known vulnerabilities. The OVAL [6] (Open Vulnerability and Assessment Language) project is also of special interest to our work. OVAL provides a collection of XML schema for representing 1) vulnerability definitions, 2) system characteristics and 3) assessment results. OVAL also provides a repository that contains OVAL vulnerability, compliance, system inventory, and patch definitions for various applications and operating systems. Because it is standardized and machine-readable, the OVAL repository seems to be the best candidate as a vulnerability definition data source for M4D4.

4.2 Attack Classes

Despite many attempts in this domain, there is currently no agreement between actors in the security area on how to describe the characteristics of attacks. Several authors have proposed taxonomies or ontologies, but security tools vendors still use their own convention to describe and name attacks. Thus, vulnerability identifiers generally remain the only common denominator for the attack names among security tools, which is insufficient. Indeed, not all attacks exploit a vulnerability, either because it is irrelevant (e.g. a flooding DoS is an attack but it does not exploit any vulnerability), or because the device is not aware of any vulnerability associated with a detected attack (e.g., anomaly-based IDS do not provide any reference to a known vulnerability since by design they do not *recognize* an attack).

Therefore, we need a way to classify attacks in order to reason about alerts coming from heterogeneous sources. The term "attack" can be misleading: it may solely refer to the attack *class*, i.e. the *method* used by an attacker (e.g. the exploitation of some specific vulnerability); it may also refer to an attack *instance*, which includes the identification of the protagonists (victim and

[4] `http://nvd.nist.gov/`
[5] `http://www.osvdb.org`
[6] `http://oval.mitre.org/`

attacker), a date, as well as the method. We are concerned here in modeling attack classes. Attack instances are modelled in Section 6.

For this purpose, we propose to reuse the suggestion of Goldman et al. [19] to build a dictionary of attack descriptions, that is to say a *lingua franca* that contains characteristics and names of attack classes.

Our attack description dictionary is modelled as directed acyclic graphs whose nodes are attack class names, and edges are inheritance relationships between these classes. The unary predicate $attackclass(K)$ is used to assert an attack class K. The binary predicate $inherits(K_1, K_2)$ models the inheritance relationships and means that the attack class K_1 is a direct sub-class of attack class K_2. For instance, $inherits(attackclass(\texttt{codered}), attackclass(\texttt{worm}))$ conveys the fact that `codered` is a kind of `worm`.

The leaves of the graphs correspond to the complete descriptions (or names) given to attacks by analyzers. This is necessary to take into account the various names given by security device vendors for a single attack class. For example, the virus "Virut" is called "`Virus.Win32.Virut.a`" by F-Secure and "`W32/Virut-A`" by Sophos; in this case, both of these names would *inherit* the generic attack class `Virut`.

The internal nodes of the graph denote characteristics of attacks, such as `worm` or `bufferoverflow`. These characteristics allow one to reason over the features of the attacks (e.g. the fact that the intent of an attack is to execute code on a victim), rather than attack names only (e.g., `W32/Virut-A`), which are generally meaningless. For example, a rule may be to set a high severity to any alert reporting an attack that inherits the `worm` characteristic if the attacker is inside the monitored domain, and a low severity otherwise. Indeed, a worm that propagates from the inside of a network means that a host has been infected, whereas external attacks are regular phenomena [7]. Without the attack dictionary, this rule should be defined for every existing worm name of every vendor.

For illustration, an excerpt of attack dictionary is depicted in Figure 1.

The predicate $attacksubclass(K_1, K_2)$ holds if an attack class K_1 is a subconcept of an attack class K_2 transitively; it is defined recursively as follows:

$$attacksubclass(K_1, K_2) \leftarrow inherits(K_1, K_2).$$
$$attacksubclass(K_1, K_2) \leftarrow inherits(K_3, K_2) \wedge attacksubclass(K_1, K_3).$$

We suggest to build the attack dictionary empirically, by drawing our inspira-

[7] By definition, a worm propagates autonomously

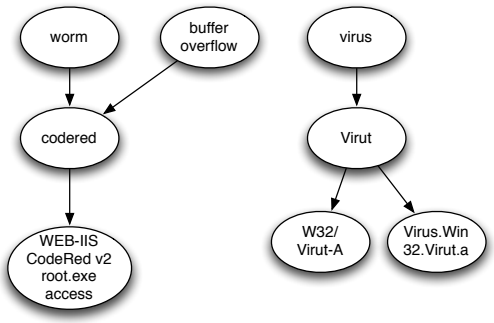

Fig. 1. Excerpt of an attack graph

tion from the human-readable attack descriptions available in existing tools. For example, the `msg:` field present in each signature of the Snort IDS contains candidate keywords for describing attacks.

Building the graph structure essentially requires human expertise to identify the relevant keywords and inheritance relationships. This can be a difficult task, given that the current Snort ruleset contains more than 3000 distinct attack descriptions. However, text analysis techniques can provide satisfactory results to extract recurrent keywords and their relationships. As an illustration, an excerpt of the keyword hierarchy obtained by applying the Camelis tool [8] on the Snort ruleset is given in Figure 1 in appendix.

Let us notice that previously "unknown" attack descriptions raise an issue. For example, anomaly based IDS cannot *name* attacks, since by design they do not *recognize* them. Thus, the description of an attack by such an analyzer may not be known until the first occurrence of the attack. These attack descriptions are consequently recorded in the knowledge base as orphan nodes of the attack graphs, since they cannot be linked a priori with other nodes by means of inheritance relationships. Nonetheless, these links can subsequently be added by experts, manually.

An important characteristic of attack classes is the type of product or architecture they target. Instead of defining a direct relationship between attack classes and products, we suggest to link attack classes with vulnerabilities. The predicate $exploits(K, V)$ conveys the fact that an attack class K exploits (or is an instantiation of) vulnerability V. Next, vulnerability V can be linked to a product configuration by means of predicate $affects$, defined previously. Here, V is a vulnerability identifier, and may not necessarily have a name associated to it by an organization like CVE. This allows an attack class to be linked to an affected product or architecture, even if no referenced vulnerability exists.

For example, one technique employed by misuse IDS to detect buffer overflow

[8] `http://www.irisa.fr/LIS/ferre/camelis/`

attacks is to monitor event flows for the occurrence of long strings of `0x90` bytes, which are indicative of shellcodes for Intel architectures. The description of such an attack by an IDS could be "*Exploit* of a vulnerability that *affects* a software product compiled for Intel architecture". We see that a vulnerability is implicitly exploited, but we are unable to identify it explicitly. This statement is translated as follows:

$$attackclass(k) \wedge vulnerability(v) \wedge configuration(c)$$
$$\wedge takespartin(c, software(\_, \_, \_, \texttt{i386})) \wedge affects(v, c)$$
$$\wedge exploits(k, v)$$

This knowledge allows us to conclude that the attack will fail if we know that the target host is not Intel-based.

5 Analyzers

This section models security devices used to secure networks, which we call *analyzers*. An analyzer is declared using unary predicate $analyzer(A)$. Any analyzer triggers *messages*, which are modelled in the next section.

We will distinguish three types of analyzers: intrusion detection systems, vulnerability scanners and firewalls. These tools inherit the characteristics of the *analyzer* concept, that is to say they trigger messages. Other components are either a specialization of those already modelled (e.g. an antivirus tool can be modelled as a knowledge and host-based IDS) or require an extension of the model that we leave for future work.

5.1 Intrusion Detection Systems

IDS detect, identify and generate alerts about active attacks in real time. Predicate $ids(A)$ qualifies an analyzer A as an IDS. Thus, an analyzer instance a that is an IDS is modelled in a fact base as both $analyzer(a)$ and $ids(a)$.

Following Debar et al.'s taxonomy [20], IDS can be classified according to their detection method (anomaly- or knowledge-based) and to their data source (network-based, host-based or application-based). In the remainder of this section, we describe how these characteristics fit into our formal framework. Notably, we discuss two concepts that are relevant for reasoning about alerts, namely the *topological visibility* and the *functional visibility* of an IDS. The former is related to the capability for an IDS to detect an intrusive activity ac-

cording to its data source, and more specifically to its location in the network. The latter refers to the capability of an IDS to detect an intrusive activity according to its detection method (e.g. the set of signatures activated for a misuse IDS).

Topological visibility and functional capabilities may also be used by administrators to identify the sweet spots and blind spots of a set of IDS [19] and anticipate those attacks that may not be detected.

5.1.1 Host-based IDS

Host-based IDS are characterized by means of the unary predicate $hids(A)$, where A is an IDS. As HIDS monitor the activity of the system they are installed on, modeling their topological visibility is straightforward: $monitors(A, H)$ means that the HIDS A monitors node H.

5.1.2 Application-based IDS

Similarly, application-based IDS are characterized by means of the unary predicate $aids(A)$, where A is an IDS. An application-based IDS monitors a specific software product running on a host. Therefore, we model the topological visibility of an application-based IDS by means of the ternary predicate $monitors(A, H, S)$, where A is an application-based IDS and S is a product hosted by H.

5.1.3 Network-based IDS

Generally speaking, network-based IDS (NIDS) analyze network packets to detect attacks. We may distinguish two types of network-based IDS, the classical ones and those called Intrusion *Prevention* Systems (IPS). This distinction is necessary because it impacts the way topological visibility is modelled. Classical NIDS passively capture network packets by means of a special network device, called a tap, whose purpose is to mirror the traffic that flows between two gateways. They are characterized by means of the predicate $nids(A)$, where A is an IDS. IPS are positioned inline and thus are able to block intrusive traffic before it reaches its target (hence their name). These IDS are characterized by means of predicate $ips(A)$, where A is an IDS.

Modeling the topological visibility of a NIDS is more challenging than HIDS and AIDS because the ability of an NIDS to detect an attack is not limited to a single host; it depends on the position of the sensor in the network and the path followed by a particular packet.

An IPS can be seen as a gateway with traffic filtering capabilities. Thus, the topological visibility of an IPS is modelled by means of the predicate $monitors(A, H_G)$, where A is an IPS and H_G is a gateway.

An IPS A can analyze an intrusive packet whose source host is H_S and a destination host H_D if the route followed by the packet includes the gateway monitored by A. This is modelled by predicate $can\_detect(A, H_S, H_D)$:

$$\begin{aligned} can\_detect(A, H_S, H_D) \quad \leftarrow \quad & ips(A) \wedge route(H_S, H_D, L) \wedge \\ & monitors(A, H_G) \wedge H_G \in L \end{aligned}$$

As mentioned earlier, a classical NIDS uses a tap to listen to the traffic on network link. Therefore, we model their topological visibility by means of the binary predicate $monitors(A, H_{G_1}, H_{G_2})$, where A is a NIDS, H_{G_1} and H_{G_2} are gateways. As for IPS, we can define predicate *can_detect* as follows:

$$\begin{aligned} can\_detect(A, H_S, H_D) \quad \leftarrow \quad & nids(A) \wedge route(H_S, H_D, L) \wedge \\ & monitors(A, H_{G_i}, H_{G_j}) \wedge [H_{G_i}, H_{G_j}] \subset L \end{aligned}$$

Here, $[H_{G_i}, H_{G_j}] \subset L$ holds if list L contains two adjacent elements H_{G_i}, H_{G_j}.

The notion of topological visibility for a NIDS should be used with caution. Let us consider for example a situation when an alert is triggered by an HIDS which has detected a local attack launched by a remote attacker who previously gained access on the host. This attack will most likely not be detected by any NIDS, even though the network session between the attacker and the victim host passes through a link or gateway monitored by an NIDS, because local attacks rarely have visible side effects on the network. This kind of situation may be taken into account by considering the requirements of the exploited vulnerability (c.f. $requires(V, \texttt{remote})$, page 11).

5.1.4 Knowledge-based IDS

Knowledge-based IDS (KB-IDS) are characterized by means of predicate $kbids(A)$, where A is an IDS. Knowledge-based IDS rely on signatures in order to detect attacks. Predicate $signature(Sig)$ states that Sig is a signature, which can be connected to a KB-IDS sensor by means of predicate $active(A, Sig)$, which means that signature Sig is active on KB-IDS A.

One may notice that not only does the concept of signature apply to KB-IDS, but also to explicit alarm correlation systems that use scenarios to detect attack sequences.

In order to model the functional visibility of a KB-IDS, we first introduce predicate $detects(K, Sig)$, which states that a signature Sig should detect an attack class K. A KB-IDS A is thus supposed to detect an attack class K if a signature Sig is active and designed to detect an attack class K' that is a superclass of K:

$$\begin{aligned} func\_vis(A, K) \leftarrow \quad & kbids(A) \wedge active(A, Sig) \wedge \\ & detects(Sig, K') \wedge attacksubclass(K, K') \end{aligned}$$

As the keywords used to describe attack classes are built from the attack descriptions found in attack signature databases, it is possible to automatically collect the facts *detects*; collecting *active* facts simply consists in parsing the configuration of a KB-IDS.

5.1.5 Anomaly-based IDS

Anomaly-based IDS are characterized by means of predicate $abids(A)$, where A is an IDS. Contrary to KB-IDS, the functional visibility of an anomaly-based IDS is modelled by a direct relationship with attack keywords: $detects(A, K)$. Thus, the above $func\_vis$ predicate for anomaly-based IDS is simply defined as follows:

$$\begin{aligned} func\_vis(A, K) \leftarrow \quad & abids(A) \wedge detects(A, K') \wedge \\ & attacksubclass(K, K') \end{aligned}$$

This definition suggests that an anomaly-based IDS that detects an attack class K is capable of detecting any attack subclass K' such that $attacksubclass(K', K)$ holds. If this is not the case in reality, then the functional detection capabilities of the IDS must be refined by asserting multiple $detects(A, K_i)$ facts for specialized attack classes K_i. For example, one may consider that `code_injection` is an attack class, whose direct subclasses are {`buffer_overflow`, `sql_injection`, `integer_overflow`}. If an anomaly IDS A can detect any code injection attack except SQL injection, then the functional capabilities will be modelled as

$$\begin{aligned} & detects(A, attackclass(\texttt{buffer\_overflow})) \wedge \\ & detects(A, attackclass(\texttt{integer\_overflow})) \end{aligned}$$

5.2 Vulnerability Scanners

A vulnerability scanner is an analyzer, whose objective is to spot latent vulnerabilities in a network and provide contextual information about the monitored

information system.

Analyzers are qualified as vulnerability scanners by means of predicate $scanner(A)$. Similar to IDS, we define the topological and functional visibility of vulnerability scanners, which are a direct translation of their configuration: the assertion $monitors(A, V, H)$ means that vulnerability scanner A is configured to perform a scan of vulnerability V on node H on a regular basis.

5.3 Firewalls

A firewall is an analyzer which can be modelled in our framework as gateway with filtering capabilities. Thus, we introduce the following predicates: $deny(H_G, A_S, P_S, A_D, P_D)$, which states that gateway H_G denies incoming connections from source address A_S with port P_S to target address A_D on port P_D.

Contrary to an IPS, a firewall does not have deep packet inspection capabilities and thus is not associated with signatures and attack classes, which is why firewalls are not modelled as IPS in M4D4.

Modeling network filtering in a network allows one to diagnose situations where a NIDS A_1 triggers an alert related to an attack, whereas another NIDS A_2 located on the attack path does not. The absence of alert may come from a firewall located in between A_1 and A_2 that has blocked the traffic. Depending on the result of the outcome of this verification, other hypothesis can be drawn to establish why A_2 did not react (e.g., A_2 is not functionally capable or not functioning any more).

6 Events and Alerts

M4D4 distinguishes two kinds of events, the *raw events* and *messages*, which are interpretation thereof. Both of these concepts inherit the characteristics of the *event* concept: an event $event(E, T)$ is an item of time-stamped information, where E is the event identifier and T is the time of occurrence. This definition complies with the one proposed by the IDWG [9] [21], where an event is defined as a low level entity (TCP packet, system call, syslog entry, for example) upon which an analysis is performed by a security device. An alert is a message from an analyzer signaling that one or more events of interest have been detected.

[9] IETF's Intrusion Detection Working Group

The remainder of this section models the various types of events identified and shows how they fit in the model.

6.1 Raw Events

A raw event is the manifestation of some activity at a data source level. The format and content of raw events thus depends on a particular data source. Examples of raw events are IP packets, TCP sessions or HTTP requests found in web server audit log files. We do not enumerate here those raw events that are modelled in M4D4. Rather, we explain how raw events types are integrated in the model.

Each raw event type is defined by inheritance of the $rawevent(E)$ predicate, which is itself a specialization of the event type. The inheritance relationship means that if a raw event $rawevent(E)$ is defined in the knowledge base, then there exists T such that $event(E, T)$ is also defined. Adding a new raw event type merely consists in defining one n-ary predicate to give the raw event a type, and as many n-ary predicates as the number of attributes that have a 1-to-many relationship with the event type.

For example, the IP packet type is defined by means of the predicate $ippacket(E, A_S, A_D, P, ...)$, where E is the raw event identifier and $A_S, A_D, P, ...$ respectively denote the source IP, the destination address, protocol number, and other header fields of the IP packet. Other raw event types are defined the same way.

In order to model the aggregative nature of raw events, we introduce the $includes(E_1, E_2)$ predicate, which means that event E_1 includes event E_2. This predicate can be used for instance to encapsulate an HTTP request raw event within HTTP log entry raw event.

6.2 Messages

A message is an interpretation of one or more raw events, which is given by an analyzer. As for raw events, we distinguish several types of messages depending on the type of analyzer.

A message is instantiated in the fact database with predicate $message(E, A)$, where E is the corresponding event identifier and A an analyzer. As for raw events, the inheritance relationship yields $\forall E, A, message(E, A) \rightarrow \exists T, event(E, T)$.

Messages are generally accompanied with raw events that act as evidence for security operators to conduct additional investigations if required. Raw events are connected to messages by means of predicate $evidence(E_M, E_R)$, where E_M is a message and E_R is a raw event.

Contrary to raw events, interpretations can be incomplete or incorrect with respect to the reality. For example, an alert stating that a host is under attack without assessing the success or failure of the attack is an example of incomplete interpretation. If this alert turns out to be a false positive, e.g. the presumed attacker actually is an administratively enabled vulnerability scanner, then it is also an incorrect interpretation. One of the objectives of correlation is to provide security operators with diagnoses that are as close to the reality as possible, by reasoning on complementary interpretations provided by various analyzers.

6.2.1 Alerts

Alerts are a kind of message triggered by an IDS about the occurrence of an attack instance, defined with predicate $alert(E, AI)$, where E is a message and AI an attack instance.

The objective of the predicate $attack(AI, T)$ is to model the features of an attack instance detected by an IDS. AI is the identifier of the attack and T its occurrence date, which is different from the time-stamp of the alert that reports the attack. Compared to the IDMEF data format [21], an alert's timestamp corresponds to `AnalyzerTime` field and an attack instance's timestamp corresponds to a `DetectTime` field.

The predicates $attacktype(AI, K)$, $attackorigin(AI, AI_O)$ and $attacktarget(A, AI_T)$ respectively connect an attack instance AI with an attack class K (see Section 4.2), an origin identifier AI_O and a target identifier AI_T.

The $origin(AI_O, F)$ predicate is intended to combine the various forms F of a single entity AI_O acting as an attacker. Similarly, $target(AI_T, F)$ provides the victim F of target AI_T. For instance, the IP address and a TCP port of a target entity will share the same target id as follows:

$$attack(a, time) \wedge attacktarget(a, t) \wedge$$
$$target(t, ipaddress(\texttt{'192.168.10.15'})) \wedge target(t, tcpport(\texttt{80}))$$

These facts mean that the attack a occurred at date $time$ and targets a host whose IP is 192.168.10.15, on port 80 (which does not mean that the host really exists and has a service listening on port 80).

A single attack instance may have many targets and origins. The various forms of origin and target attributes correspond to those identified in the IDMEF standard (source and target node, user, service), so we do not enumerate them here.

Attacks are also associated with an impact estimation, which reflects the risk of an attack instance, modelled with predicate $impact(AI, AI_S, AI_C, AI_T)$, where AI is an attack instance and AI_S, AI_C and AI_T respectively denote the severity, completion and type of an attack, as defined in the IDMEF standard [21].

Alerts also have an assessment attribute, which provides operators with an estimation of the confidence of the diagnosis of the analyzer. The confidence attribute differs from impact attribute of attacks: the impact is an absolute evaluation of an attack severity, whereas the confidence is an estimation relative to the analyzer of the likelihood that the observed activity indeed is the attack. Predicate $confidence(E, C)$ specifies the confidence C (a string that belongs to an enumerated type) of an alert E.

Alarm correlation systems combine alerts provided by third-party IDS and other observations provided by security tools (e.g., vulnerability reports) in order to enhance the semantics of alerts and/or reduce the overall number of alerts. An alarm correlation system is thus a special analyzer that acts as a message consumer and an alert provider. To model this, we add the predicate $subsumes(E_A, E_M)$, where E_A is an alert created by a correlation system and E_M is a message that took part in the diagnosis of the correlation system. Of course, several messages may be subsumed by a single alert.

The structure of an alert in this model slightly differs from the structure defined by IDMEF. However, from the content point of view, it is possible to convert an IDMEF alert into a conjunction of M4D4 predicates.

6.2.2 Vulnerability Reports

Vulnerability scanners generate specific messages called scan reports, that assert the presence of a vulnerability on a given host. Scan reports are modelled with predicate $report(E, VI)$, where E is a message identifier and VI is a vulnerability *instance*. A vulnerability instance is a fact $vulninstance(VI, H, V)$, where H is a node and V is a vulnerability, as defined in previous sections.

6.2.3 Firewall Logs

A firewall log is a message emitted by a firewall about a connection that has been blocked. A firewall log is simply modelled with predicate $fwlog(E)$. The

features of the blocked connection are asserted by means of a TCP/IP raw event E_R connected to E by means of $evidence(E, E_R)$.

7 Alert Correlation Use Case

Although our objective is not to design an alert correlation technique, we use a sample scenario to illustrate how M4D4 can be used for correlation. This example illustrates the interactions between the components of an intrusion detection platform, the required data and the reasoning that is performed in order to assess the severity of an alert. In the rest of this section, for better readability, some atom values are denoted by symbols instead of real values. This is the case for object identifiers and timestamps. For example, the term $vulnerability(vuln)$ models a vulnerability whose identifier is $vuln$.

7.1 Knowledge Modeling

7.1.1 Context

In this scenario, we consider the network of a university, divided into several subnets, notably the teachers' subnet, $network(profnet)$ and the students' subnet $network(studnet)$, which are both part of the internal network, $network(intnet)$. Each subnet has an associated gateway, $gateway(profgate)$, $gateway(studgate)$ and $gateway(intgate)$.

Among the routes that are defined in the network, we know that the connections from the students' network to the teachers' network are routed through the internal gateway: $routes(studgate, intgate, profnet)$.

We further consider two specific hosts, $node(profhost)$ and $node(studhost)$, which respectively correspond to a teacher's machine and a student's machine, located in their respective networks. Their IP address is $nodeaddress(profhost, '192.168.2.38')$ and $nodeaddress(studhost, '192.168.1.12')$.

A network mapping system has recognized that the operating system of the teacher's machine is Windows XP, but failed to identify the exact version. This is recorded in the knowledge base as the fact $hosts(profhost, software('WindowsXP', \_, os, \_)$. No information about the student's machine configuration is available.

7.1.2 Attacks and vulnerabilities

Blaster [10] is a worm which exploits a vulnerability that affects several versions of the Windows operating system. The corresponding vulnerability is referenced by CVE under serial 2003-0352. More precisely, we know that Blaster uses a buffer overflow to execute code on the victims. This knowledge is available in vulnerability databases and is encoded in M4D4 with the following facts:

$$vulnerability(blastervuln)$$

$$refersto(blastervuln,'CVE','2003-0352')$$

$$affects(blastervuln, blasterconf)$$

$$takespartin(software('Windows2000', \_, os, \_), blasterconf)$$

$$takespartin(software('WindowsXP', \_, os, \_), blasterconf)$$

$$exploits(attackclass('blaster'), blastervuln)$$

$$inherits(attackclass('blaster'), attackclass('worm'))$$

$$inherits(attackclass('blaster'), attackclass('bufferoverflow'))$$

7.1.3 Analyzers

We consider that a Snort sensor is monitoring the link between the student network and the internal network: $monitors(snort, studgate, intgate)$. Snort sensors are network-based, ($nids(snort)$) and knowledge-based ($kbids(snort)$).

According to the Snort rules documentation, we know that the Snort signature id 2351 is designed to detect Blaster worm propagations [11] : $signature('s2351')$, $detects('s2351', attackclass('blaster'))$. Actually, the attack class recognized by this signature is labeled "NETBIOS DCERPC ISystemActivator path overflow attempt little endian unicode" by Snort. Thus, we should first create an *attackclass* term with this label, and link it to the $attackclass('blaster')$ by means of an *inherits* relationship. However, to make it simpler, we will consider that there is a direct relationship between the signature and $attackclass('blaster')$.

By analyzing the sensor's settings, we also know that this signature is activated on the Snort sensor: $active(snort, s2351)$.

Next, we assume that the teacher's host is protected by an antivirus tool, which

[10] `http://www.cert.org/advisories/CA-2003-20.html`
[11] `http://www.snort.org/pub-bin/sigs.cgi?sid=2351`

is comparable to an anomaly-based intrusion detection system, i.e.: $hids(av)$, $abids(av)$, $monitors(av, profhost)$.

This antivirus is able to detect buffer overflows attempts. We encode this knowledge as $detects(av, attackclass(bufferoverflow))$.

Lastly, the internal gateway acts as a firewall $firewall(intgate)$, which restricts allowed connections from the students' network to the teachers' network.

7.1.4 Events and alerts

We now assume that the student host has been infected by the Blaster worm and starts attacking its neighborhood. The following terms encode a Blaster attack instance against the teacher's host:

$$attack(blasterattack, d_b)$$

$$attacktype(blasterattack, attackclass(blaster))$$

$$origin(o_1, ipaddress('192.168.1.12')) \quad attackorigin(blasterattack, o_1)$$

$$target(t_1, ipaddress('192.168.2.38')) \quad attacktarget(blasterattack, t_1)$$

Here, *blasterattack* is the attack instance identifier, d_b is a symbol that represents the date of the attack (e.g. "Thu Apr 17 13:46:16"), o_1 and t_1 are object identifiers which represent the origin and target of the attack instance. In this case, we consider that there is only one target and origin. Should there be several targets or origins involved in the attack, other identifiers t_i would be used to distinguish them.

This attack instance is detected by Snort and reported in an alert:

$$event(snortalert, d_s)$$

$$message(snortalert, snort)$$

$$alert(snortalert, blasterattack)$$

where d_s is the date of attack detection by Snort.

7.2 Queries

One of the first question that arises in this situation is whether the attack is successful or not. Several methods to assess the success of the attack are

available. One may object that the simplest way for a correlation system to assess the success of a worm attack is to observe identical attacks from a host that has previously been hit by the worm. However, this characteristic is specific to worms only. Moreover, the modus operandi used by a worm to infect a host can be reused by other attacks which are *not* worms (and will still trigger identical alerts by classical IDSes).

Therefore, we will proceed differently to assess the success of the attack. Our first question consists in evaluating whether the target host is vulnerable or not. We assume here that no vulnerability report is available about $profhost$. On the other hand, some knowledge about the configuration of the victim and the characteristics of the Blaster worm are available to assess the success of the attack. From the content of the alert, the following request would fail:

$$nodeaddress(H,'192.168.1.12') \wedge \text{exploits(attackclass('blaster'),V)} \wedge$$
$$not\_vulnerable(H,V)$$

In other words, it is not possible to find a host whose IP address is 192.168.1.12 that is not vulnerable to a vulnerability exploited by the Blaster attack. This might indicate that the attack is successful. In order to confirm this hypothesis, the second step in the reasoning process consists in checking if any other analyzer could have detected the same attack. This corresponds to the following query:

$$analyzer(A) \wedge$$
$$(can\_detect(A,'192.168.1.12','192.168.2.38') \vee monitors(A, profhost)) \wedge$$
$$func\_vis(A, attackclass('blaster'))$$

By definition of *can_see* (c.f. page 17) and $func\_vis$ (c.f. page 18), this query will succeed with $A = av$, because av is an anomaly-host-based IDS which monitors $profhost$ and whose functional visibility allows it to detect any attack class that is a subclass of $'bufferoverflow'$. The query would also succeed with $A = snort$, since snort monitors flows between the origin and target host. However, this answer is discarded since the analyzed alert comes from the snort sensor, the source of the alert to be confirmed.

The next step is to check if av did trigger an alert that is *similar* to the one triggered by *snort*:

$$alert(E, av) \wedge event(E, T) \wedge |T - t_s| < \epsilon$$
$$attack(E, AI) \wedge attacktype(AI, K) \wedge attacksubclass('blaster', K))$$

This request concerns alerts triggered by the av sensor whose occurrence date

is *close* to the snort alert date t_s (ϵ is a tolerance interval) and which report an attack that is a superclass of *blaster*.

Let us assume that no such alert exists. Two explanations can justify the absence of alert from *av*: either the attack instance never reached the target, or *av* did not manage to detect the attack. In order to check the first hypothesis, it is possible to check if a firewall is on the route between the origin and the target of the attack. The request $route(profhost, studhost, P)$ will succeed with $P = [studgate, intgate, profgate]$. We know that $firewall(intgate)$ holds, so we can assume that the attack failed because it has been blocked by the firewall. This assumption can further be confirmed by analyzing the firewall logs.

Even though this attack failed, it would be a mistake to consider that it is benign. Indeed, this attack is part of a worm propagation that comes from the inside of the administrative domain, which is serious incident. This situation can be encoded as an operator's preference rule, which states that any alert concerning an attack class that is a worm ($attacksubclass('worm', K)$) and whose origin is inside the monitored network should deserve high severity.

8 Overview of an M4D4 Framework Implementation

M4D4 has a practical ground, and a prototype implementation is currently being developed, the core of which is based on a Prolog engine. The architecture of the system is sketched in Figure 2. In this figure, continuous arrows denote information flows (events, alerts, context observations) sent by tools and broken arrows denote request/response interactions between components.

M4D4 is integrated in a security event management framework. Within this framework, analyzers detect attacks and send alerts to an alert manager, whose role is to store the received events in a database, and forward them to correlation processes for further analysis. The correlation processes may in turn produce higher level alerts back to the alert manager. Our prototype implementation will be based on the Prelude [12] framework for this purpose.

The correlation processes send queries to the M4D4 framework in the form of a logical formula, similar to those presented in the previous section. These queries may deal with contextual information, events, alerts, sensor configuration and capabilities, or attack and vulnerabilities characteristics.

Upon reception of a request, the M4D4 Prolog engine will try to instantiate the variables of the queries by unifying them with the facts available in the knowledge bases. As we can see in Figure 2, the knowledge may be distributed

[12] `http://www.prelude-ids.org`

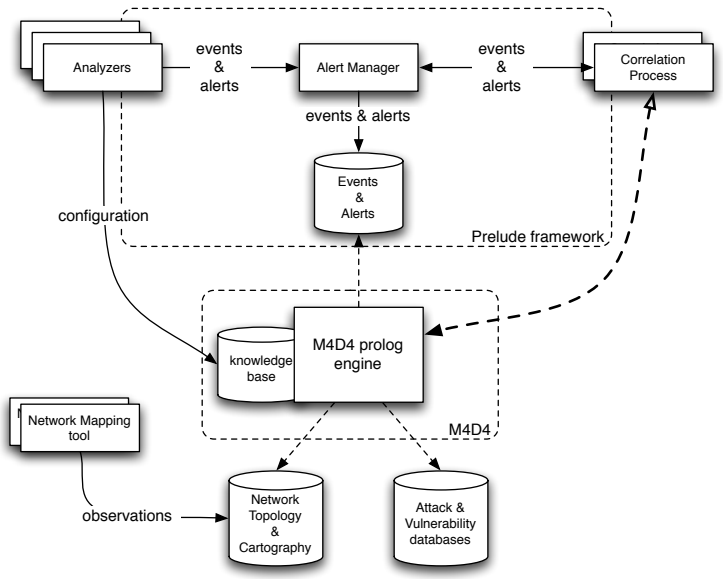

Fig. 2. Architecture of an M4D4 implementation

throughout several concrete databases, the content of which is filled by external tools. Context observations provided by network mapping systems, such as ettercap, nmap and Netexpose DNA [13]; vulnerabilities characteristics come from the Open Source Vulnerability Database (OSVDB), the National Vulnerability Database (NVD) or OVAL.

M4D4 can interact with external databases which have their own relational schema to represent data. In this case, M4D4 translates the received query into the corresponding query to retrieve the facts. When it is possible, the tools can also be modified to supply their observations directly in M4D4's own knowledge base.

Last, M4D4 predicates can also be implemented as actions that are dynamically executed to answer a request. For example, the predicate $resolve(A, D)$, which associates a hostname D to an address A, can be implemented as a DNS request that is dynamically executed when it is requested by a correlation system.

9 Related Work

Data modeling in intrusion detection has been the subject of numerous papers in the intrusion detection field. First attempts have consisted in designing taxonomies for attack classes [22]. Taxonomies are indeed useful to identify

[13] `http://www.netexpose.com`

and classify the characteristics of attacks and we take advantage of these contribution in our work, but their scope is limited to attack classes only, i.e. they do not propose an extensive model of the context, analyzers and events.

Raskin et al [23] advocate the use of ontologies for information security. Contrary to ontologies, taxonomies lack the necessary and sufficient constructs needed to reason over instances of the modelled domain. Following Kemmerer and Vigna's suggestion that additional effort is needed to provide a common ontology that lets IDS sensors agree on what they observe [24], Undercoffer et al. proposed a target centric ontology for intrusion detection in [25,26]. Authors argue that an ontology should only model properties that are observable and measurable by the target of an attack. We argue that it is insufficient, for many alerts (especially false positives) triggered by intrusion detection systems involve actors which are inside the monitored information system and whose properties are consequently also observable.

First attempts to model topology information in intrusion detection came from Vigna and Kemmerer [27], who proposed a Network Fact Base, inspired by Vigna's formal model of TCP/IP networks. M2D2's relational network model was inspired by this model ; although the topology part of M4D4 slightly differs from M2D2's, the main ideas remain the same.

Porras et al. [28] proposed an alarm correlation scheme that takes into account events produced by spatially distributed heterogeneous information security devices, such as firewalls, intrusion detection systems, and antivirus tools. However, the authors do not propose a formal data model for the configuration of analyzers and their interactions with other concepts.

The Intrusion Reference Model (IRM) proposed by Goldman et al [19] is probably the closest work to ours. IRM uses the CLASSIC object-oriented database system to model a site security policy, network topology, product configuration, and intrusion events. Some of the contextual concepts of M4D4 are similar to those found in NERD, notably hosts, products, and their relationships, as well as the attack classes graph (called event dictionary in the context of IRM). The primary goal of the authors in [19] is not so much to propose a knowledge representation model, as to design an alarm correlation architecture, whose objective is to aggregate multiple intrusion reports and try to match them with the protected site's security goals. For this purpose, authors sketch the aforementioned concepts and relationships. To our best knowledge, the contents of IRM and NERD has not been published in greater details. Moreover, much of the information represented in M4D4 is not evoked in [19], such as the analyzers capabilities.

The IDMEF (Intrusion Detection Message Exchange Format) is the IETF standard aiming at defining a common alert format for IDS to exchange their

observations. IDMEF is an XML-based representation of alert features, like the attack type, source, destination and time; it is not intended to provide other information, like network mapping data and vulnerability reports. From this point of view, the scope of M4D4 is larger than IDMEF. Nonetheless, the alert concept in M4D4 is compatible with IDMEF and IDMEF-compliant alerts can be received and processed within the M4D4 framework without loss of semantical information. IDMEF is not intended to define attack classes either. As a consequence, two heterogeneous IDSes conforming to the IDMEF standard will trigger two syntactically similar alerts, but whose semantic contents might not be comparable.

Among the models proposed to represent alerts, CISL (Common Intrusion Specification Language) [29,30] is probably one of the most comprehensive language with regard to its linguistic features. CISL proposed a syntax based on S-expressions, inspired by the LISP language, and a rich terminology to describe events and semantics. The problem of query and answer formulations has been addressed in CISL by Ning et al [31,32]. Unfortunately, despite its expressiveness, CISL fell into disuse and was replaced with IDMEF.

As related work, we may also quote the attack description languages proposed by Cuppens [33,15] and Ning [16]. These languages express two attack classes characteristics called preconditions and postconditions. These characteristics serve a specific purpose, which basically consists in building intrusion scenarios from alerts, by unifying prerequisites of attack instances with consequences of other attack instances. In other words, these contributions do not focus so much on the *modeling* of such information, as to exploit them for correlation. From this point of view, our work is complementary to theirs.

10 Conclusion & Future work

In this paper, we have presented the M4D4 data model. M4D4 formalizes the concepts and relationships that are required to reason about alerts and observations triggered by security systems distributed across the network.

As any model, M4D4 is an abstraction of reality. M4D4 is a theoretical construct that represents the security of a network, with a set of variables and a set of logical relationships between them. It is constructed to enable reasoning within an idealized logical framework, which means that the model makes assumptions that are incomplete in some detail. Such assumptions may be justified on the grounds that they simplify the model while, at the same time, allowing the production of acceptable accurate solutions.

Based on a realistic attack scenario, we have illustrated how the logical frame-

work can be used in practice to reason on alerts.

Future work can be split in two main tracks. First, from a practical point of view, our objective is to implement M4D4 in an operational tool. Our work lies within the scope of a project whose objective is to design an alert management and correlation platform that is capable of collecting complementary observations coming from various data sources and reasoning about these observations, in order to bring security operators with an enhanced diagnosis of the security events at stake. We are planning to implement the model with a deductive and distributed database that is central to the platform. We will also study to what extent the existing IDMEF standard can be used to exchange observations and propose modifications to achieve this. Indeed, the primary purpose of IDMEF is to exchange alerts between IDS; diagnosis requires, e.g., to exchange cartographic information and vulnerability assessment information, the structure of which does not currently fit within IDMEF.

Second, from a more theoretical point of view, our future work will consist in applying the formalism of *description logics* to the field of intrusion detection. Description logics [34] are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way. Predicate logics subsume most description logics, but the latter offer more expressive constructs to describe a domain than the former. More specifically, various description logics offer strong theoretical basis to design reasoning schemes capable of handling temporality, uncertainty, incomplete information, and modality, which we believe are quite relevant in the intrusion detection field, where context observations and alerts are inherently uncertain, and where the dynamics of the monitored environment are crucial. As a matter of fact, applying description logics to intrusion detection has been investigated by Goldman et al [19]. More recently, Zakeri et al [35] also investigated the use of description logic, but their model is limited to the description of vulnerabilities within TCP/IP networks. Our future work will probably be inspired by some of the ideas expressed in these papers.

Last, as this model does not contain all relevant concepts, future work will also consist in enriching it with missing concepts in order to reach comprehensiveness. Notably, *global* context information (i.e. external to the monitored information system) provided by tools like honeypots are of particular interest to us. Correlating this information with the one already defined in M4D4 will be valuable to enhance the semantics of alerts, for instance by comparing observed attack trends with attacks actually observed in the monitored information system.

Acknowledgments

This work is part of the ACES [14] project, partly funded by the French National Research Agency (ANR/Telecom program, formerly known as RNRT) under contract number ANR-05-RNRT-00101. We would like to thank all the members of the project for fruitful discussions on the model content.

References

[1] B. Morin, L. Mé, H. Debar, M. Ducassé, M2D2: A Formal Data Model for IDS Alert Correlation, in: A. Wespi, G. Vigna, L. Deri (Eds.), Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002), Vol. 2516 of Lecture Notes in Computer Science, Springer, 2002, pp. 115–127.

[2] K. Julisch, Clustering intrusion detection alarms to support root cause analysis, ACM Transactions on Information and System Security 6 (4).

[3] F. Valeur, G. Vigna, C. Kruegel, R. A. Kemmerer, A comprehensive approach to intrusion detection alert correlation, IEEE Transactions on Dependable and Secure Computing 1 (3) (2004) 146–169.

[4] A. Valdes, K. Skinner, Probabilistic alert correlation, in: W. Lee, L. Mé, A. Wespi (Eds.), Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001), no. 2212 in LNCS, 2001, pp. 54–68.

[5] O. M. Dain, R. K. Cunningham, Fusing heterogeneous alert streams into scenarios, in: Proceedings of the Workshop on Data Mining for Security Applications, 8th ACM Conference on Computer and Communication Security, 2001.

[6] O. M. Dain, R. K. Cunningham, Building scenarios from a heterogeneous alert stream, in: Proceedings of the 2001 IEEE Workshop on Information Assurance and Security, 2001.

[7] H. Debar, A. Wespi, Aggregation and correlation of intrusion-detection alerts, in: W. Lee, L. Mé, A. Wespi (Eds.), Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001), no. 2212 in LNCS, 2001, pp. 85–103.

[8] S. Manganaris, M. Christensen, D. Zerkle, K. Hermiz, A data mining analysis of rtid alarms, Web proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection (RAID'99), http://www.raid-symposium.org/raid99 (September 1999).

[14] `http://www.rennes.supelec.fr/aces/`

[9] B. Morin, H. Debar, Conceptual analysis of intrusion alarms, in: F. Roli, S. Vitulano (Eds.), 13th International Conference on Image Analysis and Processing, Vol. 3617 of Lecture Notes in Computer Science, 2005, pp. 91–98, special session on Computer Security.

[10] B. Morin, H. Debar, Correlation of intrusion symptoms: An application of chronicles, in: G. Vigna, E. Jonsson, C. Krügel (Eds.), Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'2003), Vol. 2820 of Lecture Notes in Computer Science, Springer, 2003, pp. 94–112.

[11] E. Totel, B. Vivinis, L. Mé, A language driven intrusion detection system for event and alert correlation, in: Proceedings ot the 19th IFIP International Information Security Conference, Kluwer Academic, Toulouse, 2004, pp. 209–224.

[12] C. Michel, L. Mé, Adele: an attack description language for knowledge-based intrusion detection, in: Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001), 2001, pp. 353–365.

[13] F. Cuppens, R. Ortalo, LAMBDA: A Language to Model a Database for Detection of Attacks, in: H. Debar, L. Mé, S. F. Wu (Eds.), Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000), no. 1907 in LNCS, 2000, pp. 197–216.

[14] S. J. Templeton, K. Levitt, A requires/provides model for computer attacks, in: Proceedings of the 2000 New Security Paradigms Workshop (NSPW'00), 2000, pp. 31–38.

[15] F. Cuppens, Managing alerts in a multi-intrusion detection environment, in: Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001), 2001.

[16] P. Ning, D. Xu, C. G. Healey, R. S. Amant, Building attack scenarios through integration of complementary alert correlation methods, in: 11th Annual Network and Distributed System Security Symposium, 2004.

[17] Y. Thomas, B. Morin, H. Debar, Improving Security Management through Passive Network Observation, in: First International Conference on Availability, Reliability and Security (ARES), 2006.

[18] D. E. Mann, S. M. Christey, Towards a common enumeration of vulnerabilities, in: Proceedings of the 2nd Workshop on Research with Security Vulnerability Databases, 1999.

[19] R. P. Goldman, W. Heimerdinger, S. A. Harp, C. W. Geib, V. Thomas, Information modeling for intrusion report aggregation, in: Proceedings of the DARPA Information Survivability Conference and Exposition, 2001.

[20] H. Debar, M. Dacier, A. Wespi, A revised taxonomy for intrusion-detection systems, Tech. Rep. rz3176, IBM Zurich Research Laboratory (October 1999).

[21] H. Debar, D. Curry, RFC 4765, IETF (November 2006).

[22] J. D. Howard, T. A. Longstaff, A common language for computer security incidents, Tech. Rep. SAND98-8667, Sandia National Laboratories (October 1998).

[23] V. Raskin, C. F. Hempelmann, K. E. Triezenberg, S. Nirenburg, Ontology in information security: a useful theoretical foundation and methodological tool, in: Proceedings of the 2001 workshop on New security paradigms, ACM Press, 2001, pp. 53–59.

[24] R. Kemmerer, G. Vigna, Intrusion detection: A brief history and overview, IEEE Computer - Special publication on Security and Privacy (2002) 27–30.

[25] J. Undercoffer, A. Joshi, J. Pinkston, Modeling computer attacks: An ontology for intrusion detection, in: G. Vigna, E. Jonsson, C. Krügel (Eds.), Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'2003), Vol. 2820 of Lecture Notes in Computer Science, Springer, 2003, pp. 113–135.

[26] J. L. Undercoffer, A. Joshi, T. Finin, J. Pinkston, A target-centric ontology for intrusion detection, in: The 18th International Joint Conference on Artificial Intelligence, 2003.

[27] G. Vigna, R. A. Kemmerer, NetSTAT: A Network-based Intrusion Detection System, Journal of Computer Security 7 (1) (1999) 37–71.

[28] P. A. Porras, M. W. Fong, A. Valdes, A mission-impact-based approach to infosec alarm correlation., in: A. Wespi, G. Vigna, L. Deri (Eds.), Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002), Vol. 2516 of Lecture Notes in Computer Science, Springer, 2002, pp. 95–114.

[29] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, B. Tung, A Common Intrusion Specification Language (CISL), Specification draft, http://www.gidos.org/drafts/language.txt (March 2000).

[30] B. Tung, The Common Intrusion Specification Language: a Retrospective, in: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00), Vol. 2, 2000, pp. 36 – 45.

[31] P. Ning, X. S. Wang, S. Jajodia, Modeling requests among cooperating intrusion detection systems, in: Computer Communications, Vol. 23, 2000, pp. 1702–1716.

[32] P. Ning, X. S. Wang, S. Jajodia, A query facility for common intrusion detection framework, in: 23rd National Information Systems Security Conference, 2000, pp. 317–328.

[33] F. Cuppens, A. Miège, Alert correlation in a cooperative intrusion detection framework, in: Proceedings of the IEEE Symposium on Security and Privacy, 2002, pp. 202–215.

[34] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook, Cambridge University Press, 2003.

[35] R. Zakeri, R. Jalili, H. R. Shahriari, H. Abolhassani, Using description logics for network vulnerability analysis, in: International Conference on Networking, Systems, Mobile Communications and Learning Technologies, IEEE Computer Society, 2006.

Appendix

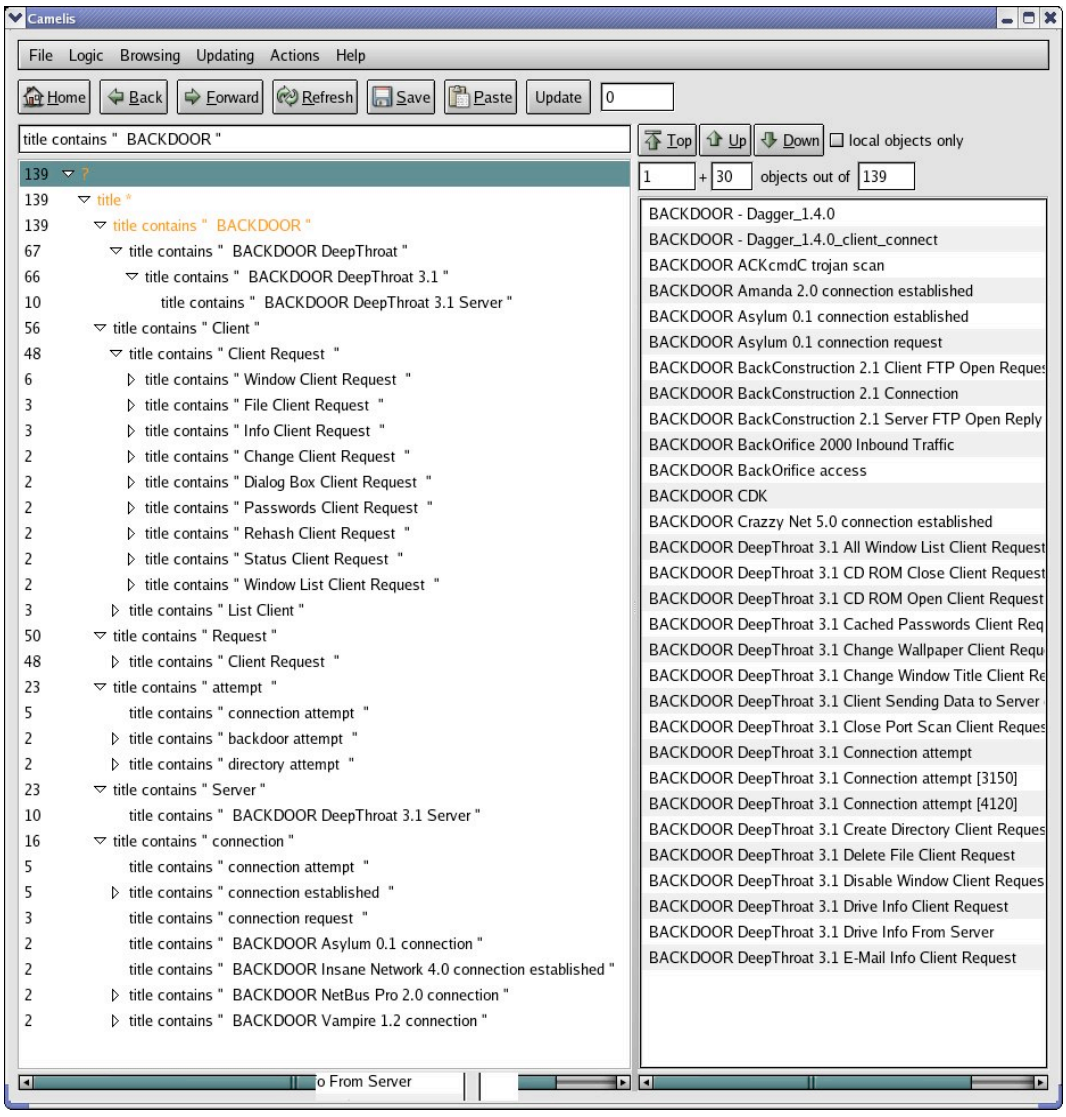

Fig. .1. Excerpt of attack keywords graph automatically extracted by text processing tools applied to the set of `msg:` features of Snort signature database.

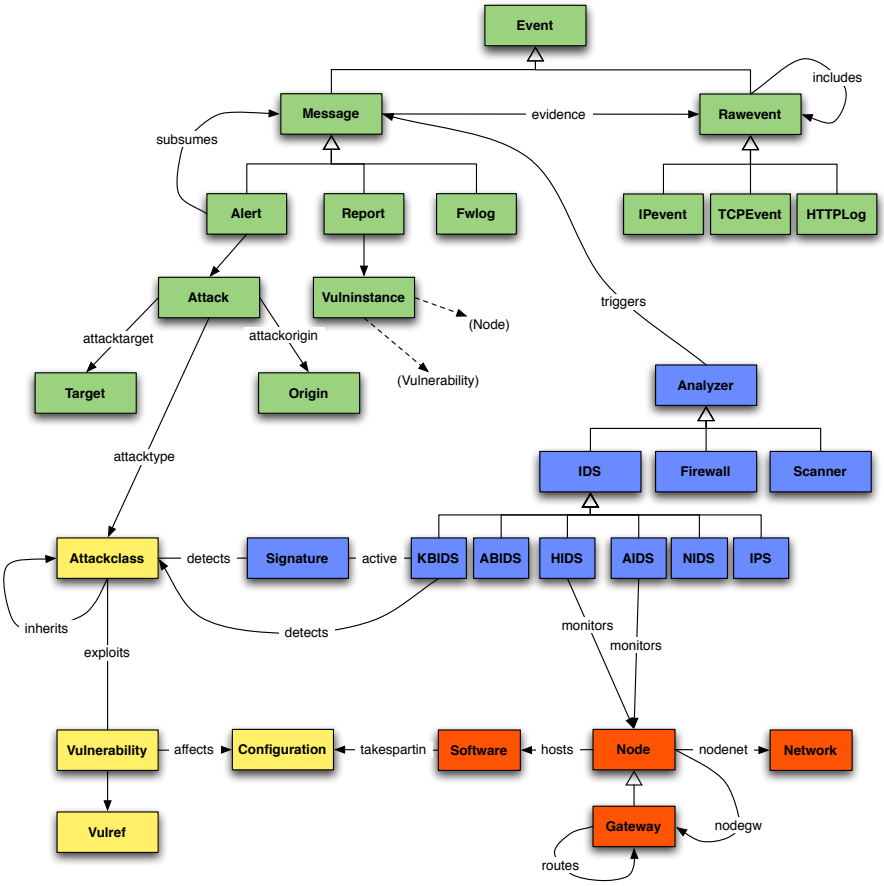

Fig. .2. Main concepts and relationships of M4D4

Predicate	Meaning
	Topology
$network(N)$	N is a network.
$netaddress(N, A_N)$	The address of network N is A_N.
$node(H)$	H is a network node.
$nodeaddress(H, A_H)$	The address of H is A_H.
$gateway(H)$	Node H is a gateway.
$nodenet(H, N)$	Node H belongs to network N.
$nodegw(H, H_G)$	H_G is one of H's next gateway.
$routes(H_{G_1}, H_{G_2}, N)$	H_{G_1} routes packets to network N through H_{G_2}
$path(H_{G_S}, H_{G_D}, L, N)$	L is a list of gateways that compose the path followed by a packet to be delivered to network N from a gateway H_{G_S} to a gateway H_{G_D}.
$route(H_S, H_D, L)$	L is one possible route between a source node H_S and a destination node H_D.
$nat(H_G, A_1, P_1, A_2, P_2)$	(A_1,P_1) is statically translated to (A_2,P_2) by H_G, where A_i and P_i are an address and a port number.
$nodename(H, SN)$	The hostname of node H is SN.
$resolve(A, DN)$	Address A's fqdn (fully qualified domain name) is DN.
	Cartography
$software(S_N, S_V, S_T, S_A)$	Defines a software product whose name, version, type, architecture are S_N, S_V, S_T, S_A.
$hosts(H, S)$	Node H hosts product S.
$process(S, U)$	Product S is executed under U's identity.
$exec(H, P)$	Node H executes process P.
$service(P, Q)$	Process P is a service which waits for connections on port Q.
$listens(H, Ser)$	Node H executes service Ser.

Table .1
Context predicates

Predicate	Meaning
Vulnerabilities	
$vulnerability(V)$	V is a vulnerability.
$affects(V, C)$	Vulnerability V affects configuration C.
$takespartin(S, C)$	Product S is involved in one vulnerable configuration C.
$refersto(V, O, V_N)$	Vulnerability V is named V_N by organization O.
$equiv(O_1, V_{N_1}, O_2, V_{N_2})$	Vulnerability names V_{N_1} (by O_1) and V_{N_2} (by O_2) refer to the same vulnerability.
$severity(V, Sev)$	Vulnerability V's severity equals Sev.
$requires(V, W)$	Vulnerability V's requirements is W.
$losstype(V, Con)$	Vulnerability V's loss type is Con.
$published(V, Date)$	Vulnerability V's publication date is $Date$.
Attack classes	
$attackclass(K)$	K is an attack class.
$inherits(K_1, K_2)$	K_1 is a direct subclass of K_2.
$attacksubclass(K_1, K_2)$	K_1 is subclass of K_2, transitively.
$exploits(K, V)$	Attack K exploits vulnerability V.

Table .2
Attacks and vulnerabilities predicates

Predicate	Meaning
$analyzer(A)$	A is an analyzer.
$ids(A)$	A is an IDS.
$hids(A)$	IDS A is a host-based.
$monitors(A, H)$	HIDS A monitors node H.
$monitors(A, H, S)$	Application IDS A monitors application S on node H.
$nids(A)$	IDS A is network-based.
$ips(A)$	IDS A is an intrusion-prevention system.
$monitors(A, H_G)$	IPS A is on gateway H_G.
$monitors(A, G_1, G_2)$	NIDS A monitors link G_1,G_2.
$can\_detect(A, H_S, H_D)$	According to its position in the network, IDS A might be able to analyze packets flowing from host H_S to host H_D.
$kbids(A)$	IDS A is knowledge-based.
$abids(A)$	IDS A is anomaly-based.
$signature(Sig)$	S is a KB-IDS signature.
$active(A, Sig)$	Signature S is active on KB-IDS A.
$detects(K, Sig)$	Signature S is intended to detect attack K.
$func\_vis(A, K)$	According to its settings, KB-IDS A should detect attacks that are subclasses of K.
$detects(A, K)$	Anomaly-based IDS A can detect attacks K and its subclasses.
$scanner(A)$	A is a vulnerability scanner.
$monitors(A, V, H)$	A checks H for presence of vulnerability V.
$deny(H_G, A_S, P_S, A_D, P_D)$	H_G denies incoming connections from address A_S, port P_S to address A_D, port P_D.

Table .3
Analyzers predicates

Predicate	Meaning
$event(E, T)$	E is an event, which occurred at T.
$rawevent(E)$	E is a raw event.
$ippacket(E, S, D, P, ...)$	E is an IP packet.
$includes(E_1, E_2)$	Event E_1 includes event E_2.
$message(E, A)$	E is a message sent by analyzer A.
$evidence(E_M, E_R)$	Message E_M is associated with raw event E_R.
$alert(E, AI)$	E is an alert which reports attack instance AI.
$attacktype(AI, K)$	Attack instance AI's attack class is K.
$attackorigin(AI, AI_O)$	Attack instance AI's origin is AI_O.
$attacktarget(AI, AI_T)$	Attack instance AI's target is AI_T.
$origin(AI_O, F)$	F is part of origin AI_O.
$target(AI_T, F)$	F is part of target AI_T.
$impact(AI, AI_S, AI_C, AI_T)$	AI_S, AI_C and AI_T respectively denote the severity, completion and type of a attack instance AI.
$subsumes(E_A, E_M)$	Alert E_A subsumes message E_M.
$report(E, VI)$	E is a report about a vulnerability instance VI.
$vulninstance(VI, H, V)$	Vulnerability instance states that host H is affected by V.
$fwlog(E)$	Message E is a firewall log.

Table .4
Events and alerts predicates