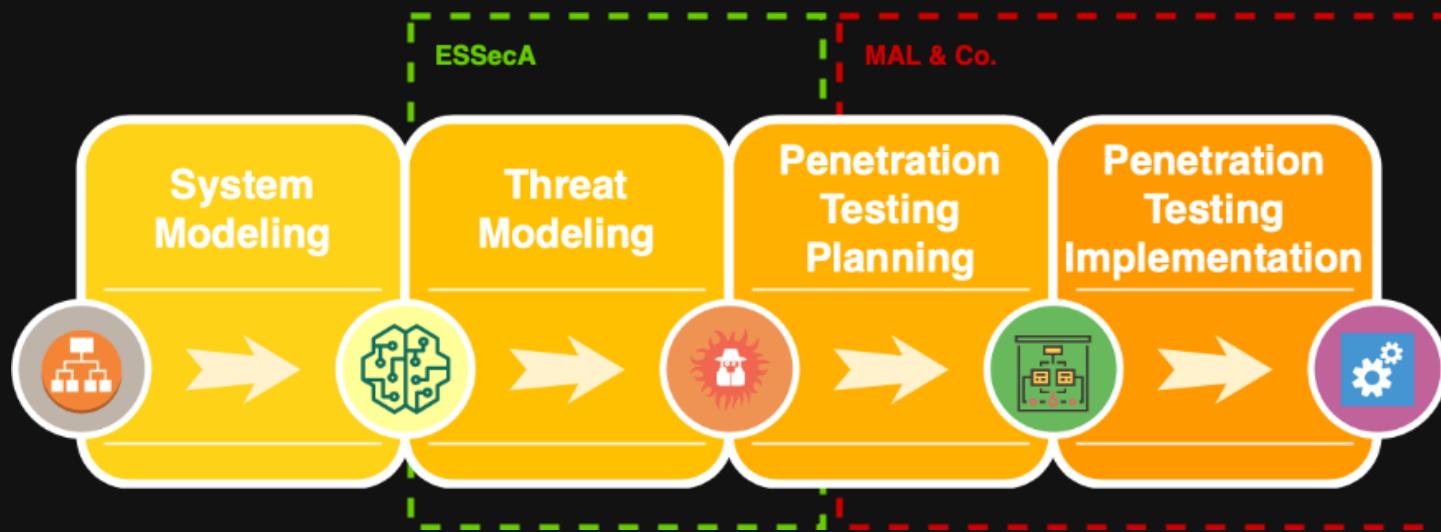


Pitch

Press Space for next page →

ⓘ Use arrows to move ← →

Target



MAL

Meta Attack Language

Purpose

- Design di linguaggi domain-specific;
- Fornisce un formalismo che consente la generazione semi-automatizzata e il calcolo efficiente di "grafi di attacco" [1] molto grandi; per esempio, un attaccante può
 1. Compromettere un host
 2. Trovare le credenziali salvate
 3. Utilizzare le credenziali per accedere ad un altro host
- Il risultato finale di una simulazione è quello di stimare il tempo impiegato in ciascun passaggio per eseguire l'attacco.

1. Grafi diretti che rappresentano le dipendenze tra i passaggi che possono essere eseguiti dall'attaccante. 

Modelling only security languages

- **CORAS:** Common Object-oriented Representation of Attack Scenarios
- **secureTROPOS:** Secure Threat and Risk Object-oriented Process
- **SecDSVL:** Secure Domain Specific Vulnerability Language

Questi linguaggi permettono di modellare le proprietà di sicurezza di un sistema, ma non sono in grado di fornire alcuno strumento automatico per analizzare o inferire conclusioni sulla sicurezza del modello stesso.

Formalism for Threat Modeling

Classi e step di attacco

- Sia x un'entità del dominio, ad esempio, un'entità potrebbe essere un diamante *cullinanDiamond*, un'altra una cassaforte *antwerpVault*.
- Gli oggetti sono divisi in un set di classi $X = \{X_1, \dots, X_n\}$, e.g.

$$cullinanDiamond \in Jewelry$$

e

$$antwerpVault \in Vault$$

- A ciascuna classe è associato un set di step di attacco $A(X_i)$.
 - Denotiamo con $X.A$ lo step di attacco per l'oggetto nella classe X .
 - Esempi di step di attacco possono essere *Jewelry.steal* o *Vault.open*.

Formalism for Threat Modeling

Link e associazioni

- Un **link** è una tupla di oggetti, ciascuno appartenente ad una classe diversa, e rappresenta una dipendenza tra gli oggetti, e.g.

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

- Ad esempio, *antwerpVault* potrebbe avere un link *containment* con *cullinanDiamond*, che rappresenta il fatto che la cassaforte può contenere il diamante.
- I link sono divisi in set di **associazioni** $\Lambda = \{\Lambda_1, \dots, \Lambda_n\}$, che collegano una classe all'altra, e.g.

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

Formalism for Threat Modeling

Ruoli e associazioni

- Nelle associazioni, le classi ricoprono dei **ruoli** $\Psi(X_i, \Lambda)$.
 - Ad esempio, *antwerpVault* potrebbe avere il ruolo *container* nella relazione *containment*.
 - Analogamente, a livello di istanze,

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

Ad esempio,

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

- Navigando tra le associazioni, gli step di attacco possono essere connessi tra di loro attraverso degli archi diretti $e \in E$.

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

- Una connessione tra due step di attacco implica che dal primo step si può passare al secondo, e.g.

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

sta a significare che se si riesce ad aprire la cassaforte, si possono rubare i diamanti contenuti.

Formalism for Threat Modeling

Attacker, tempo di attacco locale

- Una classe obbligatoria è il singleton **Attacker**, $\Xi \in X$, contenente un singolo step di attacco $\Xi.\xi$, il quale rappresenta il punto di partenza dell'attacco sul modello del sistema.
- Per ciascuno step di attacco $A(X_i)$, è definito un **tempo di attacco locale** (stimato) $\phi(A) = P(T_{loc}(A) = t)$.
 - Per esempio, il tempo di attacco locale per aprire la cassaforte potrebbe essere specificato da una distribuzione Gamma, con media 12 e deviazione standard di 6 ore.

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

Formalism for Threat Modeling

Tipi di step di attacco e tempo di attacco globale

- Gli step di attacco possono essere di tipo *AND* o *OR*

$$t(X.A) \in \{OR, AND\}$$

- *OR* significa che l'attaccante può iniziare lo step di attacco se almeno uno dei passaggi precedenti è stato completato.
- *AND* significa che l'attaccante può iniziare lo step di attacco solo se tutti i passaggi precedenti sono stati completati.
- Per esempio, se $t(Vault.open) = AND$ e

$$(myVault.compromise, myVault.open) \in E$$

e

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

allora, per poter aprire la cassaforte, è necessario che l'attaccante sia riuscito a compromettere il sistema (ottenere la chiave della cassaforte) e ad accedere alla sua posizione fisica.

- La distribuzione di probabilità del tempo totale richiesto per completare l'attacco è definita come

$$\Phi(A) = P(T_{glob}(A) = t)$$

Formalism for Threat Modeling

Difese

- Oltre a definire gli step di attacco, le classi possono anche prevedere delle **difese** $D(X_i)$, che rappresentano le azioni che possono essere intraprese per prevenire l'attacco. Utilizziamo $X.D$ per indicare la difesa per l'oggetto nella classe X .
 - Una difesa può essere *TRUE* o *FALSE*, $s(D) \in \{\text{TRUE}, \text{FALSE}\}$, che indica se la difesa è attiva o meno.
 - Ad esempio, *Vault.timeLocked* può essere *TRUE* o *FALSE*.
 - Una difesa può essere genitore di uno step di attacco, $(X_i.D, X_j.A) \in E$, e.g.

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

che significa che se la cassaforte è bloccata per un certo periodo di tempo ($Vault.\text{timeLocked} = \text{TRUE}$), allora non è possibile aprire la cassaforte.

Global time to compromise

Per il calcolo del tempo globale di attacco è stata implementata una versione modificata dell'algoritmo di Dijkstra, che tiene conto delle distribuzioni di probabilità per i tempi di attacco locali e per la scelta del prossimo step di attacco.

Se si utilizzasse il solo algoritmo di Dijkstra, staremmo considerando solo i nodi corrispondenti a delle `OR`: Dijkstra calcola il percorso minimo tra due nodi; dunque, ad ogni passo si sceglie il peso minimo, mediante una funzione `MIN`.

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

dove il collegamento tra i nodi è di tipo `OR`, $t(A_{child}) = OR$.

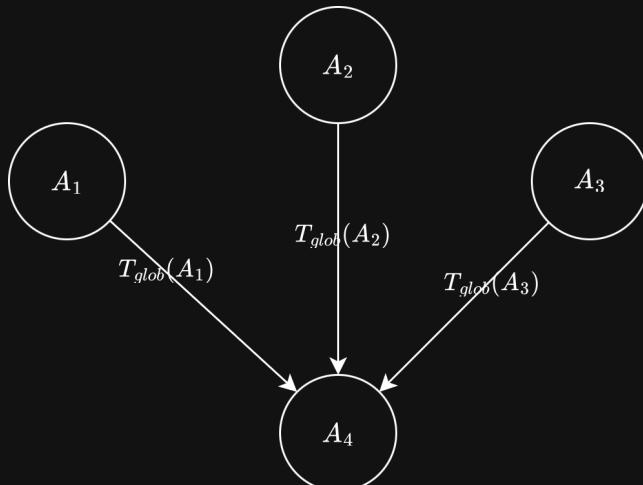
Volendo considerare anche i nodi con collegamento `AND`, dovremmo effettuare una somma dei pesi dei tempi globali dei nodi genitori:

$$T_{glob}(A_{child}) = \sum_{i=1}^n T_{glob}(A_{parent_i}) + T_{loc}(A_{child})$$

Per semplificare il calcolo, questa somma è stata sostituita con una funzione `MAX`:

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

dove il collegamento tra i nodi è di tipo `AND`, $t(A_{child}) = AND$.

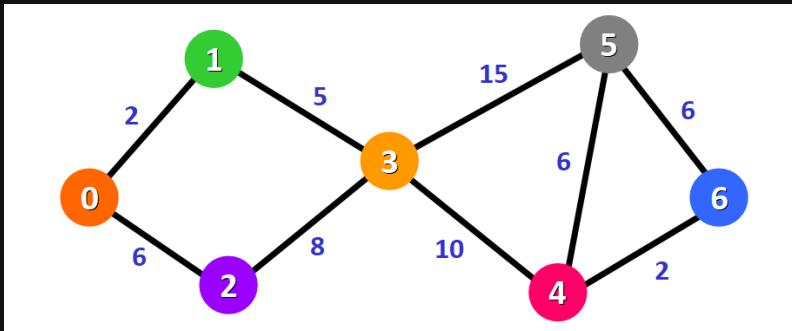


Global time to compromise

Il calcolo del tempo globale per compromettere il sistema non solo tiene conto dei tempi locali per ciascuno step di attacco, ma anche dell'ordine di esecuzione degli step di attacco stessi.

Ad esempio, un attaccante con scarse capacità di pianificazione potrebbe essere modellato attraverso una "random walk" sul grafo degli step di attacco. In tal caso, la distribuzione di probabilità per la scelta del prossimo step di attacco è uniforme.

Un attaccante esperto, invece, sceglie sempre il **percorso minimo** per raggiungere lo step di attacco successivo.



Il Meta Attack Language

Il modello matematico sopra descritto potrebbe essere codificato con un linguaggio di programmazione, come ad esempio Java, ma è molto più semplice utilizzare un linguaggio di alto livello, come il Meta Attack Language (MAL).

Gli autori hanno realizzato un compilatore che, a partire da una descrizione del modello in MAL, genera del codice in Java, che può essere utilizzato per calcolare il tempo globale per compromettere il sistema.

Il Meta Attack Language

Classi

Come specificato sopra, le classi sono le entità fondamentali di una specifica MAL. Una classe è definita come segue:

```
1  class Channel{  
2      | transmit  
3      -> parties.connect  
4  }
```

- Il simbolo `|` indica che lo step di attacco è di tipo OR; per cui, se almeno uno degli step di attacco genitori è stato completato, allora lo step di attacco può essere iniziato.
- Step di attacco di tipo AND sono indicati con il simbolo `&`, mentre le difese sono indicate con il simbolo `#` .
- La freccia `->` indica che la compromissione dello step di attacco `transmit` apre allo step di attacco `parties.connect` .

Il Meta Attack Language

Ruoli di associazione

`parties` è un ruolo di associazione, definito nella specifica MAL.

```
1  associations {
2    Machine [parties] 2-* <-- Communication --> * [channels] Channel
3 }
```

Come per l'UML, le associazioni terminano con una cardinalità, che indica il numero di istanze di una classe che possono essere associate ad un'altra classe. Nell'esempio, un oggetto della classe `Channel` deve essere associato ad almeno 2 istanze della classe `Machine`.

Entrambi gli estremi dell'associazione hanno dei ruoli, che sono utilizzati per la navigazione. Per cui, `myChannel.parties` si riferisce al set di oggetti della classe `Machine` associati a `myChannel`.

Il Meta Attack Language

Ereditarietà

Per permettere il riuso delle classi, MAL fornisce un meccanismo di ereditarietà, che è simile a quello dei linguaggi di programmazione orientati agli oggetti.

```
1 abstractClass Machine {
2     | connect
3         -> compromise
4     & compromise
5         -> channels.transmit
6 }
7 class Hardware extends Machine {}
8 class Software extends Machine {
9     & compromise
10    -> channels.transmit,
11        executors.connect
12 }
```

Nell'esempio, la classe `Machine` è astratta, e non può essere istanziata. Essa viene specializzata nelle classi concrete `Hardware` e `Software`. In particolare, la classe `Software` eredita tutti gli step di attacco e le associazioni. Inoltre, la classe `Software` effettua l'override dello step di attacco `compromise`, aggiungendo un nuovo step di attacco figlio, `executors.connect`.

Il Meta Attack Language

Tempo di compromissione

Alcuni step di attacco possono essere completati senza nessuno sforzo da parte dell'attaccante. Tuttavia, per altri step di attacco, l'attaccante deve effettuare un certo numero di tentativi prima di riuscire. Questo implica del tempo necessario per completare lo step di attacco. Ad esempio, per crackare una password con un attacco a dizionario, potrebbero essere necessarie 18 ore. Per questo motivo, MAL fornisce un meccanismo per specificare il tempo necessario per completare uno step di attacco.

```
1 class Credentials {  
2     & dictionaryCrack [18.0]  
3 }
```

Certe volte, però, non siamo a conoscenza del tempo esatto necessario per completare uno step di attacco. In questo caso, possiamo far uso di distribuzioni di probabilità.

```
1 class Credentials {  
2     & dictionaryCrack [GammaDistribution(1.5, 15)]  
3 }
```

Il Meta Attack Language

Difese

Abbiamo visto che le classi possono prevedere delle difese, che rappresentano le azioni che possono essere intraprese per prevenire l'attacco. Tecnicamente, ogni difesa include una fase di attacco. Se la difesa è falsa, allora, al momento dell'istanza, il passaggio di attacco associato è contrassegnato come compromesso.

Ad esempio, le credenziali possono essere crittate, e quindi non possono essere compromesse con un attacco a dizionario.

```
1  class Credentials {  
2      | access  
3      -> compromiseUnencrypted  
4      & compromiseUnencrypted  
5      # encrypted  
6      -> compromiseUnencrypted  
7  }
```

- Se `Credentials.encrypted` è falso, l'attaccante sarà in grado di raggiungere il passo di attacco `compromiseUnencrypted` non appena ha raggiunto `access`.
- Se, invece, `Credentials.encrypted` è vero, allora `compromiseUnencrypted` non sarà raggiunto, poiché la sua compromissione richiede quella di entrambi i genitori.

Esempio

- Classe [astratta]: `Machine`
 - Specializzazioni: `Hardware` e `Software`
 - Le macchine possono eseguire Software; ad esempio, una workstation può eseguire un sistema operativo, che a sua volta può eseguire un'applicazione.
 - Due o più macchine possono essere collegate da un `Channel`
 - Ad esempio, un software per browser web può essere collegato a un software server web tramite un canale https.
- Classe: `Credentials`
 - Le istanze possono essere, ad esempio, nomi utente e password o chiavi private.
 - Le credenziali hanno destinazioni (`Machine`), per le quali fungono da autenticazione e possono essere memorizzate sulle macchine.

```
1 abstractClass Machine {  
2     | connect  
3         -> compromise  
4     | authenticate  
5         -> compromise  
6     & compromise  
7         -> _machineCompromise  
8     | _machineCompromise  
9         -> executees.compromise,  
10            storedCreds.access,  
11            channels.transmit
```

Esempio (cont.)

```
1 abstractClass Machine {  
2     | connect  
3     | -> compromise  
4     | authenticate  
5     |     -> compromise  
6     & compromise  
7     | -> _machineCompromise  
8     | _machineCompromise  
9     | -> executees.compromise,  
10    |     storedCreds.access,  
11    |     channels.transmit  
12 }
```

Considerando gli step di attacco, la classe `Machine` ha due dei quattro step di attacco:

- `compromise` indica che l'attaccante ha ottenuto il controllo della macchina
 - Per raggiungere `compromise`, sia `connect` che `authenticate` devono essere compromessi
- `connect` rappresenta lo stabilimento di una connessione con la `Machine` da parte dell'attaccante
- `authenticate` rappresenta l'autenticazione dell'attaccante sulla `Machine`, tramite le credenziali

Se `compromise` viene raggiunto, allora tutti i `Software` che sono eseguiti sulla `Machine` saranno compromessi. Inoltre, le credenziali memorizzate sulla `Machine` saranno compromesse, e le connessioni sui `Channel` con altre `Machine` saranno compromesse.

Esempio (cont.)

```
1 class Hardware extends Machine {  
2 }  
3  
4 class Software extends Machine {  
5     & compromise  
6     -> _machineCompromise,  
7         executors.connect  
8 }
```

La specializzazione `Hardware` non ha alcuno step di attacco aggiuntivo. Invece, la specializzazione `Software` ha un solo step di attacco aggiuntivo: se un `Software` viene compromesso, allora anche il `Software` che lo esegue (`executor`) sarà compromesso.

Ad esempio, se un'applicazione viene compromessa, allora anche il sistema operativo che l'esegue sarà compromesso.

Esempio (cont.)

```
1 class Channel {  
2     | transmit  
3     -> parties.connect  
4 }
```

La classe `Channel` ha un solo step di attacco, `transmit`. Se `transmit` viene compromesso, allora tutte le `Machine` coinvolte nella comunicazione (`parties`) saranno compromesse.

Esempio (cont.)

```
1  class Credentials {
2    | access
3      -> compromiseUnencrypted,
4        dictionaryCrack
5    & compromiseUnencrypted
6      -> compromise
7    | dictionaryCrack [GammaDistribution(1.5, 15)]
8      -> compromise
9    | compromise
10     -> targets.authenticate
11   # encrypted
12     -> compromiseUnencrypted
13 }
```

La classe `Credentials` ha quattro step di attacco. Compromettendo `Machine`, l'attaccante ha accesso alle credenziali. Tuttavia, non è detto che queste siano automaticamente compromesse, poiché potrebbero essere crittate. Infatti, abbiamo qui una *difesa*, `encrypted`.

Infatti

- Se `encrypted` è vero, allora `compromiseUnencrypted` non sarà raggiunto, poiché la sua compromissione richiede quella di entrambi i genitori.
- Se `encrypted` è falso *AND* l'attaccante ha raggiunto lo step di attacco `access`, allora `compromiseUnencrypted` sarà raggiunto, e di conseguenza `compromise`.

Esempio (cont.)

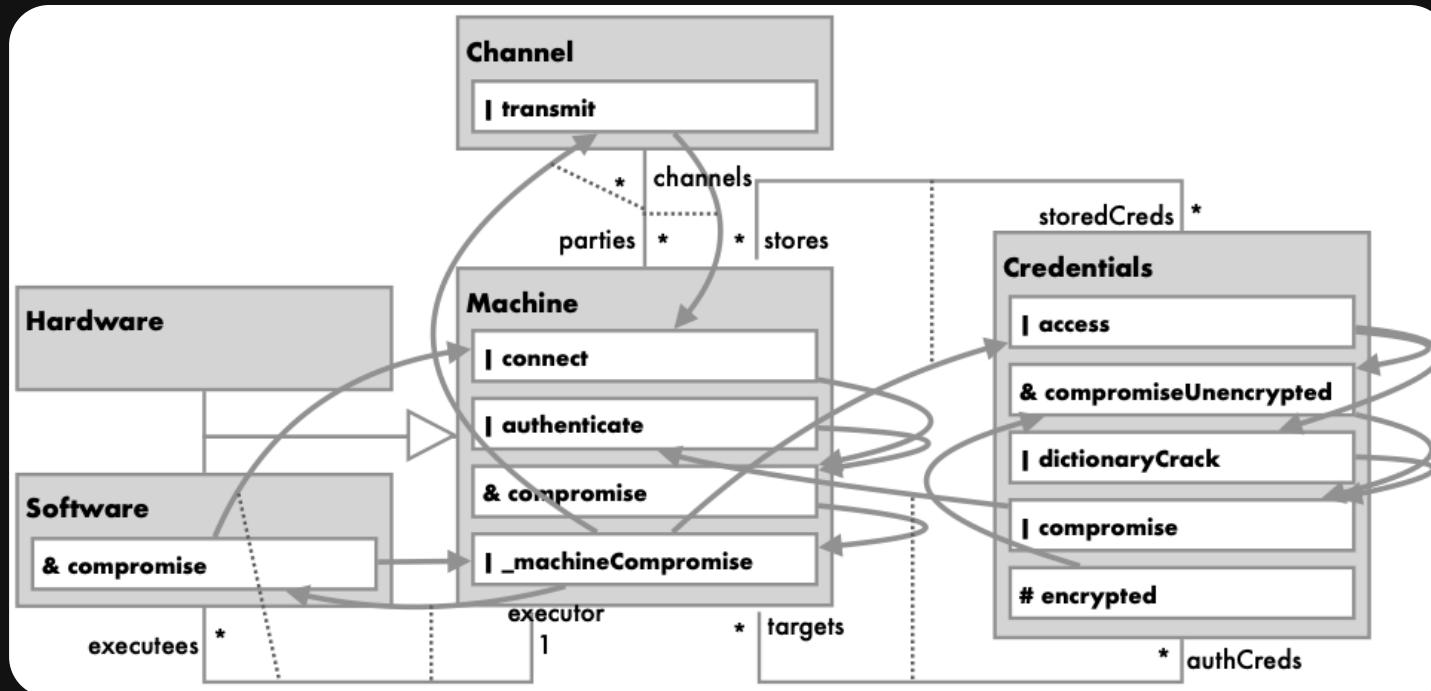
```
1  class Credentials {
2    | access
3      -> compromiseUnencrypted,
4          dictionaryCrack
5    & compromiseUnencrypted
6      -> compromise
7    | dictionaryCrack [GammaDistribution(1.5, 15)]
8      -> compromise
9    | compromise
10       -> targets.authenticate
11    # encrypted
12       -> compromiseUnencrypted
13 }
```

Tuttavia, c'è un'altra possibilità: l'attacco a dizionario (``dictionaryCrack``).

Questo attacco richiede che venga speso del *tempo* per essere portato a termine. Tale intervallo richiesto è modellato tramite una distribuzione di probabilità, in questo caso una *distribuzione gamma*.

Esempio (cont.)

Specification diagram



Esempio (cont.)

Istanziazione

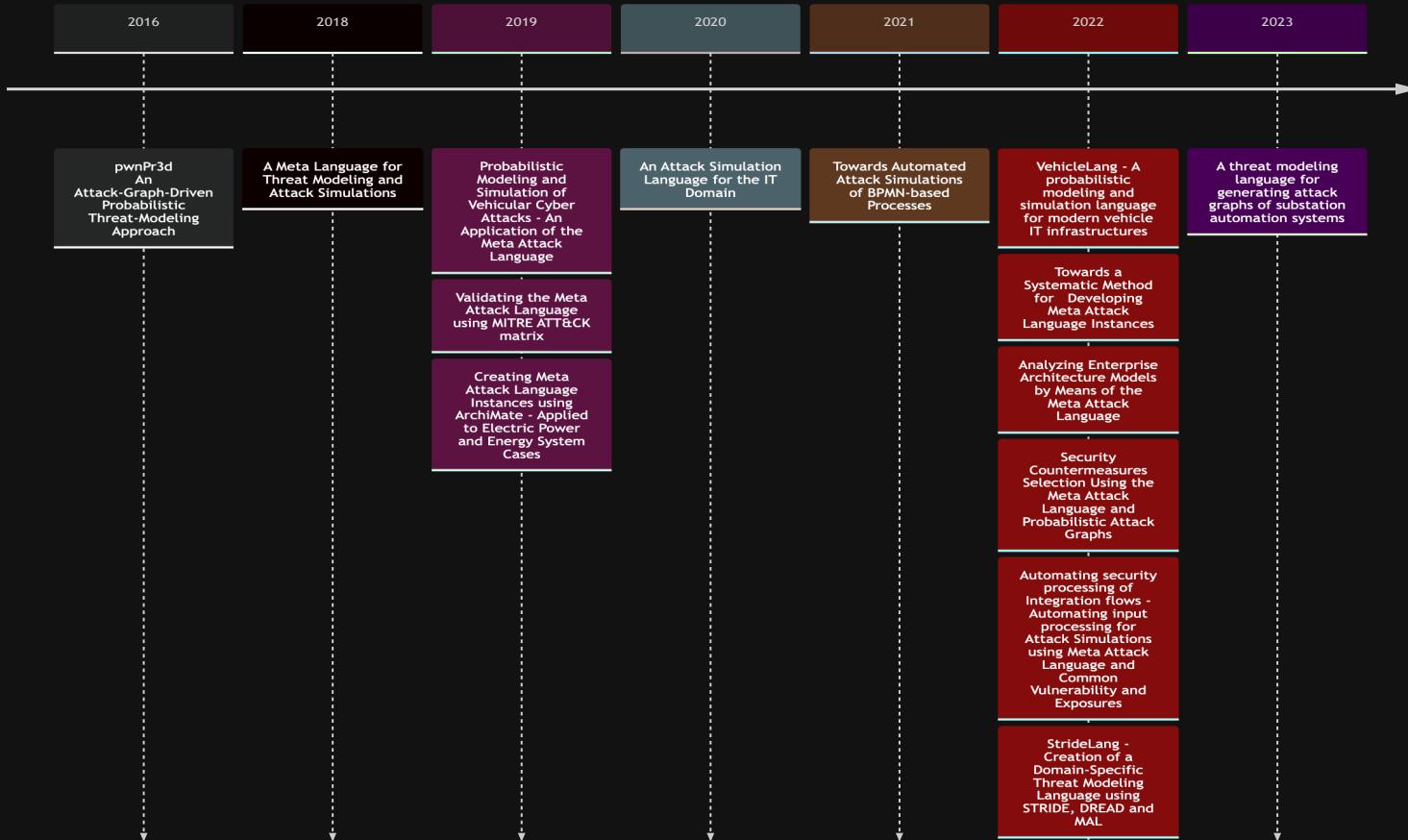
Dalla specifica, possiamo istanziare, in maniera automatica, le classi Java corrispondenti.

```
1  Hardware macBook = new Hardware();
2  Software macOS = new Software();
3  Software sshClient = new Software();
4  Credentials sshKey = new Credentials (encrypted=Bernoulli(0.5));
5
6  macOS.addExecutor(macBook);
7  sshClient.addExecutor(macOS);
8  macOS.addStoredCreds(sshKeys);
9
10 Hardware system76 = new Hardware();
11 Software ubuntu = new Software();
12 Software sshDaemon = new Software();
13
14 ubuntu.addExecutor(system76);
15 sshDaemon.addExecutor(ubuntu);
16 sshKey.addTarget(sshDaemon);
17 sshKey.addTarget(ubuntu16);
18
```

MAL

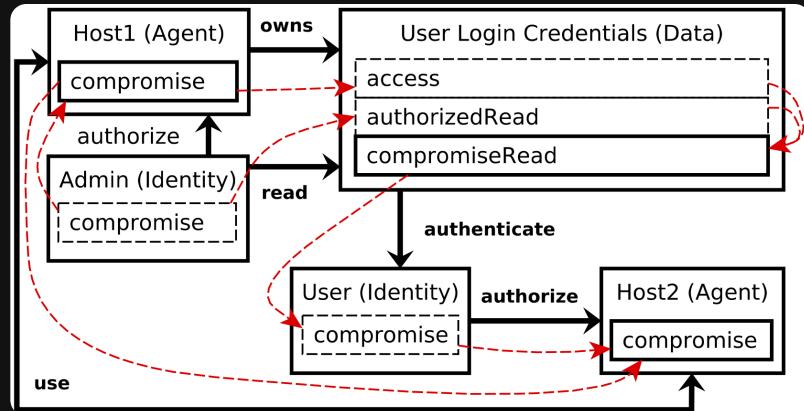
Stato dell'arte

Evoluzione di MAL



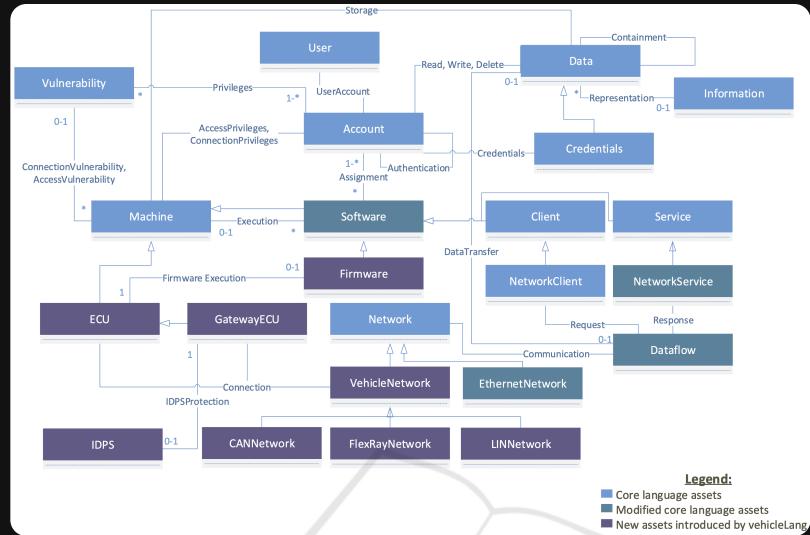
pwnPr3d

- Il paper presenta pwnPr3d, un sistema di threat modeling basato sulla modellazione della rete, che fornisce un approccio olistico alla sicurezza informatica ed è in grado di generare automaticamente grafi di attacco, con distribuzioni di probabilità sul Time To Compromise (TTC) degli asset del sistema.
- Si tratta di un precursore di MAL.
- pwnPr3d ha un'architettura di meta-modellazione chiusa, che promuove la riutilizzabilità degli elementi del modello e la personalizzazione delle esigenze specifiche.
- Il paper definisce i grafi di attacco, il calcolo del TTC e presenta esempi di modellazione. La lingua di modellizzazione è flessibile e può essere personalizzata per soddisfare esigenze specifiche.
- pwnPr3d copre la maggior parte della classificazione STRIDE tranne la repudiation, attualmente in fase di sviluppo.



VehicleLang

- Costruito su MAL e coreLang, VehicleLang è un linguaggio di modellazione e simulazione probabilistico per le infrastrutture IT dei veicoli moderni.
- Sono stati aggiunti nuovi elementi per modellare le caratteristiche specifiche dei veicoli:
 - I componenti di rete CAN
 - Le comunicazioni wireless
 - ECU (Electronic Control Unit)
 - LIN (Local Interconnect Network)
 - FlexRay
- Di questo paper ho letto anche una versione più aggiornata, che non aggiungeva nulla di nuovo rispetto a quanto già visto.



Legend:

- Core language assets
- Modified core language assets
- New assets introduced by vehicleLang

attackLang

- Si tratta di un linguaggio di modellazione e simulazione degli attacchi per il dominio IT, basandosi su MITRE ATT&CK.
- È stata scelta una tecnica di attacco per ciascuna colonna del MITRE ATT&CK Matrix ed è stata implementata in attackLang.
- Java e jUnit sono stati utilizzati per implementare il generatore di codice e Maven per la compilazione e l'esecuzione dei test.

Initial Access	Execution	Persistence	Privilege Escalation
Drive-by Compromise	AppleScript	.bash_profile and .bashrc	Access Token Manipulation
Exploit Public-Facing Application	CMSTP	Accessibility Features	Accessibility Features
Hardware Additions	Command-Line Interface	Account Manipulation	AppCert DLLs
Replication Through Removable Media	Compiled HTML File	AppCert DLLs	AppInit DLLs
Spearphishing Attachment	Control Panel Items	Appinit DLLs	Application Shimming
Spearphishing Link	Dynamic Data Exchange	Application Shimming	Bypass User Account Control
Spearphishing via Service	Execution through API	Authentication Package	DLL Search Order Hijacking

Figura 1: MITRE ATT&CK Matrix extract

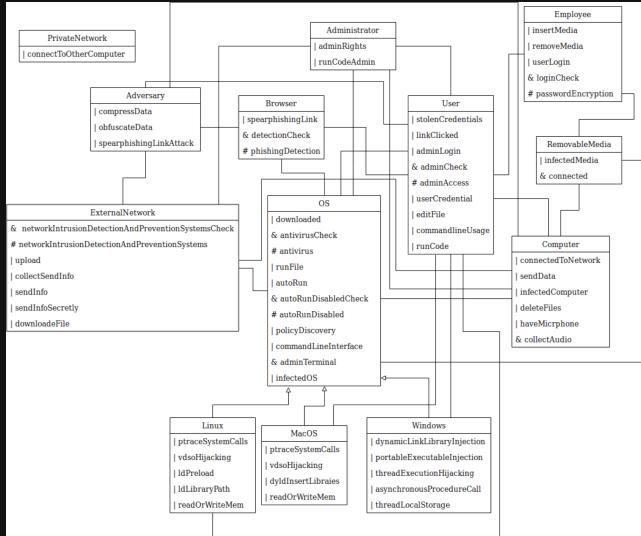


Figura 2: Graphical meta model for attackLang

attackLang (Cont.)

Attack graph [1]

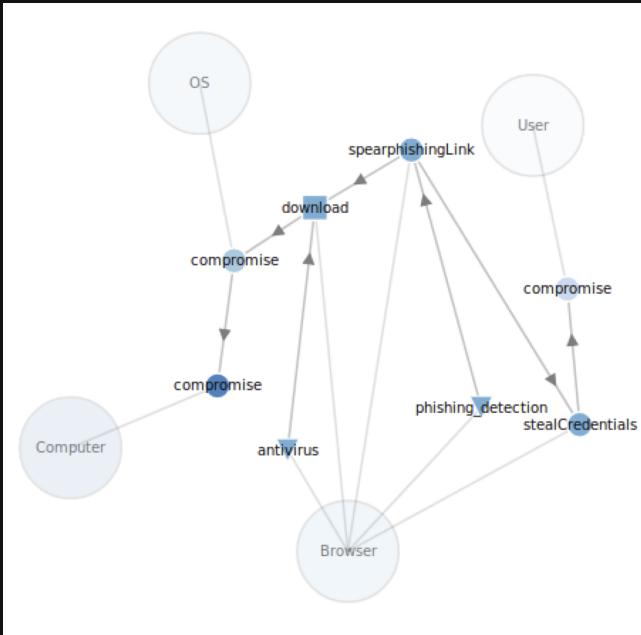


Figura 3: Attack graph of the individual spearphishingLink file

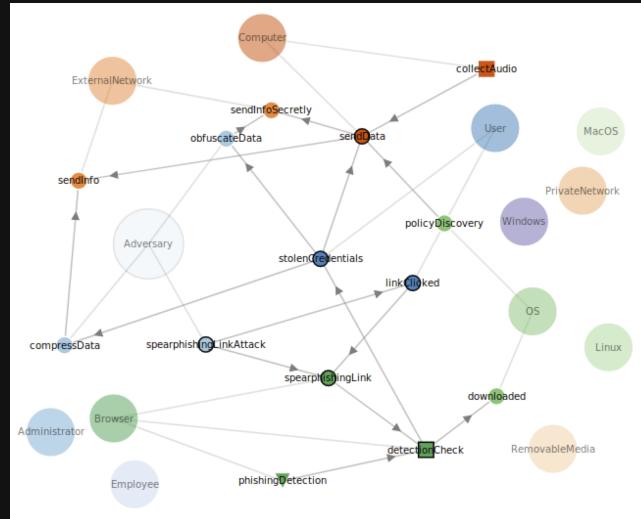


Figura 4: Attack graph of the complete attackLang (Not all connections)

1. Non è stato specificato come ottenere questi attack graph a partire dal codice generato da attackLang.

Creating Meta Attack Language Instances using ArchiMate

- Il paper presenta un metodo per creare istanze di MAL a partire da modelli ArchiMate.
- Sono stati analizzati due casi d'uso:
 - Un sistema di controllo di una centrale termoelettrica;
 - La rete elettrica ucraiana.

ArchiMate è un linguaggio di modellazione dell'architettura aziendale aperto e indipendente per supportare la descrizione, l'analisi e la visualizzazione dell'architettura all'interno e tra i domini aziendali in modo inequivocabile. ArchiMate si distingue da altri linguaggi come Unified Modeling Language (UML) e Business Process Modeling and Notation (BPMN) per il suo ambito di modellazione aziendale.

Inoltre, UML e BPMN sono pensati per un uso specifico e sono piuttosto pesanti - contenenti circa 150 (UML) e 250 (BPMN) concetti di modellazione mentre ArchiMate funziona con poco meno di 50 (nella versione 2.0). L'obiettivo di ArchiMate è quello di essere "il più piccolo possibile", non di coprire ogni scenario marginale immaginabile.

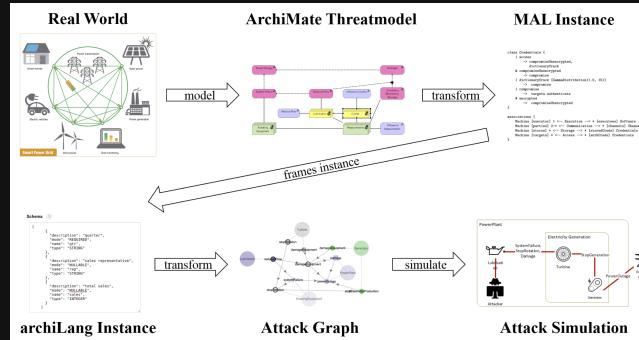


Figura 5: Transformation process

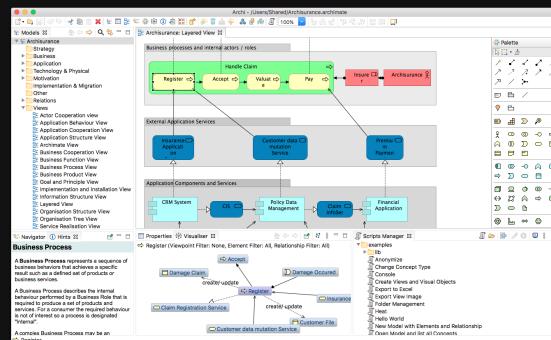


Figura 6: ArchiMate Tool

Automated Attack Simulations of BPMN-based Processes

- Il paper mostra le mancanze di BPMN dal punto di vista della sicurezza.
 - Ad esempio, in Figura 7 è riportato un BPMN in cui ci sono diverse vulnerabilità, che non erano ovvie al momento della creazione del diagramma stesso.
- Le soluzioni già esistenti, secondo gli autori, sono limitative, e.g.
 - SecureBPMN [1]
 - SecBPMN [2]
- Come soluzione, viene proposto un mapping tra le entità di BPMN e quelle di coreLang, linguaggio basato su MAL che contiene tutti i concetti generali dell'IT. In sostanza vengono trasformati i processi concreti del BPMN in modelli di attacco, che possono essere simulati con tool come securiCAD [3].

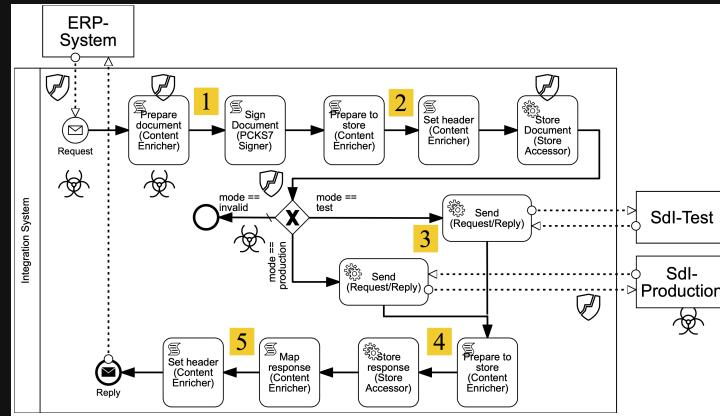


Figura 7: Esempio di BPMN

1. SecureBPMN è un DSL che consente di modellare gli aspetti di sicurezza. Esso è definito come un meta-modello che può essere facilmente integrato in BPMN e, quindi, può essere utilizzato per modellare processi aziendali e sicuri e composizioni di servizi sicure.

Automated Attack Simulations of BPMN-based Processes (Cont.)

BPMN2MAL

- Per valutare l'approccio indicato, è stato implementato un prototipo di BPMN2MAL.
- Il processo adottato è rappresentato in Figura 9. In sostanza non si lavora direttamente sul BPMN ma su un modello grafico generato a partire da esso.
- Viene generato un grafo a partire dal BPMN.
- Per ogni elemento all'interno del BPMN vengono generate diverse configurazioni, con le librerie e le versioni previste.
- Questo elenco viene utilizzato per effettuare una ricerca nel repository del NIST, per trovare le vulnerabilità associate.
- Le informazioni trovate vengono inserite nel grafo.
- I grafi vengono poi trasformati in modelli securiCAD, che sono istanze di coreLang, secondo i mapping mostrati in Figura 8.
- Per ciascun modello securiCAD vengono eseguiti diversi test di simulazione, fornendo all'attaccante diverse superfici di attacco.
- I risultati vengono poi restituiti nella rappresentazione grafica, per visualizzare le attività critiche nel modello BPMN.

		TABLE III MAPPING FROM BPMN TO CORELANG					
BPMN	Sub	Data	Credentials	Identity	User	Application	Connection
Collaboration	Participant / Process Sub-Process Event Sub-Process					✓	✓
Event	Message Start Start Timer Start Condition Message End	✓				✓	✓
Activity	User Task Script Task Service Task Send Task Receive Task Manual Task Business Rule Task Call Activity	✓	✓	✓	✓	✓	✓
Data	Data object Data Store Message	✓				✓	✓
Connecting	Sequence Flow (conditional) Message Flow					✓	✓

Figura 8: BPMN2MAL

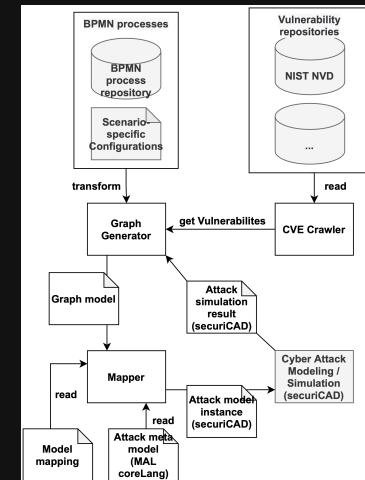


Figura 9: Prototipo dell'architettura

Security countermeasures selection using the Meta Attack Language and Probabilistic Attack Graphs

- 1. Introduzione:** Il paper presenta un metodo per selezionare contromisure di sicurezza efficienti nelle infrastrutture IT utilizzando il Meta Attack Language (MAL) e i grafi di attacco probabilistici.
- 2. Lavoro correlato:** Gli autori discutono gli approcci esistenti all'analisi del rischio e alla selezione delle contromisure di sicurezza, evidenziando l'importanza di avere un modello completo del sistema e metriche di sicurezza pertinenti.
- 3. Metodologia:** Gli autori descrivono il loro approccio, che prevede la modellizzazione dell'infrastruttura IT utilizzando MAL e la generazione di grafi di attacco probabilistici per identificare vulnerabilità e possibili percorsi di attacco.
- 4. Validazione empirica:** Gli autori forniscono i risultati degli esperimenti condotti su varie infrastrutture IT per dimostrare l'efficacia del loro approccio nella selezione di contromisure di sicurezza efficienti.
- 5. Conclusioni:** Gli autori concludono che il loro metodo può aiutare le organizzazioni a proteggere gli asset critici delle infrastrutture da potenziali attacchi informatici identificando vulnerabilità e suggerendo le appropriate contromisure.

Complessivamente, questo paper fornisce spunti su come le organizzazioni possano utilizzare MAL e i grafi di attacco probabilistici per migliorare la propria posizione in materia di sicurezza informatica.

Automating input processing for Attack Simulations using Meta Attack Language and Common Vulnerability and Exposures

- Si tratta di un'estensione del lavoro presentato in Automated Attack Simulations of BPMN-based Processes
- Lo scopo della tesi è quello di automatizzare un'attività precedentemente manuale costruendo sul lavoro precedente svolto nel progetto.
- Una rappresentazione coreLang MAL di un sistema viene istanziata, creando un mapping dai punti di attacco utilizzando l'input dell'integration flow^[1] e collegandoli a vulnerabilità ed exploit rilevanti, raccolti dal crawler NVD già presente. Questa mappatura deve essere utilizzata come input per le simulazioni di attacco eseguite in securiCAD.

securiCAD

- Il modo in cui securiCAD genera i suoi grafici di attacco è attraverso una libreria di passaggi di attacco fissa e un DSL chiamato **securiLang**.
- Le simulazioni di attacco in securiCAD si basano su simulazioni probabilistiche su grafici di attacco simili alle reti bayesiane, in combinazione con la ricerca sulla sicurezza.
- Per ogni risorsa del modello per il quale vengono simulati gli attacchi, il tempo medio in cui l'attività deve essere compromessa a un certo tipo di attacco è determinato in base a quali misure difensive sono state messe in atto per la particolare risorsa. È lo stesso principio di base di MAL!

1. Per integration flow si intende un modello BPMN-based che mostra come le diverse parti di un business interagiscono tra loro. 

StrideLang: Creation of a Domain-Specific Threat Modeling Language using STRIDE, DREAD and MAL

- Il lavoro di tesi è volto a realizzare un nuovo DSL per il threat modeling, chiamato **StrideLang**.
- È intuitibile che questo linguaggio sia basato su STRIDE.
- Più precisamente, strideLang è stato costruito andando a mappare i concetti di STRIDE su coreLang, un DSL basato su MAL per la descrizione del dominio IT.
- In strideLang sono presenti sei asset, relativi alle sei categorie di STRIDE; ciascuna di esse può contenere fino a tre attack step, ciascuno dei quali corrisponde a una categoria di rischio DREAD (low, medium, high).

```
1  asset Spoofing
2  user info : "In this type of attacks the perpetrator appears as someone that can be trusted"
3  {
4      | highRiskSpoofing ->
5          credentials.attemptCredentialsReuse
6      | mediumRiskSpoofing ->
7          identity.assume,
8          application.specificAccessAuthenticate,
9          application.authenticate,
10         networks.accessNetworkData //manInTheMiddle
11     }
```

sasLang

- Il paper presenta un nuovo DSL per la descrizione di un *substation automation system* [1], chiamato **sasLang**.
- Questo DSL è stato costruito basandosi su coreLang e icsLang, un DSL per la descrizione di sistemi di controllo industriale.
- Il criterio adottato è il seguente:
 - Caso 1: L'asset esiste in icsLang o coreLang → Nessun nuovo asset viene aggiunto a sasLang.
 - Caso 2: L'asset è un tipo di asset descritto in icsLang o coreLang → Estendi l'asset generico icsLang per creare un nuovo asset specifico in sasLang.
 - Caso 3: L'asset non esiste in icsLang o coreLang → Crea un nuovo asset e le relative associazioni dell'asset in sasLang.

Esempio

```
1  asset LogicalNode extends IcsApplication
2  {
3      | manipulationOfControl
4      +> equipment.manipulationOfControl,
5          actuator.manipulate
6  }
```

Il simbolo `+>` indica che il nuovo step di attacco non sovrascrive il vecchio.

1. Un substation automation system è un sistema di controllo che monitora e controlla le apparecchiature elettriche in una stazione elettrica. 