# A Meta Language for Threat Modeling and Attack Simulations

Pontus Johnson
KTH Royal Institute of Technology
Stockholm, Sweden
pontusj@kth.se

Robert Lagerström
KTH Royal Institute of Technology
Stockholm, Sweden
robertl@kth.se

Mathias Ekstedt
KTH Royal Institute of Technology
Stockholm, Sweden
mekstedt@kth.se

## ABSTRACT

Attack simulations may be used to assess the cyber security of systems. In such simulations, the steps taken by an attacker in order to compromise sensitive system assets are traced, and a time estimate may be computed from the initial step to the compromise of assets of interest. Attack graphs constitute a suitable formalism for the modeling of attack steps and their dependencies, allowing the subsequent simulation.

To avoid the costly proposition of building new attack graphs for each system of a given type, domain-specific attack languages may be used. These languages codify the generic attack logic of the considered domain, thus facilitating the modeling, or instantiation, of a specific system in the domain. Examples of possible cyber security domains suitable for domain-specific attack languages are generic types such as *cloud systems* or *embedded systems* but may also be highly specialized kinds, e.g. *Ubuntu installations*; the objects of interest as well as the attack logic will differ significantly between such domains.

In this paper, we present the Meta Attack Language (MAL), which may be used to design domain-specific attack languages such as the aforementioned. The MAL provides a formalism that allows the semi-automated generation as well as the efficient computation of very large attack graphs. We declare the formal background to MAL, define its syntax and semantics, exemplify its use with a small domain-specific language and instance model, and report on the computational performance.

## CCS CONCEPTS

• **Security and privacy → Security services**; **Systems security**; *Software and application security*; • **Software and its engineering → Domain specific languages**;

## KEYWORDS

Domain Specific Language, Cyber Security, Threat Modeling, Attack Graphs

**ACM Reference format:**
Pontus Johnson, Robert Lagerström, and Mathias Ekstedt. 2018. A Meta Language for Threat Modeling and Attack Simulations. In *Proceedings of International Conference on Availability, Reliability and Security, Hamburg, Germany, August 27–30, 2018 (ARES 2018),* 8 pages.


*ARES 2018, August 27–30, 2018, Hamburg, Germany*

ACM ISBN 978-1-4503-6448-5/18/08…$15.00
https://doi.org/10.1145/3230833.3232799

https://doi.org/10.1145/3230833.3232799

## 1 INTRODUCTION

Assessing the cyber security of computing systems, from embedded systems to corporate IT infrastructures is difficult. In order to identify vulnerabilities, one must understand a vast range of security-relevant features of the system, and then consider all potential attacks. This presents a number of challenges. Firstly, it is difficult to know what information about the system to collect, what exactly are all security properties? Secondly, it is oftentimes hard to actually collect that information. Thirdly, it is difficult to analyze the collected information in order to find all the security weaknesses that could be exploited by an attacker.

To support these challenging tasks, we have proposed the use of attack simulations based on system architecture models [7, 12, 15, 33]. In this scheme, a model of the system is subjected to simulated cyber attacks in order to identify the greatest weaknesses, which conceptually can be thought of as the execution of a great number of parallel virtual penetration tests. In the presence of such an attack simulation tool, the security assessor is relieved of the first and third challenge above, and may focus on the collection of the information about the system required for the simulations.

The attack simulations considered here are based on attack graphs, i.e. directed graphs representing the dependencies between the steps that may be performed by the attacker. For instance, an attacker might (i) compromise a host, (ii) find stored credentials, (iii) use the credentials to compromise a second host, etc. As will be described in greater detail below, the result of a simulation, in our case, is a probabilistic estimate of the time to compromise each attack step.

When addressing real enterprise systems, attack graphs normally become very large. Because many systems are similar, but few are identical, it is possible to ease this problem by separating the generic features of a domain (e.g. the threat of man-in-the-middle attacks in corporate IT infrastructures) from the specific configuration of a particular system (such as the network structure of a given organization). Such separation may be achieved by recording the generic features in a modeling language, which subsequently can be instantiated for any specific system. This technique is commonly employed in software design, as exemplified by the Unified Modeling Language [26] and particularly the Meta Object Facility [25] that provides a layered stack of modelling languages, with e.g. the so called platform-independent models and platform-specific models. For our case, we could envision an example of a generic corporate IT security language, which could be deployed by security engineers to generate specific attack graphs for the specific infrastructure of different organizations. Simulations on these generated attack graphs could then identify different security weaknesses

in the different organizations depending on their exact respective infrastructure configurations and properties.

This separation is convenient not only from the point of reuse, but also as it allows the separation of competencies. Thus, corporate IT security experts may create the modeling language, perhaps introducing modeling elements such as operating systems, networks, and firewalls, man-in-the-middle attacks, credentials theft and denial of service. Specific systems could subsequently be modeled and simulated by less security-savvy network and system administrators.

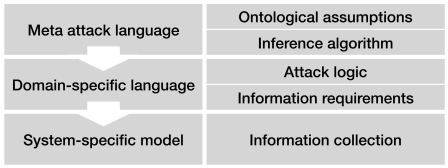


**Figure 1: Meta modeling hierarchy**

The creation of a domain-specific modeling language can thus solve the first challenge mentioned above, as the language may define what information about the system is required. It may partially address also the third question, as it can specify the generic attack logic of the domain. What remains, however, is the the way in which to represent the domain of interest, i.e. the ontological assumptions, and the underlying method to draw conclusions about the vulnerability of a system, i.e. the inference algorithm. These questions are tightly interwoven with the structure of that domain-specific modeling language. The question we address in this paper is thus how to represent domain-specific security modeling languages. Our proposed answer is a meta language that is also the main contribution of the paper. Such a meta language aims to support the definition of attack simulation languages for any domain, cf. 1.

The remainder of the paper is structured as follows. Section 2 considers related work. In Section 3, we describe the mathematical model that underlies the proposed language and simulations. Section 4 introduces the possibilities to compute the time required by an attacker to compromise various points in the system. The parts presented that far is synthesized into the Meta Attack Language (MAL), which is presented in Section 5, Next a small example in Section 6 is presented. How executable code for performing the simulations is generated from a MAL specification is explained in Section 7. Section 8 describes the performance characteristics of the simulations. Finally, the paper is concluded in Section 9.

## 2 RELATED WORK

MAL relates to two domains of previous work. Firstly, in the area of model-driven security engineering we also find domain-specific languages intended for security analysis of software and system models. Secondly, the MAL is using the formalism of attack/defense graphs as the core mechanism for its analysis.

The field of model-driven security engineering includes quite a large number of domain-specific languages where some of the earlier and more well-known include UMLsec [16, 17], SecureUML [3, 4], and SECTET [1, 10], and newer work includes STS-ml [29]. In these languages it is possible to specify a system design in terms of components and their interaction and also security properties such as constraints, requirements, or threats. Security analysis is then conducted using model checking and searches for constraint violations. So, e.g. if an access control policy is defined in the specification it is later possible to run conformance checks towards that policy for all components modelled in the specification. Different languages uses different formalisms and logic to build up their syntax and semantics, e.g. the Unified Modelling Language and the Object Constraint Language.

This approach promotes secure system design since it effectively separates security engineering from systems engineering. Here we can note that the purpose of these languages differ slightly from that of MAL. In MAL we want to support causality-based security predictions about a system (in use or on the drawing table) rather than producing a formally verified design. In brief this difference in objective stems from the intended scope of the analysis. With MAL we want to support analysis not only of the system design but also of the quality of the later implementation and configuration vulnerabilities of systems. Also the scope should cover both enterprise-wide systems-of-systems as well as individual components and small closed ecosystems.

There also exist quite a number of "modelling only" security languages, such as CORAS [21], secureTROPOS [23] and SecDSVL [2]. These languages capture security properties but does not provide any automated means to analyze or infer further conclusions from the model. Such analysis instead needs to be conducted in the heads of the modellers with out any further support. Consequently this kind of work is not so closely related to ours even though it appears so from plain sight.

Considering the general field of domain- specific languages, Marques do Nascimento et al. presents a systematic mapping study [6]. They have studied 2688 papers and found 36 related to security. None of the papers have attack simulation, threat modeling, or attack graphs as domain as is the domain of MAL.

The concept of attack trees is commonly attributed to Bruce Schneier and his seminal work from 1999 [31, 32]. The attack trees were later formalized by Mauw & Oostdijk [22] and further extended to also include defenses by Kordy et al. [18]. Since then, many attack-graph-based approaches have been presented, e.g. [14, 20, 28, 35], and largely summarized in [19].

A number of attack graph-based tools have also been developed. A common denominator for these tools is that they actively set out to collect information about some existing system or infrastructure and automatically attack graphs based on this data. Some of the more prominent tools are: MulVal [13, 27] that derives logical attack-graphs by associating the vulnerabilities extracted from scans with a probability that expresses how likely an attacker is to exploit them successfully. k-Zero Day Safety [34] extends MulVAL for the computation of zero-day attack graphs. NAVIGATOR [5] considers identified vulnerabilities as directly exploitable by the attacker, given that she has access to the vulnerable system. The TVA tool [24] models networks in terms of security conditions and uses a database of exploits as transitions between these security conditions. Similarly, NetSecuritas [9] composes scanners output and known exploits to generate attack graphs and corresponding security recommendations (mitigation metric).

Moreover, probabilistic attack graph modeling is a sub domain with quite some work, in particular employing Bayesian networks. In [8], the authors translate "raw" attack graphs obtained with the TVA-tool into dynamic Bayesian networks, and convert CVSS scores to probabilities. Similarly, the authors in [36] rely on CVSS to model uncertainties in the attack structure, attacker's actions and alerts triggering. The authors in [30] use Bayesian attack graphs to estimate the security risk on network systems and produce a security mitigation plan using a genetic algorithm.

None of the above-mentioned attack graph approaches are model based. There is work on combined attack graphs and system models, e.g. CySeMoL [33], P2CySeMoL [12], pwnPr3d [15], and securiCAD [7]. The common idea for all of this work is that probabilistic attack graphs are automatically generated and calculated from a given system specification, devised in the respective frameworks' separate language. The attack graph thus constitutes the inference engine that produces predictive security analysis results from the system model. However, all of languages are in turn implemented in hard coded, inflexible, and partly different languages, making them hard to change and re-use.

This article's proposal of a meta-language for specifying domain-specific attack modeling languages is, to the best of our knowledge, a novel contribution.

## 3 A FORMALISM FOR THREAT MODELING

In this section, we detail the mathematical formalism that underlies the Meta Attack Language.

Let $x$ denote an object or domain entity. In an example of physical security, an object might be a particular diamond, say the `cullinanDiamond`. Another object might be the `antwerpVault`. Objects are partitioned into a set of classes $\mathbf{X} = \{X_1, ..., X_n\}$, e.g.

$$\texttt{cullinanDiamond} \in \texttt{Jewelry}$$

and

$$\texttt{antwerpVault} \in \texttt{Vault}$$

Each class is associated with a set of attack steps $A(X_i)$. We use $X.A$ to denote the attack step $A$ of an object in class $X$. Examples of attack steps could be `Jewelry.steal`, `Vault.open`. A *link* relationship, $\lambda$, is a binary tuple of objects, each taken from a class,

$$\lambda = (x_i, x_j)$$

For instance, the `antwerpVault` might have a `containment` link to the `cullinanDiamond`.

Links are partitioned into a set of *associations*, $\mathbf{\Lambda} = \{\Lambda_1, ..., \Lambda_n\}$, which relate classes to each other, so that

$$x_i, x_k, \in X_m, x_j, x_l \in X_n | \lambda_1 = (x_i, x_j) \in \Lambda \wedge \lambda_2 = (x_k, x_l) \in \Lambda,$$

Classes play *roles* in associations, $\Psi(X_i, \Lambda)$, so that the association `containment` bestows the `container` role upon the `Vault` with respect to the `Jewelry`. We use $X.\Psi$ to denote the opposite end of an association, e.g.

$$\texttt{Jewelry.container} = \texttt{Vault}$$

Similarly, on the instance level,

$$\psi(x_i, \Lambda) = \{x_j | x_i, x_j \in \lambda \wedge \lambda \in \Lambda\}$$

e.g.

$$\texttt{antwerpVault.contained} = \{\texttt{cullinanDiamond}, \texttt{lesothoPromise}\}$$

Navigating over associations, attack steps may be connected to each other with directed edges, $e \in E$,

$$e = (X_i.A_k, X_j.A_l) | X_j = \Psi(X_i, \Lambda)$$

Such a connection between attack steps implies that the first step leads to the second.

$$e = (\texttt{Vault.open}, \texttt{Vault.contained.steal}), e \in E$$

would, for instance, imply that the opening of a vault leads to the stealing of its contents.

A compulsory, singleton class is the *Attacker*, $\Xi \in \mathbf{X}$, containing a single attack step, $\Xi.\xi$, constituting the starting point of an attack on the modelled system. To each attack step a probability distribution is associated specifying the expected time it would take the attacker to perform the step, the *local time to compromise*, $\phi(A) = P(T_{loc}(A) = t)$. As an example, the time required to perform `antwerpVault.open` might be specified by a Gamma distribution with a mean of 12 and a standard deviation of 6 hours,

$$\phi(\texttt{antwerpVault.open}) = \text{Gamma}(24, 0.5)$$

Attack steps may be of the type OR or AND, $t(X.A) \in \{OR, AND\}$. OR signifies that the attacker can start working on the attack step as soon as a single parent attack step is compromised, while AND requires that all parent attack steps are compromised. If, for example, $t(\texttt{Vault.open}) = \text{AND}$,

$$(\texttt{myVaultKey.compromise}, \texttt{myVault.open}) \in E,$$

and

$$(\texttt{myVaultPhysicalLocation.access}, \texttt{myVault.open}) \in E$$

then it is required to both gain access to the physical location of the vault and to obtain the key before work can commence on the opening of the vault.

The probability distribution of the total time required to compromise an attack step as measured from the initial attack step, $\xi$, thus including any intermediary required steps, is denoted $\Phi(A) = P(T_{glob}(A) = t)$.

In addition to attack steps, classes may also feature *defenses*, $D(X_i)$. We use $X.D$ to denote the defense $D$ of an object in the class $X$. A defense has a state of TRUE or FALSE, $s(D) \in \{\text{TRUE}, \text{FALSE}\}$, e.g., `Vault.timeLocked`. A defense can be the parent of an attack step, $(X_i.D, X_j.A) \in E$, e.g.,

$$(\texttt{Vault.timeLocked}, \texttt{Vault.open}) \in E,$$

denoting that only if `Vault.timeLocked` = FALSE will `Vault.open` be possible to perform. Defenses may also be impacting the time to compromise distributions.

## 4 GLOBAL TIME TO COMPROMISE

Given an instantiated specification in the above presented formalism, we are interested in computing the global time to compromise of various attack steps. Such a value will provide a measure of how secure various points of the system are in terms of attack resilience, and it will give us a quantitative way of comparing systems designs. Specifically, comparing different vault designs, *ceteris paribus*, we

would prefer the one with the higher global time to compromise on the `Jewelry.steal` attack steps, e.g. Φ(`lesothoPromise.steal`)[1]

The computation of the global time to compromise requires assumptions not only about the expected local time to compromise of the individual attack steps, but also assumptions of the order in which the attacker will attempt various available attack steps. An attacker with exceptionally poor planning capabilities could, for instance, be modeled by a random walk, featuring an even probability of selecting any of the available attack steps at a given point in the attack. An perfectly rational and all-knowing attacker would, instead, always select the shortest path to every attack step. Thus, the global time to compromise of a specific attack step would be the shortest path from the attacker to that step. It is also possible to envision attackers with capabilities between these two extremes.

## 5 THE META ATTACK LANGUAGE (MAL)

Models in the above described formalism can be encoded in a number of languages, e.g. programming languages such as Python or Java, which when executed can perform the aforementioned attack simulations, i.e. the computation of the global time to compromise of the model's attack steps. However, those programs will not be very concise or user-friendly.

Therefore, we have created a compiler that generates program code (currently Java, but in principle any programming language) from specifications in a domain-specific language we call the Meta Attack Language (MAL). As all domain-specific languages, this language is mainly intended to increase the comprehensibility of the models.[2] This section presents the core parts of the grammar (syntax) for MAL, see Appendix for details.

As presented above, classes containing attack steps constitute the core entities of a MAL specification. A class is specified as such:

```
class Channel {
    | transmit
        ->  parties.connect
}
```

`Channel` is the name of the class, and `transmit` is the name of its unique attack step. The symbol `|` indicates that the attack step is of the type OR (so that the compromise of a single parent is sufficient to reach this node). Attack steps of the type AND are represented by the symbol `&`, while defenses are denoted by a `#`. The arrow, `->`, signifies that the compromise of attack step `transmit` opens for an attack on the attack steps `parties.connect`. `parties` is an association role, which is defined in a separate part of the MAL specification.

```
associations {
   Machine [parties] 2-* <-- Communication  --> * [channels] Channel
}
```

As for traditional UML class diagrams, association ends are bound to types with multiplicities. In this example, an object of the class `Channel` needs to be connected to at least two objects of the class `Machine`. Both ends of an association have roles, which are used for navigation. Thus, `myChannel.parties` refers to the set of `Machine` objects connected to `myChannel`. And, returning to the definition of the `Channel.transmit` attack step, its compromise then opens for an attack on the `connect` attack step of the connected `Machine` objects, i.e. the `Channel`'s `parties`.

To foster the reuse of class definitions, MAL features inheritance in a manner similar to other object-oriented languages. Classes that are only intended for specialization, and never instantiation, may be defined as abstract. In the below example, `Machine` is an abstract class, specialized into the concrete classes `Hardware` and `Software`, which inherit the attack steps and associations of `Machine`. `Software` overrides the `Machines`'s attack step `compromise` with its own definition.

```
abstractClass Machine {
    | connect
        ->  compromise
    & compromise
        -> channels.transmit
}

class Hardware extends Machine {
 }

class Software extends Machine {
    & compromise
        ->  channels.transmit,
            executors.connect,
}
```

Many attack steps may be compromised without effort. For instance, in the aforementioned example, as soon as the attacker is able to `transmit` on a `Channel`, then the `Machine.connect` attack step on the communicating parties immediately become available. However, sometimes we expect attack steps to require a certain amount of time. For each attack step, we can optionally specify the required time. For instance, that the time to crack a password with a dictionary attack takes 18 hours can be specified as follows,

```
class Credentials {
    & dictionaryCrack [18.0]
}
```

Of course, the time required to complete an attack step may very well be dependent on many factors that are not specified in the considered model. For instance, the time to crack the aforementioned password will depend on the choice of password and the choice of cracking dictionary. The earlier the password is listed in the employed dictionary, the faster it will be cracked. Obviously, we will not know these specifics, we might not know the employed password encryption algorithm, and we may be ignorant of the hardware employed by the attacker. To express our uncertainties related to the language we are developing, we employ probability distributions. Thus, we may replace the above deterministic time estimate with a probabilistic one, indicating that on average, the crack is expected to take 18 hours, but with significant uncertainty.

```
class Credentials {
    & dictionaryCrack [GammaDistribution(1.5, 15)]
}
```

To allow the configuration of objects at the time of instantiation, we employ defenses. For instance, we might want to be able

[1] In the case of multiple valuable assets, we can instead consider a weighted mean of the concerned attack steps' global time to compromise.

[2] It is important to not underestimate the benefits of increased comprehensibility; were it not for the need for comprehensibility, all software would have been programmed directly in machine code.

to configure some instances of the class Credentials to be encrypted, while leaving others unencrypted. In this case, the defense Credentials.encrypted may be introduced. Defenses are represented with the symbol # and assume boolean values to indicate their status. Technically, each defense includes an attack step. If the defense is false, then, at the time of instantiation, the associated attack step is marked as compromised. The effect can be understood in the below example:

```
class Credentials {
    | access
        -> compromiseUnencrypted
    & compromiseUnencrypted
    # encrypted
        ->  compromiseUnencrypted
}
```

If Credentials.encrypted is false, then the attacker will be able to reach the AND attack step compromiseUnencrypted as soon as she has reached access. If, instead, Credentials.encrypted is true, then compromiseUnencrypted will not be reached, as it's compromise requires the compromise of both parents.

## 6 AN EXAMPLE

Although MAL is capable of expressing any kind of attack, we expect cybersecurity to be the main domain of application. In this section, we provide a simple example of the MAL specification language employed on that domain. In more realistic uses, the level of detail will be significantly higher. For instance, our previously developed threat modelling languages can be expressed in MAL, and we are currently developing a number of detailed specifications of various information security domains, including general corporate IT, cloud platform security, and cyber security in motor vehicles.

### 6.1 Example MAL specification

Below, we present a small but comprehensive MAL specification related to the fragments displayed in the previous section. The base class is Machine, which is abstract, so that only its specializations, Hardware and Software may be used in an instantiated model. Machines may execute Software, e.g. a workstation may execute an operating system, which in turn may execute an application. Two or more Machines may be connected by a Channel. As an example, a web browser Software may be connected to a web server Software over an HTTPS Channel. A final class in this small example is Credentials, representing, e.g. user names and passwords or private keys. Credentials have targets (Machines), for which they serve as authentication, and they may be stored on Machines.

```
abstractClass Machine {
    | connect
        ->  compromise
    | authenticate
        ->  compromise
    & compromise
        -> _machineCompromise
    | _machineCompromise
        ->  executees.compromise,
            storedCreds.access,
            channels.transmit
}

class Hardware extends Machine {
 }

class Software extends Machine {
    & compromise
        ->  _machineCompromise,
            executors.connect
}

class Channel {
    | transmit
        ->  parties.connect
}

class Credentials {
    | access
        -> compromiseUnencrypted,
           dictionaryCrack
    & compromiseUnencrypted
        -> compromise
    | dictionaryCrack [GammaDistribution(1.5, 15)]
        ->  compromise
    | compromise
        ->  targets.authenticate
    # encrypted
        ->  compromiseUnencrypted
}

associations {
   Machine [executor] 1 <-- Execution --> * [executees] Software
   Machine [parties] 2-* <-- Communication --> * [channels] Channel
   Machine [stores] * <-- Storage --> * [storedCreds] Credentials
   Machine [targets] * <-- Access --> * [authCreds] Credentials
}
```

Considering the attack steps, the class Machine contains four. As the name indicates, compromise represents that the attacker has gained control over the said machine. To reach compromise, both connect and authenticate must first be reached. connect represents the attacker's establishment of contact with the Machine, e.g. a login request, while authenticate represents the presentation of appropriate credentials. If compromise is reached, then all Software that is executed by the compromised Machine also becomes compromised. Furthermore, any Credentials stored on the Machine become accessible. Finally, any connected Channel becomes accessible for transmission. The specialization Hardware contains nothing beyond Machine, but Software appends one causal consequence to the attack step compromise. If a Software is compromised, then the attacker is able to connect to the underlying executor. For instance, if application is compromised, then the attacker is able to connect to the operating system.

The class Channel features a single attack step, transmit, which, if compromised, allows the attacker to connect to all of the Channel's parties.

Finally, the Credentials class contains four attack steps and one defense. By compromising the Machine that stores the credentials, the attacker gains access to the credentials. This does not, however, necessarily mean that the credentials are compromised, as they very well may be encrypted (or, which we treat as equivalent in

this example, hashed). We allow the modeller to specify whether this indeed is the case by the defense `encrypted`. If `encrypted` is false AND the attacker has reached the attack step `access`, then this will lead to the attack step `compromiseUnencrypted`, which in turn directly leads to `compromise`. However, if `encrypted` is true, then that attack path is thwarted. However, there is another option, as `access` also leads to `dictionaryCrack`. This attack step will require the attacker to spend time, as expressed by the associated probability distribution. Most likely, it will take round 18 hours, but there is a 5% probability that it will take closer to 60 hours, and a slight chance that it might take over a week. The specification is also presented as a diagram, cf. 2.

This specification is limited to the class level. Thus, instantiation remains. As mentioned, it is useful to divide class and instance level for two reasons. Firstly, this separation allows security experts to codify their knowledge on the class level, while users with less security competence but detailed knowledge of particular networks, such as corporate network and system administrators, efficiently will be able to instantiate specific object models. Secondly, the class level is typically reusable for many instances. It is possible to create reusable class level specifications of various domains. Currently, we pursue such work in multiple domains. One such domain is vehicle security, including vehicle-specific classes such as `ECU` and `CANBus`. Another domain for which we are developing a class-level MAL specification is Amazon Web Services, including such classes as `S3Bucket` and `EC2Instance`.

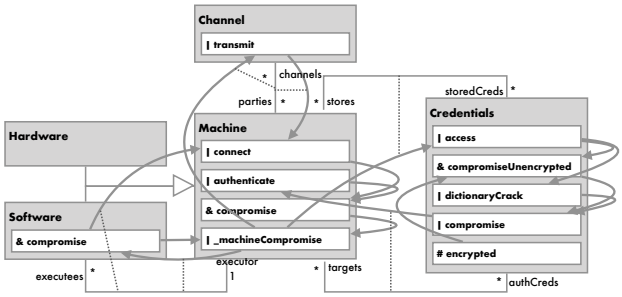


**Figure 2: Example MAL specification diagram.**

## 6.2 Example of instantiated scenario

As will be presented in the next section, we automatically generate executable Java classes from the above specification. These classes can easily be instantiated into a concrete model by a program as the one presented here:

```
Hardware    macBook =   new Hardware();
Software    macOS =     new Software();
Software    sshClient = new Software();
Credentials sshKey =
   new Credentials(encrypted=Bernoulli(0.5));

macOS.addExecutor(macBook)
sshClient.addExecutor(macOS)
macOS.addStoredCreds(sshKeys)

Hardware    system76 =  new Hardware();
Software    ubuntu16 =  new Software();
Software    sshDaemon = new Software();

ubuntu16.addExecutor(system76);
sshDaemon.addExecutor(ubuntu16)
sshKey.addTargets(sshDaemon)
sshKey.addTargets(ubuntu16)

Channel     sshChannel = new Channel();

sshChannel.addParties(sshClient)
sshChannel.addParties(sshDaemon)

Attacker    attacker =  new Attacker();
attacker.addAttackPoint(macOS.compromise)
```

In this small example, the attacker is assumed to have compromised the `macOS` operating system of a `macBook`, perhaps by logging on locally while the machine was unattended. An `sshClient` is executing on the Mac. The MAL rule `Machine.compromise -> executees.compromise` implies that the compromise of the `macOS` directly leads to the compromise of the `sshClient`. Furthermore, `Machine.compromise -> storedCreds.access` leads the attacker to access the SSH keys on the Mac. Access, however, is not sufficient, as SSH keys may be encrypted with a passphrase. The probability that the SSH keys indeed are encrypted is set to 50% in the instantiation of the sshKey object,

`Credentials sshKey = new Credentials(encrypted=Bernoulli(0.5));`[3]

If the `sshKey` is unencrypted, access will indeed be sufficient to compromise the keys. Otherwise, the attacker will need to perform a dictionary attack on the keys, which will take some time, as given by `dictionaryCrack [GammaDistribution(1.5, 15)]`.

Compromising the `sshClient`, the rules `Machine.compromise -> channels.transmit` and `Channel.transmit -> parties.connect` allow the attacker to connect to the `sshDaemon`. `connect` will, however, only lead to `compromise` if `authenticate` is also reached. If the attacker has succeeded in compromising the SSH key passphrase, the rule `Credentials.compromise -> targets.authenticate` will lead to such a compromise of the `sshDaemon`. This, in turn allows the attacker to `connect` to the underlying `ubuntu16` operating system, which is instantly compromised, as the keys were specified to provide access not only to the `sshDaemon` but also to the OS.

In this example, there are only two stochastic distributions, the first representing the probability that the passphrase is encrypted, and the second representing the time to crack the passphrase in the case that it is indeed encrypted. No other attack steps in this small example are expected to require any significant time. Adopting the assumption of a perfectly rational attacker, we can determine the global time to compromise of, e.g., `ubuntu16.compromise` by calculating the shortest path from the attacker to that attack step. In the resulting distribution, half of the probability mass will be located at time zero (because in half of the attacks, the passphrase will be unencrypted, so access will be immediate), while the remaining half will be distributed according to the `dictionaryCrack` local time-to-compromise probability distribution. This small model

[3] A Bernoulli distribution specifies the probability of a boolean variable assuming the value TRUE.

can be calculated manually to yield the described global time-to-compromise distribution of `ubuntu16.compromise`, but for larger models, computations need to be automated.

## 7 AUTOMATIC CODE GENERATION

It is desirable to automate the computation of the global time to compromise of each attack step. Efficient computation can be achieved in the following manner. A MAL specification in the above presented format can be transformed into a set of classes in an object-oriented language, such as Java or C++. The thus produced classes may subsequently be instantiated to create a concrete attack scenario, which in turn can be automatically simulated by executing the thus generated program.

We have implemented a Java code generator using the ANTLR framework. From a grammar, ANTLR generates a parser that can build and walk parse trees. By walking this intermediary representation, code can be generated by a target-code-specific listener. The listener skeleton is generated by the ANTLR framework and customized for the target language.

Each class in the MAL specification is translated into a Java class, while attack steps are translated into nested classes. The graph structure represented by the network of attack steps is represented by a set of methods. To exemplify, a MAL class

```
class Channel {
    | transmit
        ->  parties.connect
}
associations {
   Machine [parties]  2-* <-- Communication --> * [channels] Channel
}
```

would translate to a Java class

```
...

public class Channel {
   public AnySet<Machine> parties = new AnySet<>();
   transmit = new Transmit(...);
      ...

   public class Transmit extends AttackStepMin {
      ...
      @Override
      protected void setExpectedParents() {
         for (Machine machine : parties) {
            addExpectedParent(machine.compromise);
         }
      }
      @Override
    public void updateChildren(Set<AttackStep> activeAttackSteps) {
         for (Machine machine : parties) {
        machine.connect.updateTtc(this, ttc, activeAttackSteps);
         }
      }
      @Override
      public double localTtc() {
         return ttcHashMap.get("channel.transmit");
      }
   }

   public void addParties(Machine machine) {
      this.parties.add(machine);
      machine.channels.add(this);
   }
   ...
}
```

The methods `setExpectedParents()` and `updateChildren()` create the relations between attack steps, thus constructing the class-level structure of the attack graph. The object-level model is generated by instantiating the Java classes into objects, and connecting them to each other by calling auto-generated methods such as `Channel.addParties()`. The object-level model can subsequently be algorithmically traversed to compute the time to compromise of each attack step, e.g. with a shortest-path algorithm.

## 8 COMPUTATIONAL PERFORMANCE

In the previous section, we presented a small example model. For real computer networks, however, models can become very large, comprised of thousands, possibly even millions, of attack steps. Computational performance therefore becomes important.

As it seems wiser for the defender to prepare for a highly rational attacker rather than the opposite, we have implemented a variant of the shortest path algorithm to compute the global time to compromise of attack steps in the model. The traditional shortest path problem only considers graphs with nodes corresponding to OR attack steps. These are operated on by a MIN function, so that the time to reach a given node is the shorted time to reach any parent plus the local time increment of the attack step,

$$T_{glob}(A_{child}) = min(\{T_{glob}(A_{parent_1}), ..., T_{glob}(A_{parent_n})\}) + T_{loc}(A_{child})$$

where

$$t(A_{child}) = \text{OR}$$

and

$$e(A_{child}, A_{parent_i})$$

Our algorithm computes AND attack steps by replacing the MIN with a MAX function,

$$T_{glob}(A_{child}) = max(\{T_{glob}(A_{parent_1}), ..., T_{glob}(A_{parent_n})\}) + T_{loc}(A_{child})$$

where

$$t(A_{child}) = \text{AND}$$

The benefit of this modification is the ability to approximate AND attack steps with maintained computational efficiency. The downside is that the approximation affects precision. In our test graphs, which we strived to make as realistic as possible with respect to their connectedness as well as the relative proportion of AND and OR attack steps, errors were small. However, in cases where many independent parent attack steps have similar global time to compromise, the error can become significant. In these cases, an exact algorithm may be used, but at a price in computational efficiency.

We have implemented this shortest path computation in a Graphical Processing Unit (GPU) as a modified version of the Single Source Shortest Path (SSSP) algorithm proposed by Harish and Narayanan [11]. Our main modification is the incorporation of the AND attack step as described above. Also, the SSSP algorithm is deterministic. In order to perform probabilistic computations, we envelope the deterministic algorithm in a Monte Carlo simulation. Thus, a large set of graphs is generated with local time-to-compromise values for

each attack step sampled from their probability distributions. The SSSP algorithm is then used to compute the global time to compromise for each attack step in each graph. The resulting set of global time-to-compromise values for each attack step then approximates the actual distribution.

On an Apple MacBook from 2016 with an Intel HD Graphics 515 graphics card with 1536 MB RAM, this algorithm was able to compute 1000 samples of graphs will half a million nodes in under three minutes. Thus, using relatively unimpressive hardware, remarkably large networks can be computed.

## 9 SUMMARY

In this article, we have presented MAL, the Meta Attack Language. MAL allows security experts to codify domain-specific knowledge in order to allow simulations of attacks on systems in the domain of interest. The thus generated domain-specific attack modeling languages may subsequently be used and re-used by people with lesser security expertise in order to automatically assess the security of specific systems within the domain.

In the paper, we described the syntax and semantics of MAL, exemplified its use, and presented our findings on the computational performance of the associated attack simulations.

## Acknowledgements

This work has received funding from the Swedish Governmental Agency for Innovation Systems and its program on Strategic Vehicle Research and Innovation as well as the Swedish Civil Contingencies Agency through the research centre Resilient Information and Control Systems (RICS).

## REFERENCES

[1] Muhammad Alam, Ruth Breu, and Michael Hafner. 2007. Model-driven security engineering for trust management in SECTET. *JSW* 2, 1 (2007), 47–59.

[2] Mohamed Almorsy and John Grundy. 2014. Secdsvl: A domain-specific visual language to support enterprise security modelling. In *Software Engineering Conference (ASWEC), 2014 23rd Australian*. IEEE, 152–161.

[3] David Basin, Manuel Clavel, and Marina Egea. 2011. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies*. ACM, 1–10.

[4] David Basin, Jürgen Doser, and Torsten Lodderstedt. 2006. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 1 (2006), 39–91.

[5] Matthew Chu, Kyle Ingols, Richard Lippmann, Seth Webster, and Stephen Boyer. 2010. Visualizing attack graphs, reachability, and trust relationships with NAVIGATOR. In *Proc. of the 7th Int. Symp. on Visualization for Cyber Security*. ACM, 22–33.

[6] Leandro Marques do Nascimento, Daniel Leite Viana, Paulo AM Silveira Neto, Dhiego AO Martins, Vinicius Cardoso Garcia, and Silvio RL Meira. 2012. A systematic mapping study on domain-specific languages. In *Proc. 7th Int. Conf. Softw. Eng. Advances (ICSEAâĂŹ12)*. 179–187.

[7] Mathias Ekstedt, Pontus Johnson, Robert Lagerström, Dan Gorton, Joakim Nydrén, and Khurram Shahzad. 2015. securiCAD by foreseeti: A CAD tool for enterprise cyber security management. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*. IEEE, 152–155.

[8] Marcel Frigault, Lingyu Wang, Anoop Singhal, and Sushil Jajodia. 2008. Measuring network security using dynamic bayesian network. In *Proc. of the 4th ACM workshop on Quality of protection*. ACM, 23–30.

[9] Nirnay Ghosh, Ishan Chokshi, Mithun Sarkar, Soumya K Ghosh, Anil Kumar Kaushik, and Sajal K Das. 2015. NetSecuritas: An Integrated Attack Graph-based Security Assessment Tool for Enterprise Networks. In *Proc. of the 2015 Int. Conf. on Distributed Computing and Networking*. ACM, 30.

[10] Michael Hafner, Ruth Breu, Berthold Agreiter, and Andrea Nowak. 2006. SECTET: an extensible framework for the realization of secure inter-organizational workflows. *Internet Research* 16, 5 (2006), 491–506.

[11] Pawan Harish and PJ Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*. Springer, 197–208.

[12] H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. 2015. $P^2$CySeMoL: Predictive, Probabilistic Cyber Security Modeling Language. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (2015), 626–639. DOI: http://dx.doi.org/10.1109/TDSC.2014.2382574

[13] John Homer, Su Zhang, Xinming Ou, David Schmidt, Yanhui Du, S Raj Rajagopalan, and Anoop Singhal. 2013. Aggregating vulnerability metrics in enterprise networks using attack graphs. *Journal of Computer Security* 21, 4 (2013), 561–597.

[14] Kyle Ingols, Matthew Chu, Richard Lippmann, Seth Webster, and Stephen Boyer. 2009. Modeling modern network attacks and countermeasures using attack graphs. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 117–126.

[15] Pontus Johnson, Alexandre Vernotte, Mathias Ekstedt, and Robert Lagerström. 2016. pwnPr3d: An Attack-Graph-Driven Probabilistic Threat-Modeling Approach. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*. IEEE, 278–283.

[16] Jan Jürjens. 2002. UMLsec: Extending UML for secure systems development. In *International Conference on The Unified Modeling Language*. Springer, 412–425.

[17] Jan Jürjens. 2005. *Secure systems development with UML*. Springer Science & Business Media.

[18] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. 2010. Foundations of attack–defense trees. In *International Workshop on Formal Aspects in Security and Trust*. Springer, 80–95.

[19] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. 2014. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review* 13 (2014), 1–38.

[20] Igor Kotenko and Elena Doynikova. 2014. Evaluation of computer network security based on attack graphs and security event processing. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 5, 3 (2014), 14–29.

[21] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. 2010. *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media.

[22] Sjouke Mauw and Martijn Oostdijk. 2005. Foundations of attack trees. In *International Conference on Information Security and Cryptology*. Springer, 186–198.

[23] Haralambos Mouratidis, Paolo Giorgini, Gordon Manson, Ian Philp, and others. 2002. A Natural Extension of Tropos Methodology for Modelling Security. In *Proceedings Agent Oriented Methodologies Workshop, Annual ACM Conference on Object Oriented Programming, Systems, Languages (OOPSLA), Seattle-USA*. Citeseer.

[24] S. Noel, M. Elder, S. Jajodia, P. Kalapa, S. O'Hare, and K. Prole. 2009. Advances in Topological Vulnerability Analysis. In *Conference For Homeland Security, 2009. CATCH '09. Cybersecurity Applications Technology*. 124–129. DOI: http://dx.doi.org/10.1109/CATCH.2009.19

[25] Object Management Group (OMG). 2016. Meta-Object Facility (MOF) Core Specification, Version 2.5.1. OMG Document Number: formal/2016-11-01 (http://www.omg.org/spec/MOF/2.5.1). (2016).

[26] Object Management Group (OMG). 2017. OMGÂő Unified Modeling LanguageÂő (OMG UMLÂő), Version 2.5.1. OMG Document Number: formal/2016-11-01 (http://www.omg.org/spec/UML/2.5.1). (2017).

[27] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. 2005. MulVAL: A Logic-based Network Security Analyzer.. In *USENIX security*.

[28] Xinming Ou and Anoop Singhal. 2011. Attack Graph Techniques. *Quantitative Security Risk Assessment of Enterprise Networks* (Jan. 2011). DOI: http://dx.doi.org/10.1007/978-1-4614-1860-3_2

[29] Elda Paja, Fabiano Dalpiaz, and Paolo Giorgini. 2015. Modelling and reasoning about security requirements in socio-technical systems. *Data & Knowledge Engineering* 98 (2015), 123–143.

[30] N. Poolsappasit, R. Dewri, and I. Ray. 2012. Dynamic Security Risk Management Using Bayesian Attack Graphs. 9, 1 (2012), 61–74. DOI: http://dx.doi.org/10.1109/TDSC.2011.34

[31] Bruce Schneier. 1999. Attack trees. *Dr. DobbâĂŹs journal* 24, 12 (1999), 21–29.

[32] Secrets Schneier. 2000. Lies: digital security in a networked world. *New York, John Wiley & Sons* 21 (2000), 318–333.

[33] Teodor Sommestad, Mathias Ekstedt, and Hannes Holm. 2013. The cyber security modeling language: A tool for assessing the vulnerability of enterprise system architectures. *IEEE Systems Journal* 7, 3 (2013), 363–373.

[34] Lingyu Wang, S. Jajodia, A. Singhal, Pengsu Cheng, and S. Noel. 2014. k-Zero Day Safety: A Network Security Metric for Measuring the Risk of Unknown Vulnerabilities. 11, 1 (2014), 30–44. DOI: http://dx.doi.org/10.1109/TDSC.2013.24

[35] Leevar Williams, Richard Lippmann, and Kyle Ingols. 2008. *GARNET: A graphical attack graph and reachability network evaluation tool*. Springer.

[36] Peng Xie, Jason H Li, Xinming Ou, Peng Liu, and Renato Levy. 2010. Using Bayesian networks for cyber security analysis. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP Int. Conf. on*. IEEE, 211–220.