



Horizon 2020 Program
Dynamic countering of cyber-attacks
SU-ICT-2018



Cyber security 4.0: Protecting the Industrial Internet of Things

D3.2: Mitigation Engine[†]

Abstract:

This deliverable presents the first output of Task 3.3 ("Mitigation engine"): it describes the overall structure of the mitigation engine; its main components: a binary code analyser, a software-defined networking controller, and a central brain performing the analysis of possible reconfiguration based on the available inputs; the interactions internal and external to the mitigation engine, especially with close C4IIoT components, like the Cloud Layer Orchestrator.

Contractual Date of Delivery	31/05/2020
Actual Date of Delivery	30/05/2020
Deliverable Security Class	Public
Editor	<i>M Lemerre, S Bardin (CEA)</i>
Contributors	CEA, UP1PS, TSG, HPE, FORTH

^{††††} *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 833828.*

Quality Assurance	<i>Reviews by VIP and IFAG</i>
-------------------	--------------------------------

The C4IIoT Consortium

FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS	Coordinator	EL
CENTRO RICERCHÉ FIAT SCPA	Principal Contractor	IT
INFINEON TECHNOLOGIES AG	Principal Contractor	DE
THALES SIX GTS FRANCE SAS	Principal Contractor	FR
HEWLETT PACKARD ITALIANA SRL	Principal Contractor	IT
COMMISSARIAT A L ENERGIE ATOMIQUE ET AUX ENERGIES ALTERNATIVES	Principal Contractor	FR
IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD	Principal Contractor	IL
AEGIS IT RESEARCH UG	Principal Contractor	DE
UNIVERSITE PARIS I PANTHEON- SORBONNE	Principal Contractor	FR
INFORMATION TECHNOLOGY FOR MARKET LEADERSHIP	Principal Contractor	EL
SPHYNX TECHNOLOGY SOLUTIONS AG	Principal Contractor	CH
UNIVERSITY OF NOVI SAD FACULTY OF SCIENCES	Principal Contractor	SRB
UNIVERSITY OF GREENWICH	Principal Contractor	UK
VIP MOBILE D.O.O.	Principal Contractor	SRB

Document Revisions & Quality Assurance

Internal Reviewers

1. *Bojan Kovacevic, Dragan Danilovic, Zoran Gajica (VIP)*
2. *Antonio Javier Cabrera Gutierrez (IFAG)*

Revisions

Version	Date	By	Overview
0.1	23/03/2020	CEA	First version of the ToC
0.2	26/03/2020	CEA	Second version of the ToC
0.3,0.4		HPE,UP1PS,TSG,CEA	Initial version of the contribution
0.5	24/04/2020	CEA	Integrated version
0.6	08/05/2020	HPE,CEA	Updated contributions
0.7	08/05/2020	UP1PS	Updated contribution from UP1PS
0.8	08/05/2020	TSG	Updated contribution from TSG
0.8.2	14/05/2020	CEA	Integration + version sent for internal review
0.9	28/05/2020	HPE,TSG,UP1PS,CEA	Updates to address review from IFAG,VIP

Table of Contents

LIST OF FIGURES	5
LIST OF ABBREVIATIONS.....	6
EXECUTIVE SUMMARY	8
1 INTRODUCTION	9
2 FACTORY INFRASTRUCTURE TO PROTECT	10
3 MITIGATION ENGINE ARCHITECTURE WITHIN C4IIOT FRAMEWORK	13
4 VARIAMOS MITIGATION PLAN SEARCH COMPONENT	16
4.1 ROLE OF VARIAMOS IN THE MITIGATION ENGINE	16
4.2 INPUT TO VARIAMOS IN C4IIoT	18
4.3 OUTPUT OF VARIAMOS IN C4IIoT	22
4.4 TECHNOLOGIES USED BY VARIAMOS	25
4.5 INTERNAL ARCHITECTURE OF VARIAMOS	29
4.6 VARIAMOS PROTOTYPE.....	29
5 DISCO SOFTWARE DEFINED NETWORK CONTROLLER COMPONENT	31
5.1 INTERFACES	32
5.1.1 Northbound	32
5.1.2 Southbound.....	32
5.1.3 East/Westbound.....	33
5.2 ACTIONS	34
5.3 ROLE OF DISCO IN THE MITIGATION ENGINE	34
5.4 INPUT TO DISCO IN C4IIoT	34
5.5 OUTPUT OF DISCO IN C4IIoT.....	35
5.6 TECHNOLOGIES USED BY DISCO.....	35
6 HPE CONTAINER ORCHESTRATOR (HPECO) COMPONENT	37
6.1 ROLE OF CLOUD LAYER ORCHESTRATOR IN THREAT MITIGATION.....	37
6.2 INPUT FROM THE MITIGATION ENGINE TO CLOUD LAYER ORCHESTRATOR.....	37
6.3 OUTPUT FROM CLOUD LAYER ORCHESTRATOR TO THE MITIGATION ENGINE	37
6.4 TECHNOLOGIES USED BY CLOUD LAYER ORCHESTRATOR.....	37
7 BINSEC FACTORY APP BINARY CODE ANALYSER COMPONENT	39
7.1 BINSEC OVERVIEW	39
7.2 BINSEC ARCHITECTURE AND COMPONENTS	40
7.2.1 Loaders.....	40
7.2.2 Instruction decoders.....	41
7.2.3 Formal analyses	41
7.3 ADAPTATIONS FOR THE C4IIOT FRAMEWORK	42
7.3.1 Needs for C4IIoT.....	42
7.3.2 New components being developed.....	43
7.4 PRELIMINARY STUDIES ABOUT THE DEVELOPMENT OF THE NEW COVERAGE-GUIDED TESTING AND PATCH-ORIENTED TESTING ENGINES OF BINSEC	43
7.4.1 Deep coverage-guided testing for automating low-level security analysis	43
7.4.2 Directed fuzzing for incremental and targeted analysis and patch-oriented testing.....	44
7.4.3 Evaluation of possible solutions.....	44
8 CONCLUSION	49
9 REFERENCES	50

List of Figures

Figure 1: Model of a typical factory network to be protected by the C4IIoT framework	10
Figure 2: Mitigation Engine Architecture with C4IIoT Framework	15
Figure 3: VariaMos Reactive Cyberattack Mitigation Ontology	17
Figure 4: A view of VariaMos' Asset Taxonomy.....	18
Figure 5: A view of VariaMos' High-Level Threat taxonomy	21
Figure 6: Elements of a DOS Attack instantiated from the Mitigation Ontology	21
Figure 7: A view of VariaMos' mitigation action taxonomy.....	23
Figure 8: A possible mitigation plan for a DOS attack instantiated from the mitigation action taxonomy	24
Figure 9: A second possible mitigation plan for a DOS attack instantiated from the mitigation action taxonomy	25
Figure 10: COOLP paradigm and internal architecture of VariaMos	27
Figure 11: A sample interaction with the VariaMos Prototype.....	30
Figure 12 - DISCO Controller Architecture	32
Figure 13 - DISCO Controller architecture for C4IIoT.....	33
Figure 14: Comparison between the reactive and proactive setups.....	35
Figure 15: BINSEC architecture and updates	40
Figure 16: The DBA formal intermediate language.....	41
Figure 17: Comparison between Confuzz and AFL.....	44
Figure 18: Success rates of directed fuzzers.....	44
Figure 19: Importance of symbolic execution on the LAVA benchmark	45
Figure 20: Importance of symbolic execution on a synthetic benchmark	46
Figure 21: Shortcomings of current symbolic execution and fuzzing integration.....	48

List of Abbreviations

AEGIS	AEGIS IT Research UG
AFL	American Fuzzy Lop
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
AVG	Autonomous Guided Vehicle
C4IIoT	Cyber security 4.0: protecting the Industrial Internet Of Things
CEA	Commissariat A l'Energie Atomique et aux energies alternatives
CGC	Cyber Grand Challenge
CLO	Cloud Layer Orchestrator
CLP	Constraint Logic Programming
CP	Constraint Programming
CRF	Centro Ricerche Fiat
CWA	Close-World Assumption
DOS	Denial of Service
DSL	Domain Specific Language
EC	European Commission
ERP	Enterprise Resource Planning
FORTH	Foundation for Research and Technology
HPE	Hewlett Packard Italiana
IBM	IBM Israel - Science and Technology LTD
IFAG	Infineon Technologies AG
IIoT	Industrial Internet of Things
ILP	Inductive Logic Programming
IPDS	Intrusion Prevention and Detection Systems
ITML	Information Technology for Market Leadership
IoT	Internet of Things
KB	Knowledge Base
KRL	Knowledge Representation Language
LP	Logic Programming
ME	Mitigation Engine
MES	Manufacturing Execution System
MITM	Man In The Middle
NBI	NorthBound Interface
OCLP	Object-oriented Constraint Logic Programming

OLP	Object-oriented Logic Programming
OOP	Object-Oriented Programming
OS	Operating System
OWA	Open-World Assumption
PATDA	Privacy-Aware Trustworthy Data and Analytics
SAM	Security Assurance Module
SDN	Software-Defined Network
SE	Symbolic Execution
SIEM	Security Information and Event Management
STS	Sphynx Technology Solutions
TSG	Thales SIX GTS
UAF	Use-after-free
UNPMF	University of Novi Sad Faculty of Sciences
UOG	University of Greenwich
UP1PS	Université Paris I Pantheon-Sorbonne
VIP	VIP Mobile Doo Beograd (Novi Beograd)
WP	Work Package

Executive Summary

This deliverable presents the first output of Task 3.3 ("Mitigation engine"): it describes the overall structure of the mitigation engine; its main components: a binary code analyser, a software-defined networking controller, and a central brain performing the analysis of possible reconfiguration based on the available inputs; the interactions internal and external to the mitigation engine, especially with close C4IIoT components, like the Cloud Layer Orchestrator.

1 Introduction

This deliverable is the first output of Task 3.3 ("Mitigation engine"), a central C4IIoT component which provides the core building block for mitigating attacks across the different layers (edge node layer, field gateway layer, and cloud layer) of C4IIoT. It is thus a central component of an end-to-end integrated industrial IoT cybersecurity framework.

The activities of Task 3.3 are built around three main building blocks:

- A binary code analyser, used to analyse the firmware and software updates that run on the edge and cloud layer; this binary code analyser is built upon CEA's BINSEC tool;
- A software defined networking (SDN) controller, used to filter traffic destined to vulnerable or unpatched devices, so as to keep devices functional but safe by blocking or limiting traffic that contains malicious signatures or suspicious patterns; this SDN controller is built upon TSG Disco SDN Controller;
- The mitigation engine brain, that triggers automatic actions based on input received by all other C4IIoT components, and automatically reconfigure the system to mitigate attacks, thus allowing the mitigation engine to be self-adaptive. The brain logic is implemented using UPIPS's VariaMos technology.

Running on the cloud, these components also make easy use of the HPE's Cloud Layer Orchestrator as a gateway to interact with the other components of the C4IIoT platform.

The document is structured as follows:

- Section 2 describes the factory structure to protect; this factory example is the main use case used to test the mitigation engine concepts;
- Section 3 describes the mitigation engine global architecture, the interaction between the mitigation engine components and with the other C4IIoT components;
- Section 4 describes the mitigation engine brain and its implementation based on VariaMos;
- Section 5 describes the mitigation engine interaction with the network built upon the Disco SDN controller;
- Section 6 summarizes important aspects of the Cloud Layer Orchestrator which is the main gateway through which the mitigation engine interacts;
- Section 7 describes the mitigation engine interaction with binary code, built upon the BINSEC binary code analyser;
- Section 8 provides concluding remarks.

This model is needed since the inputs and outputs used in the mitigation engine's demonstrator, such as those shown in sections 4.2 and 4.3, will contain instances of these classes. The IIoT factory infrastructure is divided in two layers: the edge layer that comprises the IIoT devices deployed on the plant floor, and the factory field layer containing the local servers deployed in the office space of factory. For simplification purposes, we assume here, following what was specified in deliverable D1.2 that C4IIoT will focus on protecting three main classes of IIoT devices: (a) assembly line robot controllers, such as welding robots, (b) ***Autonomous Guided Vehicle (AVG)***, self-driving cart carrying containers around the shop floor following trails painted on it, and (c) mobile devices such as tablets, handheld by factory workers. We also assume that the C4IIoT protection includes dedicated anomaly detection software installed on these three classes of edge devices.

The controllers of all these devices are remotely orchestrated by ***Manufacturing Execution Systems (MES)*** running on servers deployed on the office premise of the factory. This remote orchestration is carried out through three types of connections: (a) SDN controlled Profinet connections between the MES and the assembly line robots, (b) wifi connections between the AVG and mobile devices and SDN controlled access points and (c) SDN controlled Ethernet connections between the MES and the wifi access points. In order for this communication and control infrastructure to be minimally attack tolerant, we assume here that some redundancy is implemented for its critical components: the Profinet connections, wifi connections, wifi access points and the MES. This is why these elements have a multiplicity of two to three in the diagram of Figure 1. This way, if one instance of each of them goes down following an attack, such an attack can be mitigated by the availability of other running instances and alternative connections between edge and field devices and so that factory infrastructure production throughput can gracefully degrade instead of coming to a full halt.

The MES serves as a gateway between the edge and factory field layers. In order for the various factory field applications and connections to be attack tolerant, redundant instances of each of them must be deployed. In addition, all connections must be SDN controlled. Almost all connections between field layer applications is carried out through and SDN controlled Ethernet connection. The one exception to this rule is the connection between the MES and the plant data repository management system that centralizes all the data gathered from various sources to be accessible for various data analytics purposes including production monitoring, preventive maintenance and cybersecurity. This connection uses the POC-UA Net protocol.

The MES executes high-level production goals defined and updated daily by the ***Enterprise Resource Planning (ERP)*** software to which it is connected. Both the MES and the ERP are themselves connected to a production management user-interface. This interface is also connected to an on-premises preventive maintenance data analytics application that leverages data stored in the plant data repository. A key component of this repository is the Historian that aggregates raw transactional data logs into historical data pre-processed for analytics purposes. The repository is connected to a system administration user interface.

The interface is also connected to the ***Security Information and Event Management (SIEM)*** system in charge of supervising cybersecurity monitoring. This SIEM is connected to ***the Intrusion Prevention and Detection Systems (IPDS)*** that detects traffic anomaly that can be cues to the occurrence of an intrusion inside the factory infrastructure. These anomalies are presented to the incident response team through a user interface. The SIEM and IPDS are two common components in charge of improving cybersecurity in IIoT infrastructures. They will thus typically constitute a cybersecurity protection baseline that C4IIoT will complement to bring a higher level of protection. The IPDS will thus be connected with the C4IIoT factory field anomaly detection component which will provide a second layer of anomaly detection at the field layer, building upon the results of the edge layer detection installed on the controllers

of the assembly line robots, AVG and factory worker tablets. This C4IIoT second layer is connected to the MEDICI anomaly detection off-loader. This component decides what data and analysis to offload to the third cloud located layer of anomaly detection when it realizes that the computing power needed to provide a more accurate detection is superior to that available at the edge and factory field layer. It then sends the appropriate data and analysis request to the cloud through the SDN controlled internet gateway that connects the factory field layer intranet to the internet.

3 Mitigation Engine Architecture within C4IIoT Framework

The *Mitigation Engine (ME)* internal architecture and its interactions with the other components of the C4IIoT framework are shown in Figure 2. This figure is a partial refinement and update of the C4IIoT framework architecture from Figure 1 of deliverable D1.3 that focuses on the components related to reactive cyberattack mitigation.

Shown at the centre top of Figure 2, the ME is composed of three main internal components: VariaMos, DISCO and BINSEC. The first is an automated reasoning component that reacts to a cyberattack alert associated with the estimated negative business impact that it will cause if not mitigated.

This alert is a representation of a detected **attack plan** containing the **attack actions** that are suspected to be occurring (these two concepts are detailed further below in section 4.1). Each suspicion comes with its probability, since attack detection is always an uncertain process with some irreducible residual proportion of false positives and false negatives. This impact-annotated alert input to the ME and VariaMos is aggregated by two other C4IIoT components from three sources themselves generated by three other C4IIoT components. This aggregation process is shown on the top left of Figure 2. It starts by *ITML Privacy-Aware Trustworthy Data and Analytics (PATDA)* component that derives cyberattack alerts from two complementary traffic anomaly detector components:

- One provided by FORTH which relies on matching anomalous patterns hand-written by cybersecurity experts;
- One provided by UNSPMF which relies on semi-supervised machine learning.

STS Security Assurance Module (SAM) then annotates the cyberattack alert derived by PATDA from these two sources with a risk model built following a methodology provided by HPE. This model associates a negative business impact with each attack, by estimating the cost of the damage that it can provoke if left unmitigated.

Given an impact-annotated attack alert, VariaMos searches the space of possible mitigation plans to find a **plan list** ordered by descending estimated negative business impact of the attack tolerant factory degraded working mode resulting from applying the plan. These plans are analogous to the aforementioned **attack plans** insofar as they are also composed of **mitigation actions** aiming to mitigate the detected **attack actions** (this is likewise further detailed in section 4.1).

The actions of such plans can be applied in three main ways:

1. By DISCO-controlled factory network traffic rerouting (network reconfiguration);
2. By uninstalling application containers running on factory network host nodes suspected to be compromised, followed by installing alternative application containers providing the same service with a different technology stack that have passed some security verification (application container reconfiguration)
3. By a factory worker physically changing some physical elements in the factory, including sensing, computing, networking and actuating devices, and/or update the firmware and OS kernel sitting below the Docker container manager in the technology stack running at each factory network host.

The application container reconfiguration actions of a chosen mitigation plan can be automated by issuing commands to the *Cloud Layer Orchestrator (CLO)* component of C4IIoT. Since cybersecurity mitigation is a business-critical decision, VariaMos is not an autonomous artificial intelligence, but rather an augmented intelligence. It thus merely proposes mitigation plans to a human cybersecurity officer that makes the final decision.

To do so, this officer interacts with the *AEGIS cybersecurity dashboard*, shown on the top-right of Figure 2, to visualize VariaMos suggestions and either:

- Pick one of them;
- Manually input through the dashboard a different plan not among those suggested by VariaMos;
- Simply ignore the attack alert estimating that it is either a false positive, or that any feasible mitigation would have worse consequences than the attack itself.

When the officer chooses one of the plans suggested by VariaMos, the latter then calls DISCO and CLO to implement the actions in the chosen plan that can be automated. It calls DISCO through an intent-based *NorthBound Interface (NBI)* that abstracts the physical infrastructure of the network switches controlled by DISCO to allow VariaMos to reason exclusively at the application layer and solely specify desired connectivity constraint changes among applications running at network hosts.

To control changes in the technological stack on top of which these applications are running, VariaMos feeds an updated factory app deployment model to CLO. To realize this model, CLO first accesses the factory app container registry manager to find the container images to be installed on network host suspected of having been compromised at a layer above the container manager.

For each new container pushed to the Private Image Registry, CLO then calls a container image file analyser to run a scan at the level of container image. For the C4IIoT context only container images with predefined low levels of vulnerabilities will be enabled to be deployed, so that only verified containers will be able to replace the suspected compromised ones.

CLO manages only Container images entities as a whole, therefore executable files in ARM or other formats, need to be checked by BINSEC after the build phase of the SW development cycle, before the docker image containing that executable is created and be pushed to the C4IIoT private image registry.

To control changes in the technological stack on top of which these applications are running, VariaMos feeds an updated factory app deployment model to CLO. To realize this model, CLO retrieves docker images from its private registry (where they have been previously checked) and deploys them instead of the compromised ones.

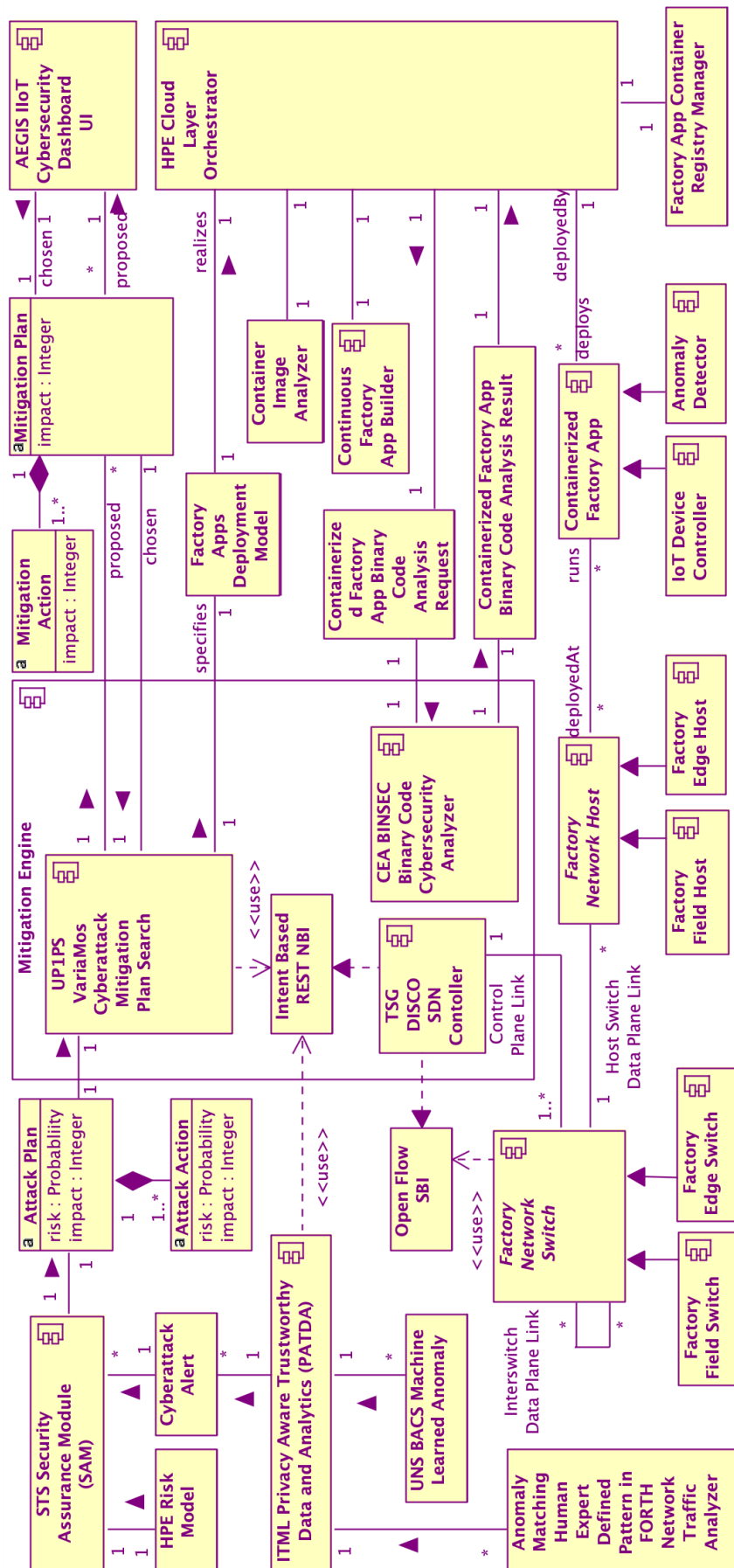


Figure 2: Mitigation Engine Architecture with C4IIoT Framework

4 VariaMos mitigation plan search component

4.1 Role of VariaMos in the mitigation engine

The role of VariaMos in the mitigation engine is to search the space of possible plans to mitigate the impact of a suspected cyberattack. To perform this search task, VariaMos requires an internal representation of the domain over which it is reasoning. This has led to the creation of a reactive cyberattack mitigation ontology shown as a UML class diagram in Figure 3. To ease integration with the **Security Assurance Module (SAM)** of the C4IIoT framework provided by project partner STS, which will provide the main input to the mitigation engine and VariaMos, we included the concepts encapsulated in the SAM as a starting point for this ontology, though we have renamed some of them. These SAM inspired concepts are those displayed with a white background in Figure 3.

Though a strong starting point, the concepts from SAM were not enough since this tool focuses on static preventive cybersecurity threats and countermeasures, whereas the mitigation engine and VariaMos address dynamic, reactive threats and countermeasures. To handle this dynamicity and reactivity, VariaMos must reason in terms of how an attack actually plays out over time. To support such reasoning, we have added our own concepts, highlighted with a yellow background, in Figure 3. The first important ones are the concepts of **attack goals, plans and actions**. They together capture the notion that an attack occurs in several discrete action steps that each have their own preconditions and possible effects and exploit particular vulnerabilities in the system. An **attack plan** is an actual materialization of a **threat** that can target the system. It models how the attacker intends to carry out its attack by exploiting, in sequence, several vulnerabilities of the system. These concepts allow us to view advanced attack detection as a *plan recognition task* (Sohrabi, Riabov, & Udrea, 2016): how to infer the high-level intention of a malicious actor that manifests itself through a sequence of low-level attack actions that are individually detectable through network traffic or host behavior anomalies. This attack will then be found to be violating one (or more) instances of the key **dependability properties** that are the key cybersecurity properties that must be maintained for an adequate operation of the system. These properties have been derived from (Knight, 2012).

The key insight in our ontology is that in order to establish an adequate defensive strategy, a **mitigation plan** must “mirror” the progression of the **attack plan** and formulate a collection of **mitigation actions** that stop the **attack actions** being (or having been) carried out. To this end, we postulate that in order to defeat these malicious actions, we must retaliate with **mitigation actions** that are designed to either **contain, eradicate or recover from** these malicious actions as part of the **incident response** lifecycle as laid out in (Cichonski, Millar, Grance, & Scarfone, 2012). In addition to this, we can state that the search performed over the space of possible plans must have a particular optimization criterion, which in this model is represented by the overall **risk** (expressed as a probability inherent to both **attack actions** and **attack plans**) caused by the particular threat. These attack actions are either inferred through **pattern recognition** if they have been provided by the system’s detection components or through **plan recognition** if they have been determined logically from the observed effects on the system.

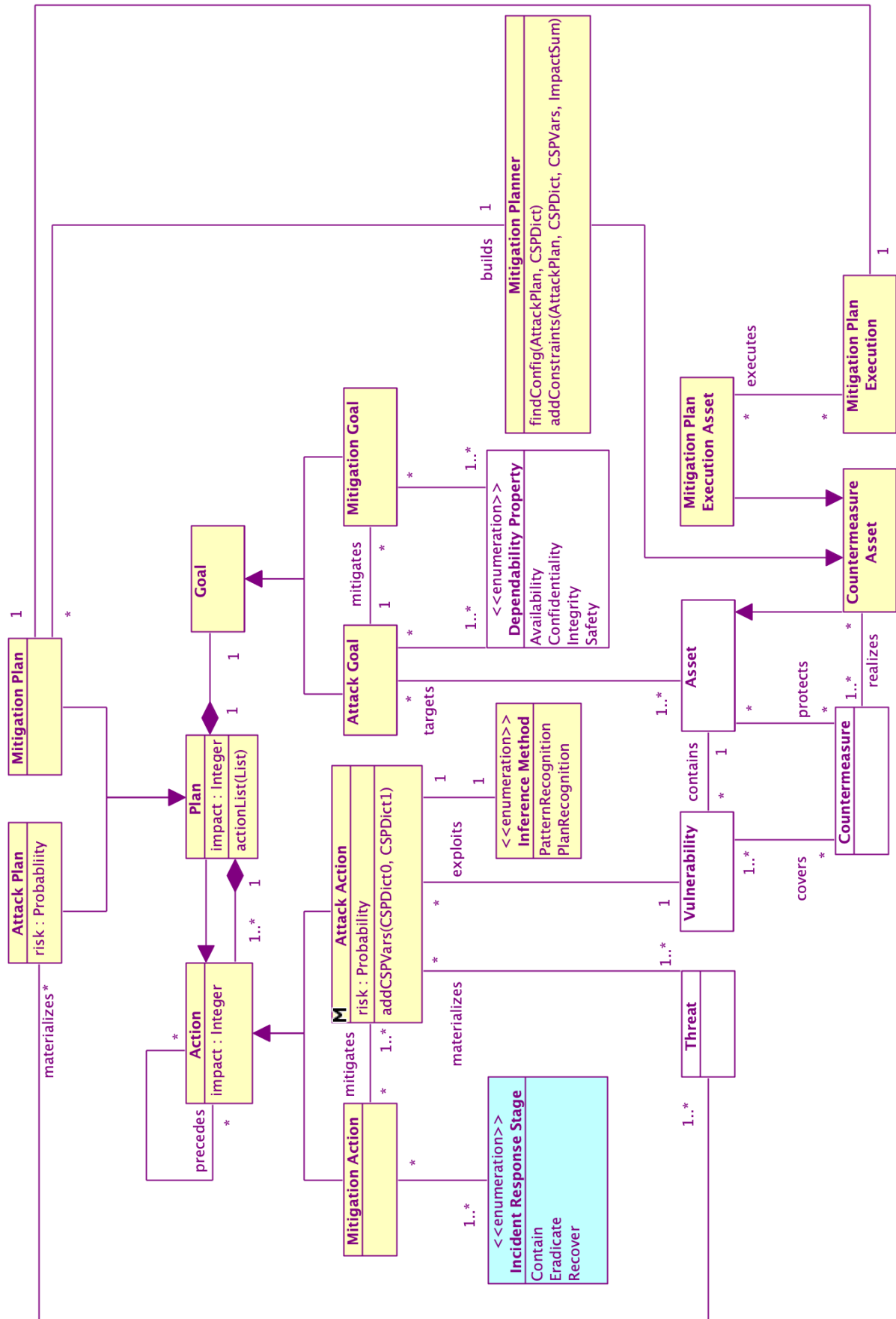


Figure 3: VariaMos Reactive Cyberattack Mitigation Ontology

In terms of our internal representation of the system, we have enriched the ontology shown in Figure 3 with an asset taxonomy (shown in Figure 4) that will aid in the calculation of both the **risk** and the possible **impact** posed by any particular threat. This taxonomy has been developed from the **Security Assurance Module's** internal model, and includes several additions based on the factory use case's description. It is of particular note that our reasoning observes assets as belonging to one of two particular classes: **hardware assets**, which are essentially any physical computational entity that is part of (or interacts with) the factory's systems; and, **software assets**, which are the actual files and programs that are executed on the aforementioned **hardware assets**. With this categorization in mind, it is worth noting that among the **hardware assets** we make a clear distinction among two different subclasses: **network devices**, which are charged with both organizing the network and providing access to it (wherein you find elements such as **switches and access points**); and, **network hosts**, which are **assets** that either provide or consume **network exposed services**. This distinction is fundamental insofar as it determines both the degree of actions that may be performed on each of these assets and the purpose that each is determined to fill within the frame of the overall system.

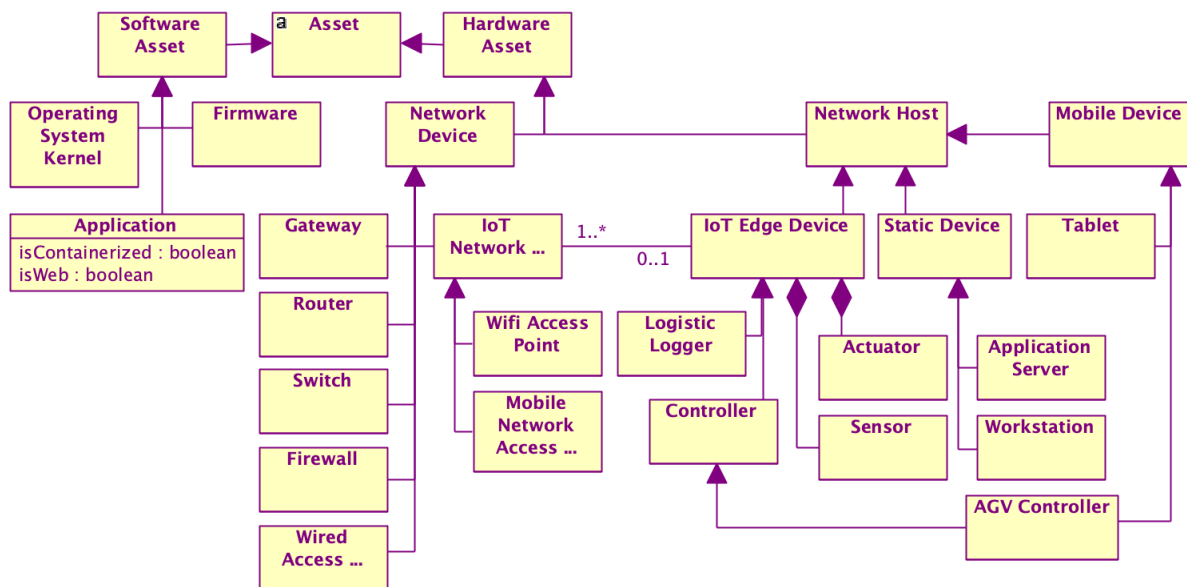


Figure 4: A view of VariaMos' Asset Taxonomy

4.2 Input to VariaMos in C4IIoT

VariaMos takes two inputs. The first is a cyberattack alert annotated with the negative business impact that the attack will cause if left unmitigated. The second input comes from the factory cybersecurity officer through the AEGIS cybersecurity dashboard: it is the mitigation plan that the officer has chosen among the list of plans found by VariaMos and proposed to the officer. This plan consists of network connectivity reconfiguration actions and/or containerized application reinstallation on network hosts.

In order to correctly interpret the detected attack as reported, we developed a **threat taxonomy** adapted from (Hindy, et al., 2018) and shown as a UML class diagram in Figure 5. This categorization is important insofar as it will allow the detected threats to be adequately put in context, and, crucially, to correctly specify the defensive actions to be taken, for, evidently,

every threat must be dealt with in a unique manner (despite the fact that some threats do, in fact, have a certain degree of similarity). As mentioned in the previous section, the conceptual representation has been developed in conformity with STS's **SAM conceptual model**.

This **taxonomy** categorizes possible threats into four main categories:

- **Network threats:** these types of threats aim to affect the communications layer of the underlying system.
- **Host threats:** these threats emerge from the presence of malicious software entities within the assets of the underlying system.
- **Software threats:** these threats seek to remotely interfere with the normal operation of a network accessible software platform (e.g. to perform code injection).
- **Other threats:** these threats arise from either human error, malicious insiders seeking to obtain undue access (or otherwise disturb the systems to which they have access) or physical tampering of assets.

Among the main **network threats** modeled in VariaMos, we find instances of (**Note:** The following definitions in quotes are cited from (Hindy, et al., 2018)):

- **Denial of Service (DOS):** “[...] where an attacker floods the network with requests rendering the service unresponsive.”
- **Packet Forging:** “is the action in which the attacker generates packets that look the same as those of the network. These packets can be used to perform certain action, steal information, etc.”
- **Man in the Middle (MITM):** “When the attacker intercepts communications between two or more entities and starts to either control the communication between them and alter the communication or listen to the network [...]”
- **Impersonation:** “[...] pretending to be another user [...]”
- **Network reconnaissance:** “[where] the attacker [searches] the network for information such as, active nodes, the running operating system, software versions, etc.”

Network threats can then emerge among a number of possible **system layers** represented as an enumeration associated to the network threat class in Figure 5. They are based on the classification of the **TCP/IP Layers** (also known as the **Internet protocol suite**):

- **Network Access Layer (Link Layer):** This layer corresponds to the lowest level of the communications between assets. Attacks at this layer target the actual communications infrastructure (e.g. ARP tables or MAC addresses).
- **Internet Layer:** This layer corresponds to the source and destination handling protocols in the communication infrastructure. Attacks at this layer aim to disrupt or discover the assets present in the communications infrastructure.
- **Transport Layer:** This layer corresponds to the actual information transfer protocols and the actual handling of connections between clients and servers. Attacks at this layer aim to manipulate the operations of these protocol to generate adverse effects and behaviors on hosts.
- **Application Layer** This layer corresponds to the applications that either generate or receive the network traffic, and thus attacks at this layer aim to either generate incorrect behaviors or exploit flaws in their programming.

Among the main **host threats** modeled in VariaMos we find:

- **Malware:** in its various forms it can be defined as “computer software specifically designed to perform malicious or unwanted actions” (Whitman & Mattord, 2017), of which several subtypes can be enumerated (e.g. trojans, worms, etc.) and has the characteristic of having as an initial entry point the host within which it is executed.
- **Privilege escalation:** “The unauthorized modification of an authorized or unauthorized system user account to gain advanced access and control over system resources [...]” (Whitman & Mattord, 2017)

Pertaining to **Software threats**, we model several different subtypes, which are of particular importance given the fact that several critical elements in the factory use case’s architecture expose network accessible applications. Among these instances, we find:

- **Code Injection:** “an application error that occurs when user input is passed directly to a compiler or interpreter without screening for content that may disrupt or compromise the intended function.” (Whitman & Mattord, 2017) This can be particularly disastrous when it can affect the operations of one of the critical application servers.
- **Misconfiguration:** Configuration errors in deployed application servers can leave them vulnerable to exploitation.
- **Drive-by downloads:** This threat refers to the possibility that simple web navigation by mobile devices or workstation can lead to inadvertent installation of malicious software that can act as an attacker’s initial foothold in the system.

Among threats posed to the **humans** that interact with the system, we may find:

- **Phishing:** Phishing attacks are social engineering attacks that can allow an attacker to lead a user astray to either a malicious site where it can initiate a **drive-by download** or the tainted communications can itself contain malicious attached files that can, as before, be an initial foothold into the system.

Finally, in terms of **physical threats** that may affect the system we can find:

- **Physical Tampering:** Physical tampering of devices can allow attackers to manipulate them at will and allow perhaps the installation of malicious code or changing its operating parameters.

VariaMos expects to receive observations about the elements constituting an attack that have been detected by the different detection systems deployed throughout the system; with these observations, assembled into a **partial attack plan**, it will be possible to first construct and inferred **attack plan** (as it is illustratively shown in Figure 6). It is these **attack actions**, targeting specific **assets** that will determine how VariaMos will formulate a defensive stance to mitigate these adverse events. These different detected elements of an attack will allow VariaMos to determine the progression of the detected attack. The probabilistic nature of attack detection is accounted for in the probabilities (expressed as a **risk**) that are derived both for **attack actions** and **attack plans**. Furthermore, to a quantitative impact, expressed here in euros, constitutes a key attribute of each **attack action (and plan)**, for it is in this way that the implied “cost” of the adverse event is calculated. In addition, a key element in the deduction of the sequence of adverse events is the notion of **precedence** among the **attack actions**, for it is in this manner that it will be possible to reconstruct the progression of events in the attack, and, above all, respond to it in its entirety. Each of these **attack actions** can then be seen as a manifestation of particular **threat** in a complex multi-stage attack. It is evident that merely addressing one symptom of an ongoing **attack** is insufficient to adequately put an end to it; thus, it is for this reason that, as will be shown in the next subsection, the primordial goal of

this formal representation is to allow the formulation of a **mitigation plan** that can adequately address each part of the **attack plan** and thus allow the **recovery** of all affected assets.

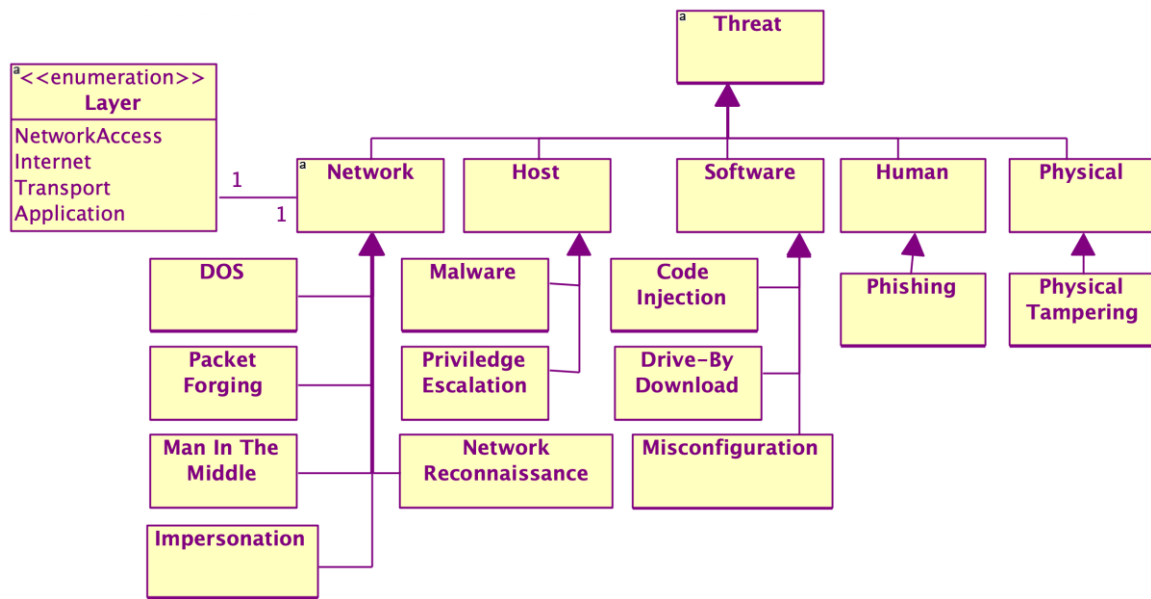


Figure 5: A view of VariaMos' High-Level Threat taxonomy

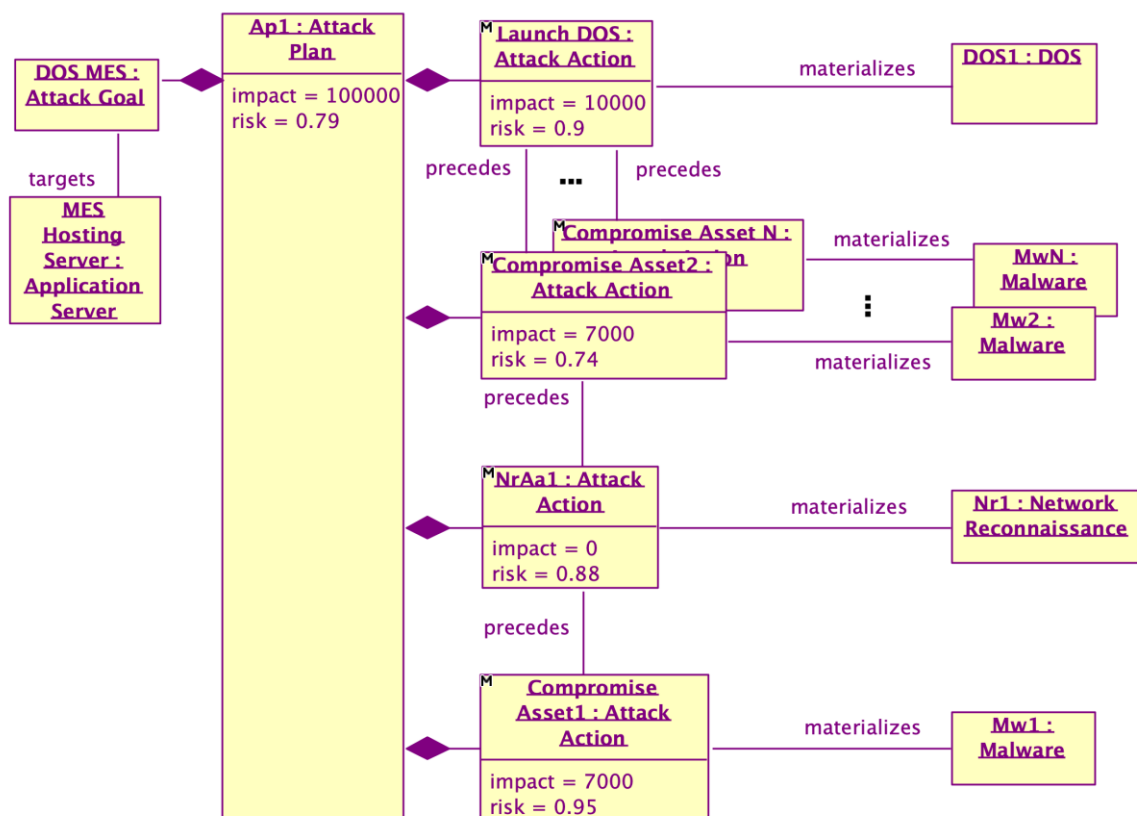


Figure 6: Elements of a DOS Attack instantiated from the Mitigation Ontology

Now, pertaining to the second type of input that VariaMos is to receive, it is in essence the choice that the operator has made according to the recommendations set forth by VariaMos,

which will then, if applicable, set in motion VariaMos' interaction with either the SDN Controller or HPE's Cloud Layer Orchestrator. As detailed in section 3 this will either consist of instructions for modifying the disposition of the network; changes to the factory's application deployment model, which is the basis of HPE's Cloud Layer Orchestrator; or actions that ought to be performed by a human operator (in which case such interactions would not be necessary as they would fall outside the scope of what VariaMos can interact with). This is further detailed in the next subsection.

4.3 Output of VariaMos in C4IIoT

VariaMos produces three outputs. The first is a list of attack mitigation plans ordered by descending estimated negative business impact of the attack tolerant factory degraded working mode resulting from applying the plan. The second is the set of connectivity reconfiguration actions of the mitigation plan chosen by the cybersecurity officer, among factory network hosts running the factory applications. These actions are carried out by **DISCO**. The third output is the set of container substitution actions in the chosen mitigation plan at hosts running applications suspected by the attack alert to have been compromised. These actions are performed through interaction with **CLO**.

Figure 8 and Figure 9 demonstrate partial (and illustrative) views of **mitigation plans** for a certain attack scenario (in this case a DOS attack directed at the Manufacturing Execution System (MES)) that can be set in motion to mitigate different stages of the given attack. What it illustrates, then, is a partial view of the results of searching the **search space of mitigation actions** to arrive at a suitable mitigation plan definition, chaining several possible defensive actions that have been inferred as useful to mitigate the different stages of the attack. There is a direct link then between what these actions do, and the attack actions, for they are precisely designed to **mitigate** what has occurred (or is occurring) at different stages of an attack. With this being said, it must be recognized that the more complete and well defined the attack alert input is, the better and more precise the output can be. These **mitigation actions**, and therefore the **search space** that will be analysed, are constructed from the basis of a **mitigation action taxonomy** shown in Figure 7. This taxonomy has been constructed to correspond to the three basic pillars of action that VariaMos has at its disposal:

- **Network Traffic Rerouting Actions:** These actions correspond to manipulations of the network configuration through interaction with **DISCO**.
- **Container Replacement Actions:** These actions correspond to manipulations of the **factory application deployment model** that manages the actions of the **container orchestrator**.
- **Factory Worker Actions:** These actions will necessitate some form of external interaction with the affected systems as they are beyond the reach of the mitigation and its associated components.

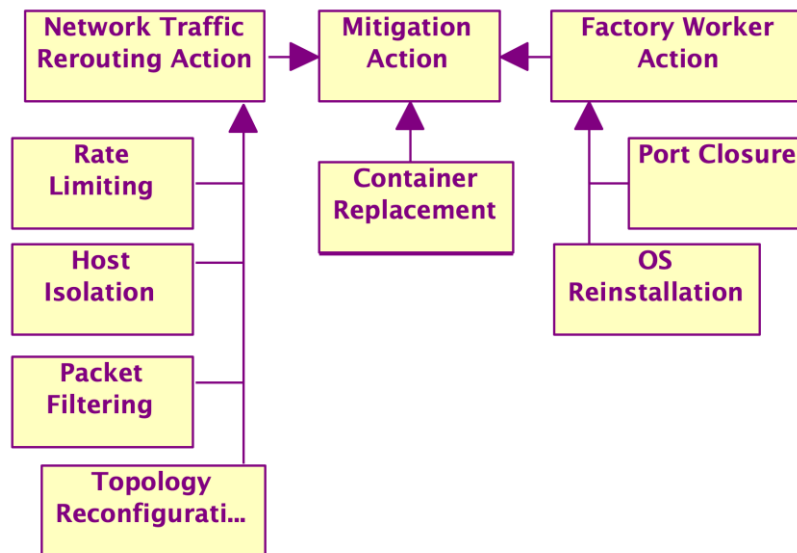


Figure 7: A view of VariaMos' mitigation action taxonomy

As an illustrative example, if we examine the elements of a possible **DOS attack** as depicted in Figure 6, we can state that, for instance, the “**Launch DOS**” **attack action** can be mitigated by a **Host isolation** action, which would stop all communication emerging from the source of the attack (as shown in Figure 9). Alternatively, this attack action could be stopped by a **combination** of the **Rate Limiting** and **Packet Filtering** actions (as shown in Figure 8); this would have the effect of effect of both limiting the network throughput from the malicious nodes and would potentially drop packets with particular unwanted characteristics.

Keeping mind the attack depicted in Figure 6, we can observe that a **compromised host** could, for instance, be addressed by either **Container Redeployment** (if it has had only its memory corrupted or the attack is contained within the bounds of the container) or by the far more drastic **OS Reinstallation action** (as is shown in both mitigation plan examples). This would evidently depend on whether the affected asset has had its process isolation mechanisms broken (i.e. the containers) or they are not available given the type and architecture of the asset. It is precisely these types of judgements that VariaMos must make in order to minimize the effects on the continuity of business whilst simultaneously minimizing the expected residual risk from the suggested actions.

Continuing with the example attack shown in Figure 6, it is shown that **Network Reconnaissance** attack actions have multiple angles of mitigation. From our two example plans, we can observe that they can be stopped by either: performing **Port Closure** on the target nodes so as to render them invisible to such observation; performing **Packet Filtering** to impede packets from addressing unauthorized ports; or, drastically, performing a **topology reconfiguration** to render any information gleaned from reconnaissance obsolete and useless, as the network would then be imbued with a completely different structure.

Taking all of the above into account and turning to the **search** to be performed over the space of possible **mitigation plans**, it is important to mention that the optimisation and subsequent ranking criteria for the mitigation plans is the **impact** that is inherent to each one. This one, in contrast with the impact associated to particular **attack actions** is a representation of the **residual impact** (as shown in Figure 8 and Figure 9) after applying a given mitigation action, and thus the mitigation plan would have its impact calculated from the combination of these residual impacts. After a successful search of this space has been concluded, the aforementioned ranked list of defensive action plans would be the output sent to the operator on the **AEGIS**

cybersecurity dashboard. We expect the operator to analyse and determine the suitability of the proposed action plan in order to be able to engage the other possible outputs. The operator could for instance be provided with the two mitigation plans exemplified, and he would then need to consider the feasibility of the proposed plans with the factory's operational requirements and end up choosing one of them. Then this choice is sent back to VariaMos for subsequent execution. Should he choose to perform modifications, they will be made manifest in the newly arrived input plan. Evidently, VariaMos will only set in motion that which is within its purview. It is this key interaction between VariaMos and the security officer that best illustrates its true purpose, to serve as a decision support tool that generates adequate suggestions for dealing with adverse security incidents, as previously stated.

Turning to the second possible output, and keeping in mind the actions depicted above, for instance, **Packet Filtering** would be an action that would be directed towards the **SDN Controller**. Simply put, the intents that such an operation would produce are high level directives that can be understood by the SDN controller's Northbound Interface and express network actions in a **Domain Specific Language (DSL)** which are then transformed into the actual network operations on the control-plane by the **SDN Controller**. In the case of **Packet Filtering**, this would lead all switches to **drop** all traffic that meets certain desired criteria to impede certain undesirable traffic from moving around the network, as stated above.

In terms of the third output described above, one can see that an action such as **container redeployment** would be an action that would be optimally performed by **CLO**. In practical terms, this would imply changes to the **factory application deployment model**, which would subsequently set in motion the container analysis and redeployment procedures mentioned in section 2.

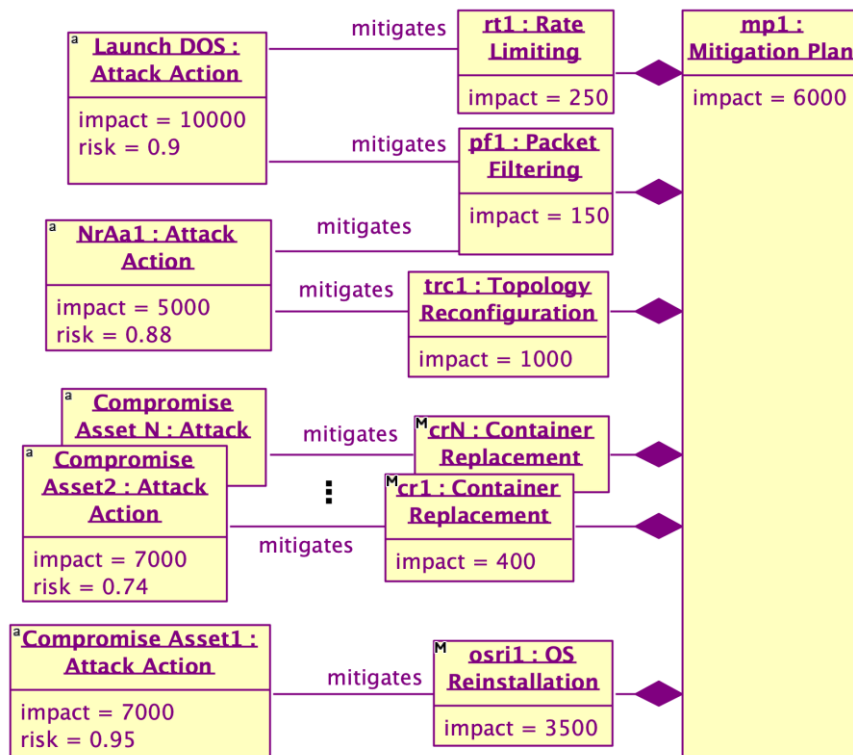


Figure 8: A possible mitigation plan for a DOS attack instantiated from the mitigation action taxonomy

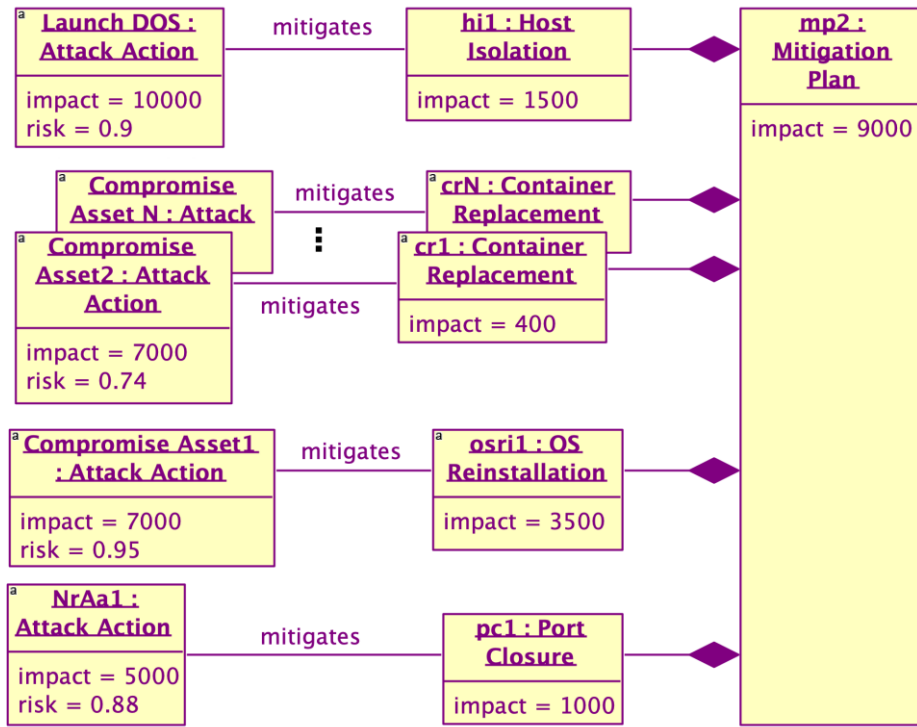


Figure 9: A second possible mitigation plan for a DOS attack instantiated from the mitigation action taxonomy

4.4 Technologies used by VariaMos

VariaMos uses the *Constraint Object-oriented Logic Programming (COOLP)* paradigm. COOLP extends *Logic Programming (LP)* with two orthogonal additional layers of *Constraint Programming (CP)* (Stuckey, Marriott, & Tack) on the one hand, yielding *Constraint Logic Programming (CLP)* (Frühwirth & Abdennadher, 2003), and *Object-Oriented Programming (OOP)* on the other hand, yielding *Object-oriented Logic Programming (OOLP)* (Moura, Logtalk: Design of an Object-Oriented Logic Programming Language, 2003).

LP and its various extensions occupy a unique place in computer science at the intersection of programming and knowledge representation. Indeed, Prolog, the standard LP language is both a Turing-complete programming language, *and, simultaneously, also* a declarative *Knowledge Representation Language (KRL)* (Van Harmelen, Lifschitz, & Porter, 2008) with formal semantics in first-order logic. As any other computer program, a Prolog program, and by extension also an COOLP program, can be either interpreted or compiled into executable code. But as a knowledge base it can also serve as the basis for deductive and abductive reasoning to answer queries supporting straightforward implementation of a variety of *Artificial Intelligence (AI)* tasks. In addition, instead of being manually written, Prolog programs can also be directly machine learned from data using *Inductive Logic Programming (ILP)* (Cropper, Dumancic, & Muggleton, 2020) tools. An LP runtime system encapsulates a search-based inference engine performing deduction and a limited form of abduction under the *Closed-World Assumption (CWA)* (Russell & Norvig, 2020). Such built-in AI reasoning ability is not provided by any other Turing-complete programming paradigm such as imperative, functional or imperative object-oriented programming. On the other hand, other declarative KRL paradigms such as description logics (Van Harmelen, Lifschitz, & Porter, 2008) used to represent ontologies under the *Open-World Assumption (OWA)* (Russell & Norvig, 2020), pure constraint programs (Stuckey, Marriott, & Tack) or Bayesian networks (Russell & Norvig,

2020) are meant to be interpreted by special-purpose inference engines unable to perform arbitrary Turing-complete computation. Therefore, with such KRL any form of algorithmic meta-inference involving sequencing, looping and conditional repeated calls to these engines with different query parameters depending upon each other must be programmed in an extraneous programming language such as Python or Java coupled with a translation bridge between these languages and the KRL. Such extraneous language must also be used for any communication with other software components in a complex system implemented in a technologically heterogeneous architecture such as the C4IIoT framework. In contrast, LP allows uniformly writing the declarative knowledge bases, meta-inference programs and the integration middleware all in a single language.

A key limitation of pure LP is that it is only able to declaratively represent *symbolic* knowledge and reason about it using first-order logic term unification and exhaustive backtracking search. Such general-purpose search strategy does not scale for many combinatorial and optimization problems that require more specialized scalable search strategies. CLP extends LP to additionally allow *declaratively representing numerical knowledge* over various mathematical domains such as bounded integers (also called finite domains), unbounded integers, rational numbers, reals, arrays, bit vectors and others. A CLP(X) engine provides built-in exhaustive and/or heuristic search components able to efficiently solve most cases of constraint systems relating mathematical expressions with variable over domain X. It also seamlessly integrates its X-specific search components with the general-purpose symbolic search provided by the Prolog engine by using user-transparent co-routing between them.

Another limitation of pure LP is that it provides very little concepts supporting both software reuse and structuring large and complex software. Since the best concepts for such support such as encapsulation, composition and inheritance come from object-oriented programming, OOLP incorporates them and adapts them from the imperative algorithmic substrate of traditional object-oriented languages to the declarative, state-change free substrate of LP. In addition, since OOP has become by far the most popular general-purpose programming paradigm, incorporating an AI programmed in pure LP, which is relational in nature, with external OOP software creates an impedance mismatch similar to the one between relational databases and OOP application software¹ OOLP is an elegant and conceptually sound solution to overcome this mismatch. Logtalk (Moura, The Logtalk Handbook, 2020), a *porte-manteau* word for proLOG and smallTALK since it mainly integrates the powerful, complementary concepts of these two languages, is the most complete and actively maintained OOLP language and inference engine to date.

How COOLP integrates LP, with CP and OOP is explained by the components and classes with white backgrounds in the UML diagram of Figure 10. An COOLP inference engine consults an **COOLP Knowledge Base (COOLP KB)** and performs deduction, abduction and constraint solving to answer queries about what can be inferred from it.

An COOLP engine uses the Logtalk OOLP engine together with a set of CLP(X) engines. They are integrated seamlessly by a simple declaration that imports the CLP(X) engines in the top-level object of the Logtalk program. This is made possible because both Logtalk and the CLP(X) engines runtimes all reuse the same underlying Prolog engine. Due to this import statement, the Logtalk engine knows that whenever it encounters, in the premise of a Prolog rule encapsulated in a Logtalk entity, a predicate that is a constraint that CLP(X) is able to solve, it calls the CLP(X) engine to handle it instead of the default Prolog engine.

¹ Indeed, the set of facts of a logic programs literally constitutes a main memory relational database.

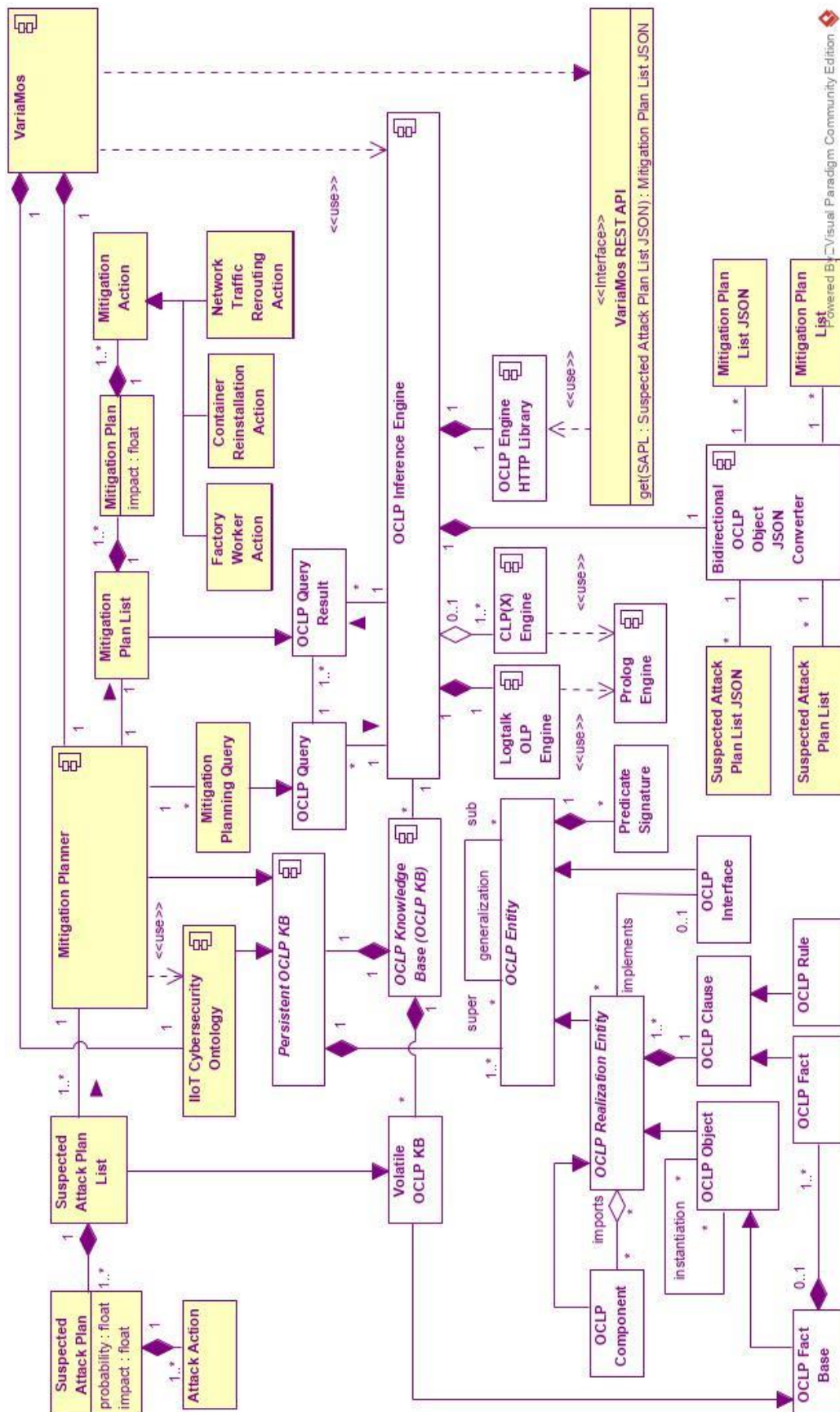


Figure 10: COOLP paradigm and internal architecture of VariaMos

An COOLP KB is typically decomposed into on (a) a *persistent* component that is specific to an application domain such as cybersecurity attack mitigation but remains the same for each inference task instance within this domain and (b) a *volatile* component that is particular to each such task instance. For example for VariaMos as used in the context of C4IIoT, the volatile component of the KB represents the knowledge gathered so far about a particular cyberattack plan that is suspected to be currently executed by a malicious actor targeting the IIoT infrastructure protected by C4IIoT. It also contains the plan under construction to mitigate this attack.

A Logtalk-based COOLP program is a set of COOLP *entities* that contain predicate signatures and can inherit these signatures and other elements that they encapsulate from one another down a generalization hierarchy. The signature of a predicate specifies type, instantiation directionality and multiplicity constraints on its arguments. Such predicate signature constitutes the externally exposed interface of an COOLP entity.

The simplest COOLP entities are COOLP *interfaces* (called *protocols* in Logtalk jargon) that only contain signatures and generalization relationship declarations (using the *extend* Logtalk keyword). These interfaces can be implemented by COOLP *realization entities* that additionally encapsulate COOLP *clauses*. These are CLP clauses extended with the possibility to use object-oriented message passing syntax to invoke the predicates of realization entity.

These entities can be either:

- COOLP *components* (called *categories* in Logtalk jargon) that can be imported in another COOLP realization entity to support the composition pattern much like the mixings of Smalltalk, Python and JavaScript, or
- COOLP *objects* that can be structured in instantiation hierarchies.

Note that in Logtalk, the concept of *class* is not a first-class entity but rather a role played in an instantiation relationship. For example, if object I instantiates object C, then, in this relationship, I plays the role of instance and C the role of class and it inherits the elements of C by class to instance inheritance. But if C itself instantiates object M, then in that second relationship C this time plays the role of instance and M plays the role of class. In such case, M is a metaclass of C. If an COOLP object O does not instantiate any other object but only specializes (*i.e.*, extends) another object P, then Logtalk implements prototype inheritance from P to O rather than class inheritance.

COOLP clauses decompose into:

- COOLP *facts* that are extensional declarations of what is known to be true (axioms) on a given domain at the beginning of an inference task;
- COOLP rules that allow inferring new facts from known facts.

Volatile COOLP KB only contains OLCP facts while persistent COOLP KB may contain both facts and rules.

The specific OLCP engine that VariaMos reuses is the Logtalk engine that runs on top of the SWI-Prolog engine² (Logtalk engines are available for all major Prolog engines). We chose this underlying LP engine because it also provides middleware libraries to encapsulate an COOLP application behind a REST API and make it seamlessly interoperate with services implemented on other runtime platforms in complex heterogeneous applications such a C4IIoT framework instantiation. This middleware includes an HTTP library and a bidirectional converter between,

² <https://www.swi-prolog.org>.

one the one hand, JSON objects encoded as strings, and, on the other end, objects in Logtalk syntax containing the same information.

4.5 Internal architecture of VariaMos

The internal architecture of VariaMos is shown by the components, classes and interfaces with a yellow background in Figure 10. VariaMos realizes a REST API interface that allows external software to get from VariaMos a list of mitigation plans in response to a set of attack plans that are suspected to be under execution by a malicious agent on the IIoT infrastructure protected by C4IIoT. Both the input set and output lists of this REST API are in JSON format. VariaMos first uses the bidirectional converter between COOLP objects and JSON objects built in the COOLP inference engine consisting of the stack Logtalk + CLP(FD) (Triska) + SWI-Prolog to construct the COOLP set of suspected attack plans and insert them in the volatile COOLP fact base. Each plan in this set has a probability property measuring its likelihood and an impact property quantifying the negative cost of the suspected attack if left unmitigated.

To find a counterplan that best mitigates the whole set of suspected attack plans, VariaMos relies on its mitigation planner component. This planner reuses concepts from the other main component of VariaMos its IIoT cybersecurity ontology. Both these components are persistent COOLP KB. Note that semantically, the IIoT cybersecurity ontology of VariaMos differs from ontologies specified using semantic web standards such as the *Ontology Web Language (OWL)* (Van Harmelen, Lifschitz, & Porter, 2008) since it makes the CWA of COOLP inference whereas OWL makes the OWA. CWA allows finding mitigation plans through abduction (Kakas, Michael, & Mourlas, 2000) when insufficient knowledge is available to find such plan through pure deduction. It thus searches for plans heuristically in a space pruned by *a priori* abductive bias. Taking incomplete knowledge as input is an inherent characteristic of cybersecurity attack mitigation since attack detection is always derived from incomplete and uncertain cues.

The VariaMos mitigation planner is a persistent COOLP KB that allows the COOLP inference engine to answer a mitigation planning query. The result of such query is a list of mitigation plans in order of decreasing negative business impact of the mitigated suspected attack. Each plan in the list is composed of three subclasses of actions:

1. Network traffic rerouting actions between application running on network host nodes, which can be automated by leveraging the NBI of the TGS DISCO SDN controller;
2. Software reinstallation in the stack running at given network host nodes suspected to be compromised, which can be automated by leveraging the CLO;
3. Other actions that require manual intervention by a factory worker and that can be displayed on AEGIS cybersecurity officer dashboard.

Once the plan list has been built-by the COOLP inference engine using (a) the volatile KB constructed from the JSON objects passed to VariaMos as input and (b) the VariaMos mitigation planner and IIoT cybersecurity ontology persistent KB, VariaMos then converts the mitigation plan list into a JSON object string as an HTTP message response. For this last step, VariaMos leverages the COOLP object to JSON converter that the COOLP engine provides as built-in.

4.6 VariaMos Prototype

We have implemented the UML diagrams shown in Figure 3, Figure 4, Figure 5 and Figure 7 as a Logtalk program. In Figure 11 we show a screenshot of an example run of the current implementation. In it, VariaMos takes as input a Logtalk object containing the attack plan

instance as its first parameter, accompanied by two option parameters. The first of these option parameters is the number of solutions that we are asking VariaMos to generate. The second parameter is a search heuristic option stating that VariaMos should only search for a single mitigation action instance in the mitigation plan instance to cover each attack action instance in the attack plan instance. The resulting two mitigation plan instances found by VariaMos are pretty printed in sequence below the call to it.

```
?- variaMos4Cybersecurity::chooseMitigations(ap1, 2, mutex).
-----INPUT-----
Received Attack Plan -> ap1
Found attack action -> launchDos of type dos with impact -> VeryHigh and probability -> 90%
Found attack action -> compromiseAsset1 of type malware with impact -> High and probability -> 75%
Found attack action -> compromiseAsset2 of type malware with impact -> High and probability -> 75%
Found attack action -> compromiseAsset3 of type malware with impact -> High and probability -> 75%
Found attack action -> compromiseAsset4 of type malware with impact -> High and probability -> 75%
Found attack action -> nrAt1 of type networkRecon with impact -> Medium and probability -> 75%
-----
Formulating Constraint Satisfaction Problem
Finding 2 solutions for the problem
-----
Mitigation Plan -> mp1
mp1 -> Impact Reduction 78% with 6 Actions
For attack action -> compromiseAsset1 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset2 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset3 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset4 of type malware the chosen mitigation is of type containerReplacement
For attack action -> launchDos of type dos the chosen mitigation is of type hostIsolation
For attack action -> nrAt1 of type networkRecon the chosen mitigation is of type topologyReconfiguration
-----
Mitigation Plan -> mp2
mp2 -> Impact Reduction 74% with 6 Actions
For attack action -> compromiseAsset1 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset2 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset3 of type malware the chosen mitigation is of type containerReplacement
For attack action -> compromiseAsset4 of type malware the chosen mitigation is of type containerReplacement
For attack action -> launchDos of type dos the chosen mitigation is of type hostIsolation
For attack action -> nrAt1 of type networkRecon the chosen mitigation is of type packetFiltering
-----
true.
```

Figure 11: A sample interaction with the VariaMos Prototype

5 DISCO Software Defined Network controller component

DISCO follows the Software Defined Networking (SDN) trend. The SDN paradigm has emerged in the industry to overcome some limitations of current networks, such as vendor dependence, proprietary protocols, lack of scalability and the incompatibility with commodity hardware. The SDN leverages three concepts: separation of the software and physical layers, centralized control of information and network programmability. The physical layer is composed of various networking equipment which provide the underlying data path. By default, those devices are empty of any business logic and require an external controller to handle the traffic. The control layer contains the necessary logic and states to configure the data path. The control layer builds various networking rules and pushes them into the equipment to “program” the network. The OpenFlow protocol is the open standard that describes such rules.

DISCO is a distributed multi-domain SDN control plane which enables the delivery of end-to-end network services. A DISCO controller is in charge of a network domain. A domain could be for example an IaaS domain which provides virtualized computing and network resources. Another domain could be a Radio Access Network (RAN) domain that gives access and the configuration of radio resources. The DISCO controller communicates with neighbor domains to exchange aggregated network-wide information for end-to-end flow management purposes. The **Error! Reference source not found.** presents the architecture. It is composed of two parts: an intra-domain part, which gathers the main functionalities of the controller, and an inter-domain part, which manages the communication with other DISCO controllers (reservation, topology state modifications, disruptions ...). In addition to this east-west interface, a controller has at least one southbound SDN interface used to push policies and rules to the network elements and retrieve their status. Finally, a northbound interface enables to push management policies to the controller (e.g., flows, service and user priorities), to manage SLAs and report network service status. A controller is composed of several modules managed by the Core component. It enables to start, stop, update the modules and provides them with a communication bus.

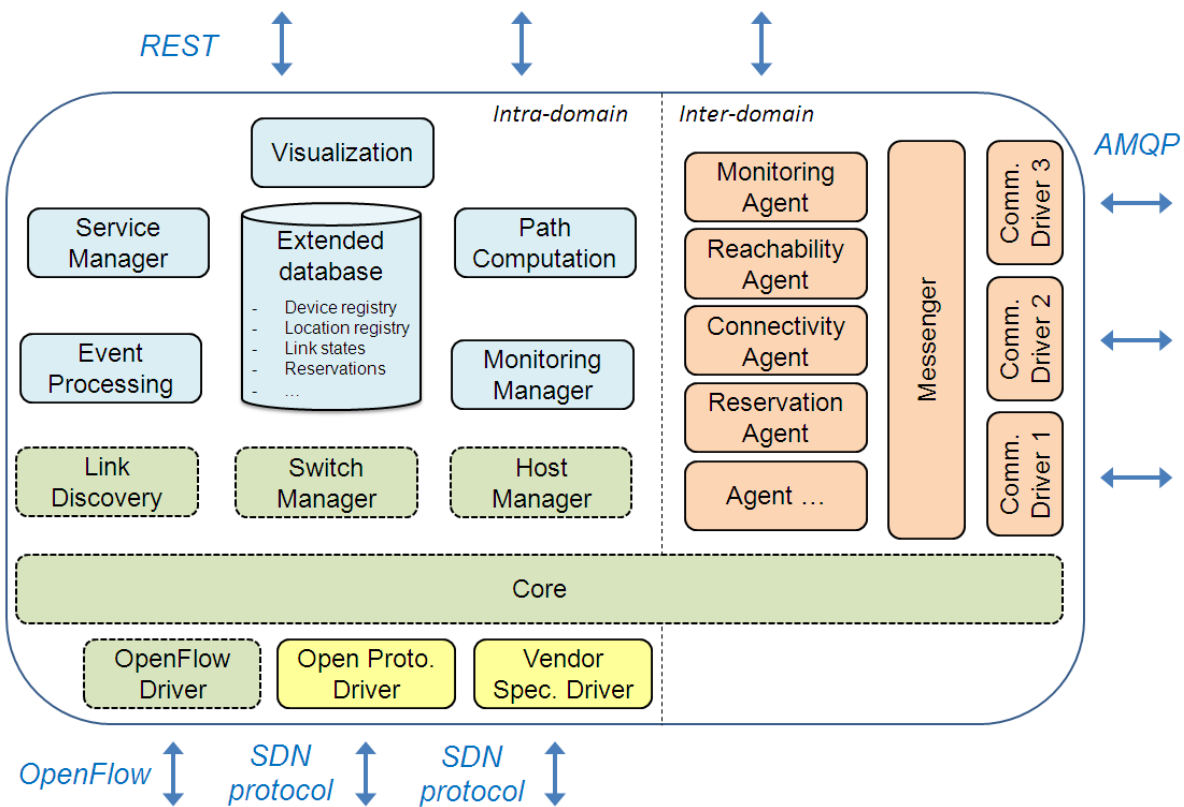


Figure 12 - DISCO Controller Architecture

5.1 Interfaces

5.1.1 Northbound

The controller northbound interface is a REST API which will translate the intent of the network-related output from VariaMos. The intent can take various forms: from low level raw OpenFlow rules to a high-level abstraction that hides some complexity. This interface will interact with a newly created module in the controller which will be developed specifically for C4IIoT (see **Error! Reference source not found.**). The goal of this module will be to output actionable events for the SDN-enabled switch/routers in the factories using a southbound interface. It will translate the actions decided by the VariaMos system to an intermediate format more related to the underlying networking devices.

5.1.2 Southbound

The SDN controller will use OpenFlow as its southbound protocol. OpenFlow is one of the leading standard communication interfaces defined between the control and data transmission layers of the SDN architecture, and specifies primitives used to program the network devices. First developed by Stanford University it is now being standardized by the Open Networking Foundation (ONF), a consortium led by the major telecommunication and network companies, and is already implemented by the industry.

The networking devices, supporting the OpenFlow protocol, maintain decision rules into an OpenFlow table. In this ecosystem, these rules are called “flows” and they are stored in “flow

tables”. When the networking devices boot, the flow tables are initially empty and the devices are unable of performing any logic to handle the network traffic. In such a state, when a network packet hits a network device, it forwards the packet to the SDN controller which builds the corresponding OpenFlow flow to drop, switch, nat, etc the packet. The network device stores this flow into the flow table and then processes the suspended packet. The following packets that match any of the flow in the flow table are immediately handled.

Other southbound interfaces will not be used in the context of C4IIoT (see **Error! Reference source not found.**) as the OpenFlow protocol is powerful enough and widely supported.

5.1.3 East/Westbound

Because only one SDN controller will be in the architecture, the multi-domain capability of DISCO will be left deactivated (see **Error! Reference source not found.**). Nevertheless, it would be possible to share information between controllers located in different places (e.g. transmit a possible malware detection) to pre-emptively act on unaffected factories.

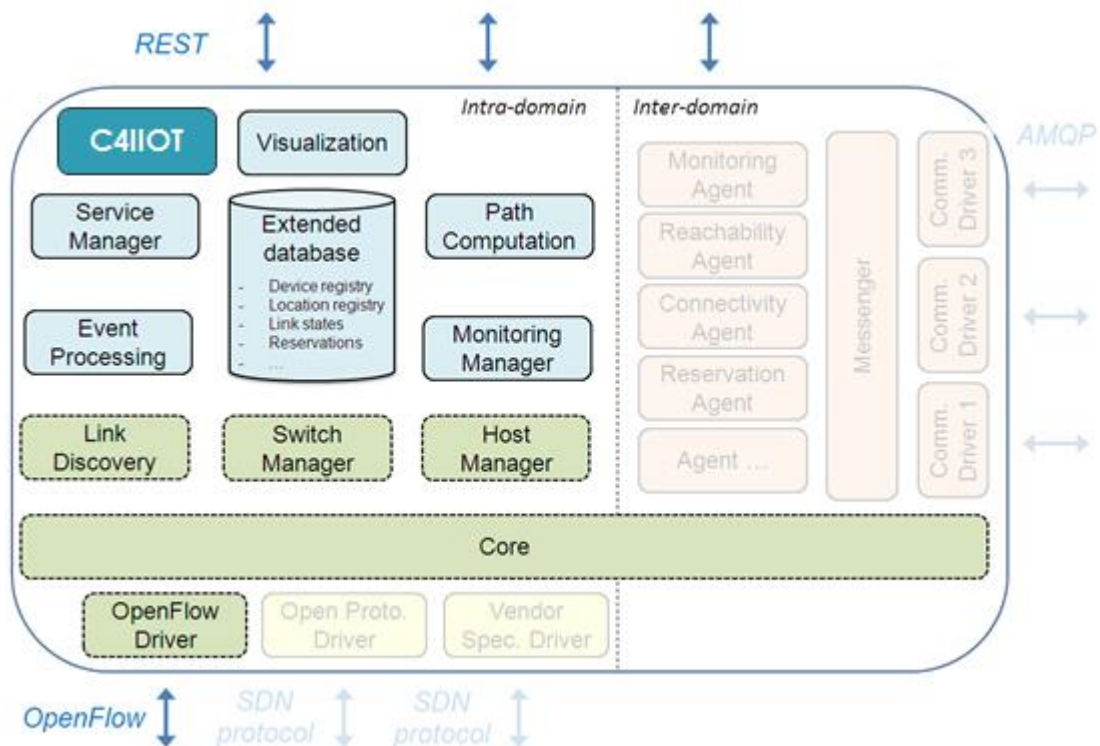


Figure 13 - DISCO Controller architecture for C4IIoT

Moreover, most modern SDN solutions are designed to live in the same network region (datacenter), which means that network failures are sparse and their impacts are limited to a small subset of devices. But in a context of a network attack that severs the link between two factories, a simple decentralized SDN controller deployed widely may enter into a split-brain scenario. Classical decentralized solutions rely on consensus algorithm like Paxos or Raft to elect a leader. Two approaches exist to deal with this problem. The first one is to let each SDN controller subset to work as usual and have reconciliation mechanisms to merge the various states when the link is restored. Sometimes those mechanisms are not enough and a manual tricky intervention is required. The second solution is to let the side with the quorum to work

as usual and to lock the other one into a read only mode. With this solution, the limited side may be unable to enact actions to mitigate the attack.

5.2 Actions

The SDN controller will be able to affect the network in multiple ways. We can see below a selection of these actions:

- IP blocking: if an element (e.g. network equipment, IoT device) is acting suspiciously the SDN controller can enact a total block of the device from the network to isolate it or a partial block by banning an external IP with which the device was communicating.
- Port blocking: similar to the previous actions the SDN controller can isolate a specific suspicious port.
- Rerouting: if a network path is compromised, the SDN controller can redirect flows to an alternate route (provided such route exists).
- Prioritization: in case of increased traffic, the SDN controller can apply prioritization rules to shape the traffic.
- Capturing: the SDN can redirect a suspicious flow into a network honeypot for extended probing or inspection.
- Reporting and reconfiguration: as the SDN controller survey and maintains the flow tables in the networking devices, compromised equipment with corrupted tables can be detected and reported.

The full list of actions will evolve with discussion with other elements of the mitigation engine and the additional actions will be integrated in the C4IIoT module of the DISCO controller.

5.3 Role of DISCO in the mitigation engine

In the context of C4IIoT, the SDN controller will act as the network actuator. It will interface with the output of VariaMos and transmit orders to the SDN-enabled switch/routers. The SDN controller acts as a centralized interaction point to configure and manage the underlying low level networking devices. It may also provide as more abstract and higher level api depending on the project needs. This indirection, core of the SDN trend, makes the networking devices focus on the data path and provides a main access point for the other project components to interact with the network control plane.

5.4 Input to DISCO in C4IIoT

The main DISCO input will be actions emitted from the VariaMos mitigation plan. Upon analysis, VariaMos will restrict the network rules depending on the detected threats. At first, these actions will be simple and may take the form of low level OpenFlow rules. The Disco controller will parse, validate and cache these rules then push them into the network equipment. Depending on the needs, a custom higher level api may be implemented to simplify the VariaMos actions.

Depending on how the DISCO controller is setup and the understanding the VariaMos system has on the network, two scenarios can arise. First the DISCO SDN controller can provide a default open network with automatic path learning. In this mode, the hosts connected to the switches can communicate which others freely. The VariaMos can retrieve the topology from the SDN controller itself or discover it from logs and then emits actions. Second the DISCO SDN controller may disable all flow by default and wait for the VariaMos mitigation engine to allow communication between hosts by emitting a precise static flow entry.

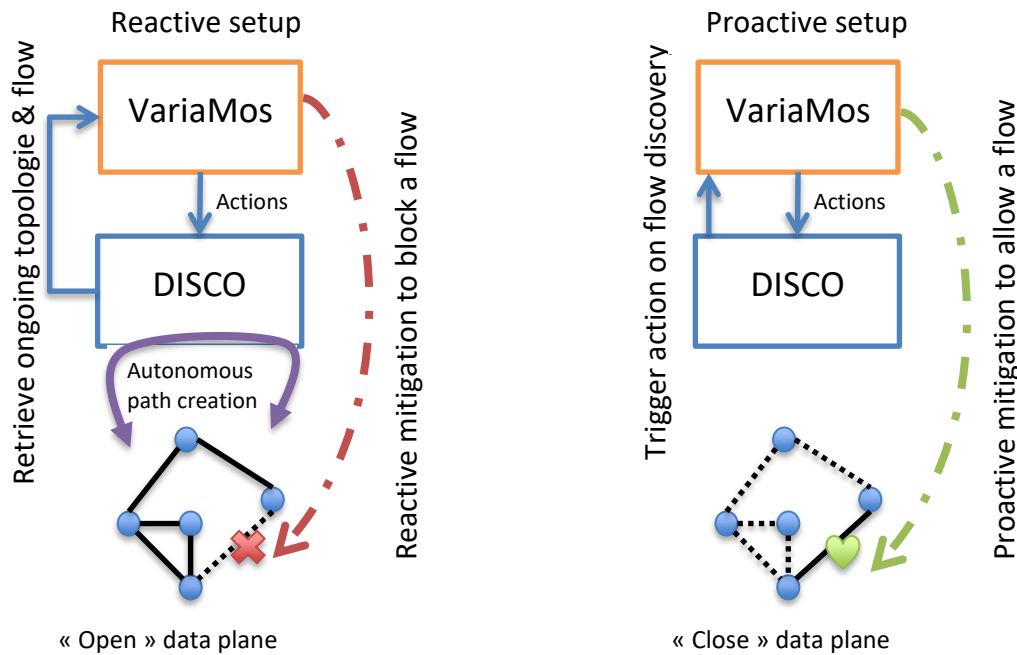


Figure 14: Comparison between the reactive and proactive setups

5.5 Output of DISCO in C4IIoT

The DISCO controller produces one main output which is the network policies. Those rules are pushed into the networking devices managed by the C4IIoT platform. They will emerge from two sources: first from the network itself to configure the ongoing traffic in a loosely fashion. Second from the VariaMos component that will emit actions to DISCO to tighten or block the flows.

But, as a SDN controller, DISCO also acts as a central API point to discover and list the network equipment supporting the OpenFlow protocol, and also serves as an attachment point to them. To work, DISCO requires to keep track of the different network elements through a switch and a host manager.

5.6 Technologies used by DISCO

DISCO is a custom proof of concept controller based on Floodlight, a SDN controller made in Java that can be extended with custom logic. Those extension ranges from specific REST APIs for a specific intent based API, to low level flow rules generation and configuration for advanced network management.

Indirectly, DISCO generates and manipulates the OpenFlow rules to manage the networking equipment.

DISCO also uses the Advanced Message Queuing Protocol (AMQP) as a base for its internal message oriented middleware. It offers various message oriented mechanisms such as queuing with priority, point-to-point communication, publish and subscribe messaging, etc. The AMQP system provides multiple delivery guarantees such as at-most-once, at-least-once and exactly-

once. AMQP is lightweight and highly controllable; it is for example used in OpenStack for communication between its internal components.

6 HPE Container Orchestrator (HPECO) component

The Mitigation Engine, like all the other C4IIoT components of the Cloud Layer, will be deployed as Docker Container inside a Kubernetes cluster managed by C4IIoT Orchestrator.

6.1 Role of Cloud Layer Orchestrator in threat mitigation

The full description of the Cloud Layer environment, its internal services as well as the security mechanisms to protect C4IIoT components is described inside deliverable D3.1 in chapter 3 “Heterogeneous Hybrid Cloud Environment” - refer to that document for the full details.

Specifically related to Mitigation Engine third set of actions mentioned in section 3.3, CLO will expose native Kubernetes control commands to support the set of container substitution actions in the chosen mitigation plan at hosts running applications suspected by the attack alert to have been compromised.

6.2 Input from the mitigation engine to Cloud Layer Orchestrator

The mitigation engine will provide a factory app deployment model, expressed as YAML formatted descriptions of the Kubernetes resources configuration related to the affected app, e.g. deployments, pods, replica set, services, etc³.

6.3 Output from Cloud Layer Orchestrator to the mitigation engine

The updated factory app deployment model will be returned to the Mitigation Engine with YAML formatted descriptions of the updated resources.

6.4 Technologies used by Cloud Layer Orchestrator

The Technology used inside the Cloud Layer for the private docker registry is based on the Open Source project Harbor⁴ project that is offering a trusted cloud native repository for Kubernetes that integrates all the features needed by C4IIoT: i.e. private docker image registry with integrated control on image signing process and image vulnerabilities scanning.

For the vulnerability scan of the stored images, Clair docker image vulnerability scanning tool⁵ is integrated by Harbor, and automatically executed for each newly stored image.

The factory app deployment model for Cloud Layer Orchestrator is described in terms of Kubernetes description of resources in YAML format. Specifically, the K8s Deployment object⁶ contains the attribute “template/spec/containers/image” that stores the reference to the image that will be executed.

The Replica Set⁷ Controller of Kubernetes controls the execution of a deployment inside a Pod resource: modifying the attribute “spec/replicas”, the number of deployments instances can be tuned, and setting it to value “0” means turn the app off.

³ <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

⁴ <https://goharbor.io/>

⁵ <https://coreos.com/clair/docs/latest/>

⁶ <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

⁷ <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>

In addition, the Mitigation Engine could use the feature “Rolling Update”⁸ feature of Kubernetes – that is typically used to perform app version updates without stopping services – to perform a graceful switch from a “suspect” version to a “safe” one.

⁸ <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>

7 BINSEC factory app binary code analyser component

7.1 BINSEC overview

BINSEC is a platform for static analysis of binary codes using formal methods.

The main characteristic of BINSEC is that it operates at **binary level**, i.e. on program executables (e.g. `.exe` files) after they have been compiled from source, which is typical for programs written in languages like C, C++, Rust, Go, Fortran, Ada or Pascal.

Binary executables contain **machine code**, i.e. a combination of low-level instructions, designed for execution by a specific processor. Thus, BINSEC must analyse for different hardware architectures, with different instruction sets.

Typical examples of its usage include:

- **vulnerability discovery**, i.e. finding new security bugs in existing programs;
- **vulnerability analysis**, e.g. finding the specific conditions under which a vulnerability can be exploited by an attacker;
- **reverse engineering**, i.e. understanding the behaviour of a program without access to its source code;
- **malware analysis**, i.e. reverse engineering on *obfuscated* binary code, produced to make reverse engineering more difficult;
- **program verification at the binary level**, i.e. proving that a binary program meets some property, for instance that a binary program does not suffer from buffer overflows.
- **program verification in programs containing assembly/binary fragments**, i.e. verifying a program at the source level, using BINSEC to help it understand the assembly parts or external binary-only libraries used by the program.

Program analysis using formal methods is a very challenging research domain, which began before the creation of the first computers in the 40s and is still not solved. Machine code analysis adds another difficulty on top of that, in the fact that machine code is made for programs to execute, and not for humans to understand; in particular, it lacks all the common programming language abstractions (functions, control structures, data structures) that makes a program easier to understand. A binary executable is literally a collection of ones and zeros.

7.2 BINSEC architecture and components

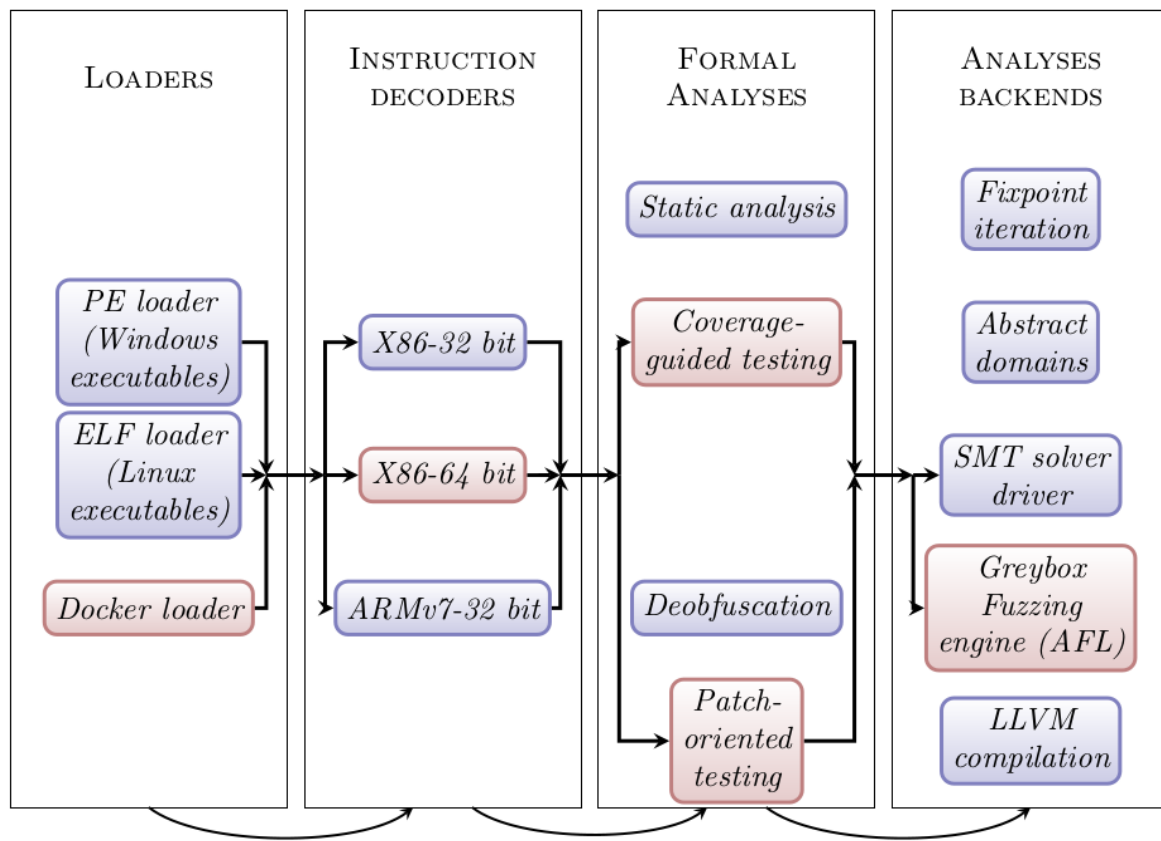


Figure 15: BINSEC architecture and updates

The BINSEC architecture consists of 4 different layers. The analysis of a binary executable by the BINSEC tool can be viewed as choosing a path that goes through one element in each layer.

7.2.1 Loaders

This layer is responsible for parsing the format of a machine-code executable file. Executable binaries contain information about how the operating system will place the different parts of the program (the code, the stack, the global variables) in memory. Loaders are used to parse this file, and provide an environment that simulates the loading of the file by the operating system.

At the beginning of the project, there were two available loaders in BINSEC:

- The ELF loader is responsible for parsing binary executables in ELF (Executable and Linking Format). This format is widely used in Unix-like operating systems, such as Linux, Solaris, Irix, FreeBSD, NetBSD, or OpenBSD.
- The PE loader is responsible for parsing binary executable in the PE (Portable Executable) format. This format is used for binary executables for the Windows operating system.

7.2.2 Instruction decoders

This layer is responsible for providing a semantics for the different architectures. This is done by the decoder, whose role is to decode the machine instruction into a machine-independent language called DBA. The DBA language has a central role in BINSEC, as all the analyses are based on its semantics. It is a very low-level language, as witnessed by its syntax:

$\langle \text{stmt} \rangle$	$:=$	$\text{store } \langle e \rangle \langle e \rangle \mid \langle \text{reg} \rangle \leftarrow \langle e \rangle \mid \text{goto } \langle e \rangle$ $\mid \langle \text{stmt} \rangle ; \langle \text{stmt} \rangle \mid \text{if } \langle e \rangle \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$
$\langle e \rangle$	$:=$	$\langle \text{cst} \rangle \mid \langle \text{reg} \rangle \mid \text{load } \langle e \rangle \mid \langle \text{unop} \rangle \langle e \rangle \mid \langle e \rangle \langle \text{binop} \rangle \langle e \rangle$
$\langle \text{unop} \rangle$	$:=$	$\neg \mid - \mid \text{uext}_n \mid \text{sext}_n \mid \text{extract}_{i..j}$
$\langle \text{binop} \rangle$	$:=$	$\langle \text{arith} \rangle \mid \langle \text{bitwise} \rangle \mid \langle \text{cmp} \rangle \mid \text{concat}$
$\langle \text{arith} \rangle$	$:=$	$+ \mid - \mid \times \mid \text{udiv} \mid \text{urem} \mid \text{sdiv} \mid \text{srem}$
$\langle \text{bitwise} \rangle$	$:=$	$\text{and} \mid \text{or} \mid \text{xor} \mid \text{shl} \mid \text{shr} \mid \text{sar}$
$\langle \text{cmp} \rangle$	$:=$	$= \mid \neq \mid >_u \mid <_u \mid >_s \mid <_s$

Figure 16: The DBA formal intermediate language

As we can see, DBA instructions consist in moving bitvectors between registers and memory, computing values in registers, computing flags registers, conditional and indirect jumps, and is thus a very low-level language.

At the beginning of the project, there were two available mature instruction decoders in BINSEC:

- **x86-32**, an instruction decoder for instructions of the Intel x86 instruction set, but without support for the more recent 64bit instruction set.
- **armv7-32**, an instruction decoder for the ARMv7 instruction set (corresponding to the Cortex M-,A-,R- families of processors).

7.2.3 Formal analyses

The BINSEC platform implements a variety of formal analysis techniques. Formal techniques can be categorized in several categories:

- **Sound methods** can prove that a property is valid, but cannot be used to find a counter-example if the property is not. Examples of sound method include static analysis based on abstract interpretation, typing, or weakest-precondition computation
- **Complete methods** can find a counter-example when a property is invalid, but cannot prove that a property is valid. Example of complete methods include fuzzing and symbolic execution.
- **Heuristic methods** are neither sound nor complete, but can be used to find likely properties or likely bugs. These can be completed using sound methods (to confirm that the invariant is correct) or complete methods (to confirm the presence of a bug).

Because most of the problems that BINSEC attempts to solve are undecidable, it is impossible to implement methods that are neither sound nor complete.

BINSEC implements the following family of analyses:

- **Static analysis based on abstract interpretation** [15,20,21] is a *sound* method to automatically verify properties of binary programs, by constructing a representation describing a superset of all the behaviours of the program. The main advantage of this method is that it allows to prove that a binary executable is invulnerable to some kinds of attacks. The main drawback of this technique is that it is very sensitive to imprecision, which in practice does not allow using it on large programs without some amount of manual work.
- **Coverage guided testing** [16,18] is a *complete* method whose goal is to reach the maximum number of targets in the code; the targets can be either instructions, functions, branches, paths... The method works by finding new inputs of the programs that allows reaching the target. There are two main challenges in this area: handling large programs efficiently, and finding inputs allowing reaching hard-to-reach code, e.g. relying on conditions of the input that are unlikely to be found randomly.
- **Deobfuscation** [17,19] are *sound* techniques for simplifying a machine code program, typically for malware analysis.

7.3 Adaptations for the C4IIoT Framework

7.3.1 Needs for C4IIoT

For integration in the C4IIoT Framework, we established that the key constraints for integrating a binary code analysis for continuous verification in the mitigation engine are, in this order:

1. **Automation:** Binary code analysis should be performed with minimal intervention from the platform user. In particular, we should avoid tedious configuration or the need for a user to supply annotations about the source code. A too long configuration phase could deter the user from performing the code analysis, and would thus jeopardize the security of the system.
2. **Precision:** The issues reported by the analyser should have a very low rate of false positive, i.e. we should report only real problems upon which the system needs to act. Indeed, too many false alarms could lead the end user into ignoring the issues. Additionally, false alarms can cause the rest of the mitigation engine to react when there is no need to.
3. **Security guarantees:** Given the above constraints, we should strive to provide the strongest possible security guarantees. However, we can not trade more security for less automation.

In the light of these constraints, we concluded that, in general, sound methods like static analysis are not practical for the project, because they become imprecise without user annotations, and report many false alarms. They can still be used to guarantee the security of critical components that are unlikely to change, but we must not focus on them.

Thus, our goal is to focus on complete methods, that can find bugs instead of proving their absence. Even if these methods provide lower security guarantees than sound methods, their ability to be more automated and the fact that they mostly report true errors are in line with the focus of the project.

7.3.2 New components being developed

In practice, the following changes have been done or are planned in the BINSEC platform to support C4IIoT.

1. **A new x86-64 bit** instruction decoder has begun to be implemented, as it is an instruction set likely to be used in the C4IIoT platform. This poses many challenges, because this instruction set is probably the most complex one used industrially, and because BINSEC has always been used in 32-bit platforms.
1. **A new docker loader:** The C4iiot platform has converged on the use of docker as a standard for images for programs running on the cloud layer. In order to analyse them, we have to integrate a docker loader in the platform. This is a large change to BINSEC, because docker is not a single-executable file format, but rather a format describing a whole virtual machine.
2. **Integration of a greybox fuzzing engine (AFL) with BINSEC.** Greybox fuzzing is an effective way to find bugs in binary programs. For greybox fuzzing to be effective, it must be fast (to create a large number of tests) and good at covering code. AFL is a widely used standard greybox fuzzing tool, that we have integrated into BINSEC.
3. **Update to our coverage-guided testing**, which previously relied only on symbolic execution, to also handle greybox fuzzing. The combination of both symbolic execution and fuzzing is useful to address the challenges of coverage-guided testing, as greybox fuzzing can handle large programs efficiently, and symbolic execution is good at finding hard-to-reach code.
4. **A new patch-oriented-testing** [22] method, that allows to focus the testing on specific targets of the code. Its role is to help incremental testing by a targeted patch differential analysis (i.e., trying to find security vulnerabilities by focusing on the recent additions to the code). This relies especially on the integration of the greybox fuzzing engine AFL.

7.4 Preliminary studies about the development of the new coverage-guided testing and patch-oriented testing engines of BINSEC

The main technical difficulties in the work carried out by CEA since the beginning of the project is in the improvement of its automatic testing engine that will be used in the project. We briefly report on the results that we obtained.

7.4.1 Deep coverage-guided testing for automating low-level security analysis

The problem here is that standard fuzzing have difficulties to reach certain parts of a code under test, typically complex conditions or nested conditions. Prior works, such as Driller or Eclipser try to overcome this issue by shallow combination symbolic execution (SE) and fuzzing. However, this approach brings only very slight improvement (less than 10% coverage), as SE is too expensive compared with fuzzing. Hence, such combination either fires SE very rarely (bringing mostly nothing) or is too slow. We have performed extensive benchmarks to get a deep understanding of these results (presented in the next section).

We have been working on a smart integration of semantic approaches into fuzzing, bringing semantic reasoning directly at the core-level of fuzzers. Our technique, coined Confuzz (Constrained Fuzzer), performs much better than AFL on some hard-to-solve examples from the CGC (cyber grand challenge) cybersecurity benchmark. These initial results are promising.

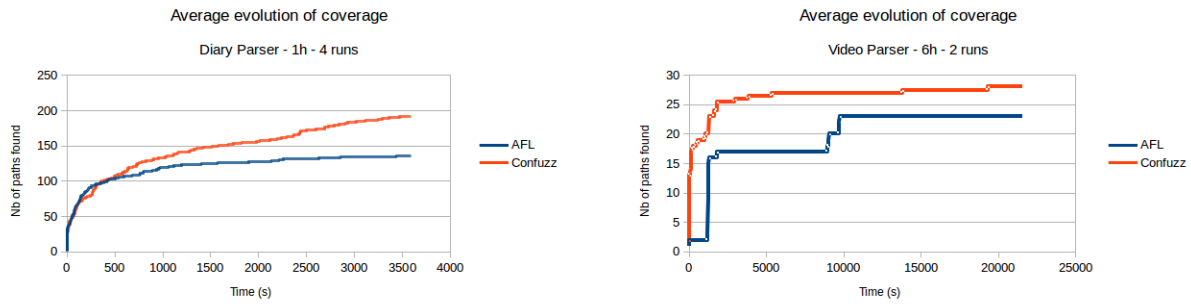


Figure 17: Comparison between Confuzz and AFL

7.4.2 Directed fuzzing for incremental and targeted analysis and patch-oriented testing

Directed fuzzers seek to reproduce a known bug from a partial trace description, typically coming from debug information, stack trace, or, patches. It appears that existing tools do not address well the case of use-after-free bugs (UAF), a rising class of bugs used for example for information leakage or protection bypass. It turns out that UAF are extremely hard-to-find for fuzzers: (1) they require to find input chaining three sequences of events in the code (allocation, deallocation, usage), which is very unlikely; and (2) these bugs are often silent and do not crash, hence requiring to pass all generated input to a Valgrind-like runtime monitor, with an additional cost of postprocessing. Finally, existing tools have a very expensive instrumentation cost.

We are currently working on a binary-level directed fuzzing method geared at UAF bugs, with excellent results against existing directed fuzzing methods. Especially, our method shows very light instrumentation overhead, performs significantly better than competitors and reduce the total number of inputs to be postprocessed by one order of magnitude.

Fuzzing tool	directed	number of successful runs
AFL-Qemu	x	65/110
AFLGo	yes	70/110
Hawkeye	yes	49/110
BINSEC/UAFuzz	yes	99/110

Figure 18: Success rates of directed fuzzers

7.4.3 Evaluation of possible solutions

To understand why 1. fuzzing alone is problematic to cover some C4IIoT use cases and 2. why shallow combination of fuzzing and symbolic execution was insufficient to solve the problem, we performed extensive set of benchmarks using existing programs and tools, and we give their main teaching here.

EVALUATION ON THE STANDARD LAVA-M BENCHMARK

First, we have compared a standard fuzzing tools (AFL, AFL++ and AFL-Qemu), with a tool combining symbolic execution with fuzzing, using the standard LAVA-M fuzzing benchmark. We ran each test for 5 hours. The results are summarized in the following table.

Tool	AFL		AFL++		AFL-Qemu		Eclipser	
	paths	crashes	paths	crashes	paths	crashes	paths	crashes
LAVA-base64	388	0	345	0	123	0	55	742
LAVA-md5sum	247	0	328	0	250	0	41	238
LAVA-uniq	103	1	119	0	106	0	83	1431
LAVA-who	178	0	44	0	70	0	1	0

Figure 19: Importance of symbolic execution on the LAVA benchmark

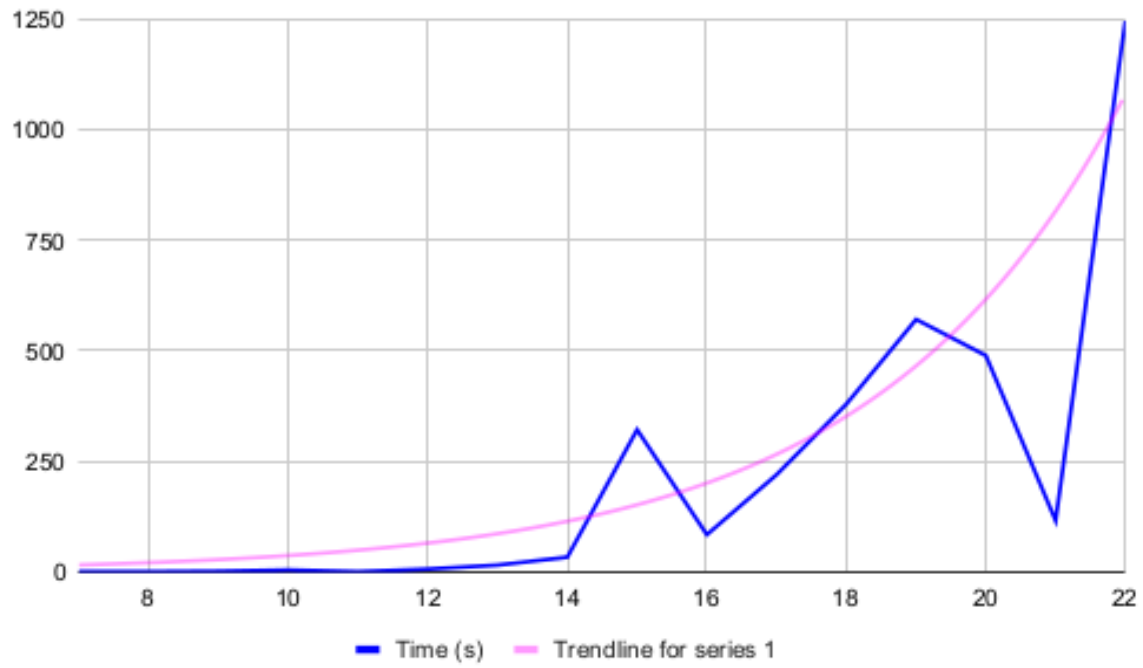
We can see that on this benchmark, Eclipser is generally much more effective than AFL-like fuzzers at finding crashes; except for the who case. After investigation, we found out that Eclipser had trouble finding a correct pass when no seed was provided (when provided with a normal utmp file, it could find 70 paths and 14 crashes in 5 minutes).

Actually, these results are very bad for AFL and its variants on the LAVA-M benchmark which is considered as the standard benchmark for the evaluation of fuzzers. Even with 12 hours, AFL++ (the best AFL) on md5sum finds 399 paths but no crash. Still, care must be taken. Indeed, bugs in LAVA-M are synthetic and they all share the same pattern: if (expr = magicbyte) crash. This kind of bug is extremely hard to find for AFL (basically, 1 chance out of 2^{64} in case expr is an input integer), while its representativeness is questionable. In many cases, a bug can be triggered by much more than a single value, e.g. a buffer overflow. Also, we must keep in mind that LAVA-M has been created to distinguish between state of the art fuzzers at a time when AFL was already well established. Hence, LAVA-M is in some way designed to be hard for AFL.

STUDY: ABILITY OF AFL AND ECLIPSER TO FIND COMPLEX/HARD TO REACH BUGS

To understand better the bad results of AFL on the Lava benchmark, we have created a synthetic benchmark with a single program, which triggers a bug only when N specific bit values are found in the input. Intuitively, LAVA-M corresponds to the worst case setting, where all bits of a given input must be found, and simple crash (crashing any execution reaching a given point) corresponds to the simplest case, where only 0 bit must be found.

N	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Time to first crash (s)	0	1	2	5	1	7	16	34	323	85	221	381	573	491	117	1250



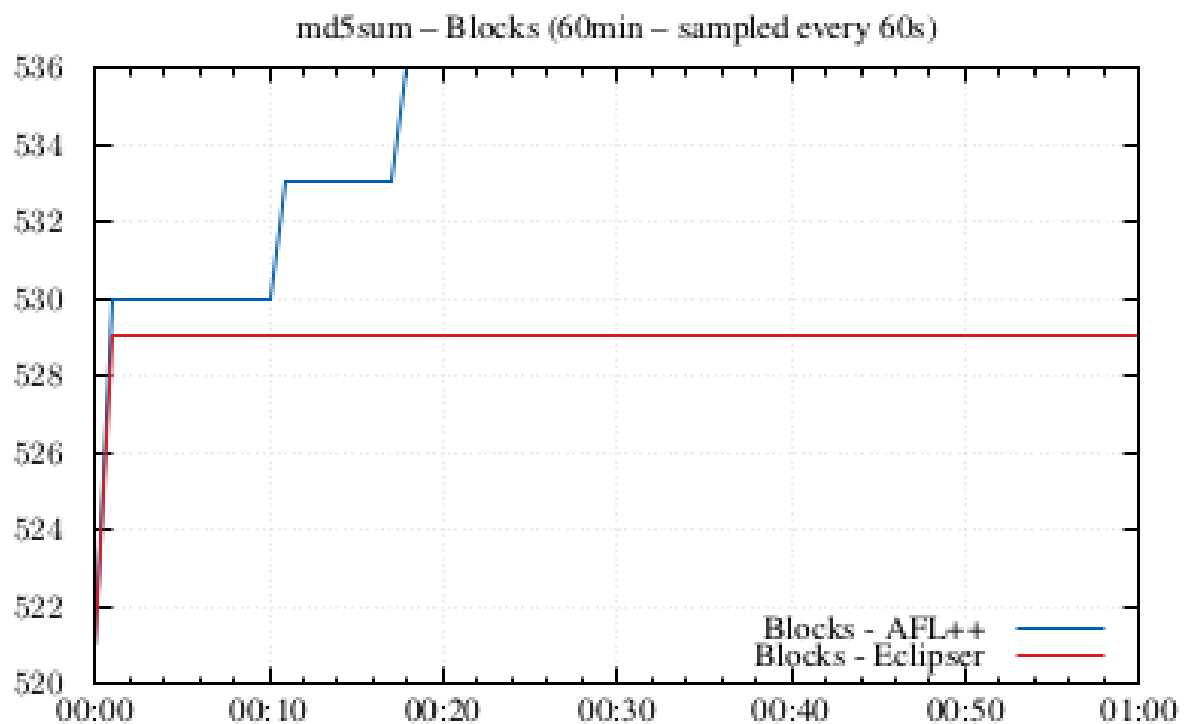
N	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	31	32
Time to first crash (s)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 20: Importance of symbolic execution on a synthetic benchmark

We conclude that this approach of combining fuzzing with symbolic execution is well-appropriate for handling of hard to reach path conditions, which explains its good results on the LAVA benchmark. We also conclude that using fuzzing alone is not a good idea if the goal is to reach a thorough coverage of the code, because fuzzing will miss hard-to-reach conditions. This motivates our choice to combine symbolic execution and fuzzing for C4IIoT, as such hard-to-reach conditions are very common in IoT programs (for instance, when there are “magic bytes” used to identify a IoT protocol in a network stream).

STUDY: SHORTCOMING OF CURRENT FUZZING/SYMBOLIC EXECUTION INTEGRATION

In this last sequence of experiments, we wanted to evaluate the ability of greybox fuzzers augmented with symbolic execution to reach the different parts of the code on real benchmarks, and their scalability to large use cases. For this, we have evaluated the coverage of the fuzzer over time on programs of small size (md5sum), medium size (bc), and large size (ghostscript). The results show that, in real programs, Eclipser does not help in finding more paths. Thus, the interaction between fuzzing and symbolic execution has to be done carefully if we want results that work correctly on real programs such as those developed in C4IIoT.



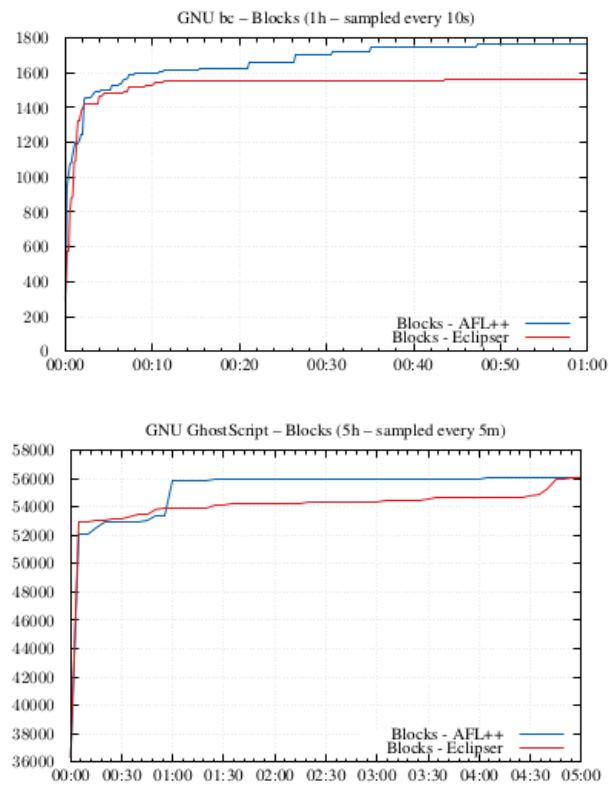


Figure 21: Shortcomings of current symbolic execution and fuzzing integration

8 Conclusion

This deliverable presented the first output of Task 3.3 ("Mitigation engine"). We have described the overall structure of the mitigation engine, which builds upon three main components: a binary code analyser, a software-defined networking controller, and a central brain performing the analysis of possible reconfiguration based on the available inputs.

A central part of the work lies in defining the interactions, both between the components of the mitigation engine, but also with the reminder of the C4IIot framework. In particular, this includes interaction with HPE's Cloud Layer Orchestrator, through which every connection with the rest of the platform will go through.

Furthermore, we have provided detailed specification of the necessary updates to each of the main components of the project in order to implement the envisioned architecture for the C4IIoT mitigation engine, paving the way for the complete implementation that will be developed during the next two years of the project.

Despite the care taken to plan and analyse the interactions between the mitigation engine and the rest of the project, it is possible that the envisioned architecture may need some adaptations to accommodate the need of other components of the project. There are several opportunities to present these future adaptations; first in the deliverable D3.3 (Level-2 and Level-3 security mechanisms of C4IIoT), planned at month 18 of the project; second in deliverable D3.4 (Cyber assurance and protection in an industrial cloud infrastructure), planned at month 30.

9 References

- [1] M. Whitman and H. Mattord, Principles of Information Security (6th ed.), Boston, MA: Course Technology Press, 2017.
- [2] B. Schneier, "Attack Trees," *Dr. Dobbs Journal*, vol. 24, no. 12, pp. 21-29, 1999.
- [3] J. Knight, Fundamentals of Dependable Computing for Software Engineers, Boca Raton, Florida, USA: CRC Press, 2012.
- [4] P. Cichonski, T. Millar, T. Grance and K. Scarfone, "Special Publication 800-61 (rev2) Computer Security Incident Handling Guide," National Institute Of Standards and Technology (NIST), 2012.
- [5] H. Hindy, D. Brosset, E. Bayne, A. Seeam, C. Tachtatzis, R. Atkinson and X. Bellekens, "A Taxonomy and Survey of Intrusion Detection System Design Techniques, Network Threats and Datasets," *arXiv preprint arXiv:1806.03517*, 2018.
- [6] P. Stuckey, K. Marriott and G. Tack, "MiniZinc Handbook Release 2.4.3," [Online]. Available: <https://www.minizinc.org/doc-2.4.3/en/index.html>.
- [7] T. Frühwirth and S. Abdennadher, Essentials of Constraint Programming, Springer, Ed., 2003.
- [8] P. Moura, "Logtalk: Design of an Object-Oriented Logic Programming Language," PhD. Thesis, University of Beira Interior, Covilhã, Portugal, 2003.
- [9] F. Van Harmelen, V. Lifschitz and B. Porter, Eds., Handbook of Knowledge Representation, Elsevier Science, 2008.
- [10] A. Cropper, S. Dumancic and S. Muggleton, "Turning 30: New Ideas in Inductive Logic Programming," in *29th International Joint Conference on Artificial Intelligence (IJCAI'20)*, Yokohama, Japan, 2020.
- [11] S. Russell and P. Norvig, Artificial Intelligence, a Modern Approach, 4th ed., Pearson, 2020.
- [12] P. Moura, "The Logtalk Handbook," 2020. [Online]. Available: <https://logtalk.org/manuals/index.html#>.
- [13] M. Triska, "The Finite Domain Constraint Solver of SWI-Prolog," *Lecture Notes in Computer Science (LNCS)*, no. 7294.
- [14] A. Kakas, A. Michael and C. Mourlas, "ACLP: Abductive Constraint Logic Programming," no. 44, 2000.
- [15] Adel Djoudi, and Sébastien Bardin. "Binsec: Binary code analysis with low-level regions." International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2015.
- [16] Robin David, et al. "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis." 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

- [17] Sébastien Bardin, Robin David, and Jean-Yves Marion. "Backward-bounded DSE: targeting infeasibility questions on obfuscated codes." 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017.
- [18] Robin David, et al. "Specification of concretization and symbolization policies in symbolic execution." Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016.
- [19] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. "Symbolic deobfuscation: From virtualized code back to the original." International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 2018.
- [20] Adel Djoudi, Sebastien Bardin, and Éric Goubault. "Recovering high-level conditions from binary programs." International Symposium on Formal Methods. Springer, Cham, 2016.
- [721] Nicole, O., Lemerre, M., Bardin, S., & Rival, X. (2020). Automatically Proving Microkernels Free from Privilege Escalation from their Executable. arXiv preprint arXiv:2003.08915.
- [22] Nguyen, M. D., Bardin, S., Bonichon, R., Groz, R., & Lemerre, M. (2020). Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. arXiv preprint arXiv:2002.10751.