



DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

HypervisorLang

Attack Simulations of the OpenStack Nova
Compute Node

FREDDY AASBERG

HypervisorLang: Attack Simulations of the OpenStack Nova Compute Node

FREDDY AASBERG

Master in Computer Science

Date: February 20, 2021

Supervisor: Viktor Engström

Examiner: Mathias Ekstedt

School of Electrical Engineering and Computer Science

Host company: Saab AB

Swedish title: HypervisorLang: Attacksimulering av OpenStack

Novas Beräkningsnod

Abstract

Cloud services are growing in popularity and the global public cloud services are forecasted to increase by 17% in 2020[1]. The popularity of cloud services is due to the improved resource allocation for providers and simplicity of use for the customer. Due to the increasing popularity of cloud services and its increased use by companies, the security assessment of the services is strategically becoming more critical. Assessing the security of a cloud system can be problematic because of its complexity since the systems are composed of many different technologies. One way of simplifying the security assessment is attack simulations, covering cyberattacks of the investigated system.

This thesis will make use of Meta Attack language (MAL) to create the Domain-Specific Language (DSL) HypervisorLang that models the virtualisation layer in an OpenStack Nova setup. The result of this thesis is a proposed DSL HypervisorLang which uses attack simulation to model hostile usage of the service and defences to evade those. The hostile usage covers attacks such as a denial of services, buffer overflows and out-of-bound-read and are sourced via known vulnerabilities. To implement the main components of the Nova module into HypervisorLang, literature studies were performed and included components in Nova together with threat modelling.

Evaluating the correctness of HypervisorLang was performed by implementing test cases to display the different attack steps included in the model. However, the results also show that some limitations of the evaluations have been found and are proposed for further research.

Sammanfattning

Molntjänster växer i popularitet och de publika molntjänsterna förväntas öka med 17% år 2020[1]. Populariteten beror bland annat på en förbättrad resursanvändning hos leverantörer och enkelheten för kunden att införskaffa resurser. På grund av molntjänsternas ökande popularitet och deras ökade användning hos företag blir säkerhetsanalyser av tjänsterna mer kritisk. Att bedöma en molntjänsts säkerhet kan vara problematiskt på grund av dess komplexitet. Detta eftersom systemen oftast består av många olika tekniker. Ett sätt att förenkla säkerhetsanalysen är attacksimuleringar som täcker cyberattacker mot den undersökta tjänsten.

Detta examensarbete kommer att använda Meta Attack Language (MAL) för att skapa ett domänspecifikt språk som modellerar virtualiseringslagret i en OpenStack Nova-installation. Resultatet av examensarbetet är HypervisorLang som använder attacksimuleringar för att modellera attacker mot tjänsten samt säkerhetslösningar för att undvika dem. Några av attackerna som täcks av modellen är 'denial-of-service' (DOS), Out-of-bound-read, buffer overflow och är hämtade via kända sårbarheter. Utvecklingen av språket genomfördes med hjälp av litteraturstudier av komponenterna i Nova tillsammans med studier kring hotmodellering gällande de komponenter som ingår i modellen.

Utvärderingen av HypervisorLang utfördes genom att implementera testfall för att bekräfta att de olika attackstegen som ingår i modellen fungerar som tänkt. Resultaten visar också att vissa begränsningar av utvärderingarna har hittats och föreslås för framtida forskning.

Acknowledgements

I would like to express my deepest gratitude to Prof. Mathias Ekstedt and Viktor Engström for their valuable and constructive suggestions and continuous help. I also wish to thank Joakim Ekblad and Filip Lundqvist for their support and advice during this thesis.

Contents

1	Introduction	1
1.1	Problem definition	3
1.1.1	Research Question	3
1.1.2	Delimitations and scope	3
1.1.3	Ethics and sustainability	4
1.1.4	Thesis outline	4
2	Related work	6
3	Background	8
3.1	Cloud computing	8
3.1.1	Deployments models	10
3.2	Probabilistic Threat modelling	11
3.2.1	Mathematical formalism of the Meta attack language .	12
3.3	The Meta attack language	14
3.3.1	coreLang	16
4	Methodology	18
4.1	Domain Survey	19
4.1.1	Attack Lists	19
4.1.2	Creation of Domain-specific language	19
4.1.3	Evaluation	19
5	Domain Survey	20
5.1	OpenStack	20
5.2	Hypervisor	21
5.2.1	Instances	23
5.3	SELinux & sVirt	23
5.3.1	Secure Virtualisation	24

6	Results	25
6.1	Feature Matrix and Attack Lists	25
6.1.1	Feature Matrix	25
6.1.2	Attack and defence lists	26
6.2	HypervisorLang	28
6.2.1	coreLang	29
6.2.2	HypervisorLang	30
7	Evaluation	36
7.1	Evaluation of Testcases	36
7.1.1	Example test case 1 - Access to Instance	36
7.1.2	Example test case 2 - Breakout from Instance	38
7.1.3	Example test case 3 - Access to host	39
8	Discussion	40
8.1	Completeness of HypervisorLang	40
8.2	Completeness of Attack Lists and Defenses	40
8.3	Completeness of testing	41
8.4	Using MAL as a base for HypervisorLang	41
8.5	Regarding the works of HypervisorLang	41
9	Conclusion and future work	43
9.1	Conclusion	43
9.2	Future work	43
	Bibliography	45
A	Test-Cases	50
B	Core	57
C	HyperVisorLang	66
D	Testmodel	74
D.1	TestCase1 in Result	74
D.2	test2	75
D.3	test3	78

Chapter 1

Introduction

Cloud services are getting more and more popular every year, the global public cloud services are forecast to grow 17% in 2020[1], whereas the IaaS cloud services are the fastest-growing segment of the market, with a forecasted growth of 24% year after year[1].

Cloud within the IT sector is a broad concept of providing many different types of services via the web. The concept is not new and was first thought of in the early 1960s when John McCarthy opined that “computation may someday be organised as a public utility” and the first scholarly usage of the term "cloud computing" was in a lecture by Ramnath Chellappa, in 1997[2].

The main change which cloud computing brings is the possibility for companies to outsource their need for infrastructures such as servers and network. For example, if a small startup would like to launch a web service, they only need to pay for a cloud service which covers their needs usually described as "pay-as-you-go", instead of investing in a complete server. The opposite of cloud computing is "on-premise" where the company host their hardware for their applications.

Two of the key players taking Cloud computing to the broad masses were Amazon and Salesforce. Amazon saw the need for maximising utilisation of their computing resources, which was using as little as 10 per cent of the capacity at any given time[2]. Amazon Web services (AWS) was launched as a utility-based computing service in 2006[2]. Salesforce was launched to customers in 1999 providing a SaaS "Software as a service", to their customers.

In 2010 OpenStack was founded by Nasa and Rackspace[3][4]. OpenStack provides a free opensource cloud operating system with a large variety of ser-

vices, amongst these services are a standard IaaS functionality, container and function services, orchestration provided by additional components, fault and service management[5]. Some of the commercial providers of OpenStack are RedHat[6] and Rackspace.

Assessing the security of cloud systems could be difficult since the infrastructure is composed of numerous technologies which are working together to provide a wide variety of services. This implies that the person assessing the security of cloud systems require information regarding the technologies and their security-relevant features, and when this is completed an assessment of the system as a whole is needed. The system can be described as a chain, and a chain is only as strong as the weakest link.

One of the key technologies that enable cloud IaaS is the Hypervisor, which let servers run multiple virtual-machines (VM) or Instances which can be seen as virtual servers. The Hypervisor monitor the virtual machines and presents an operating platform for each VM. This solution enables the cloud provider to optimise the resource utilization on the server and enabling multiple customers using one server, or one customer using a server for multiple projects. The VM or Instances can be used to host a web server or other types of software that needs computing resources.

There are several approaches for assessing the security of a system, formal verification and model checking, which is a way of exploring and verifying all states of a model. Formal verification is used in the aerospace industry for verifying software.

Threat modelling and attack simulation are two ways of assessing the security of a system. There are many different definitions of threat modelling[7], and one definition given by Banquero et al., is "threat modelling is the technique that assists software engineers to identify and document potential security threats associated with a software product, providing development teams a systematic way of discovering strengths and weaknesses in their software applications"[8]. Attack simulations are a non-intrusive way of simulating attacks on a model of a system. Penetration testing or ethical hacking is a way of assessing security by hiring professionals to find exploits in the system. Meta Attack Language (MAL) provides the means of codifying domain-specific knowledge. With MAL it is possible to create domain-specific languages that describe applications and systems, which then in the domain-specific language is paired with attacks and defences. The outcome of the codification is attack

graphs which represent different attack paths in the system described in the domain-specific language (DSL). One domain-specific language used in this thesis is coreLang[9], which models an abstract domain with assets such as applications, firewalls and networks[10].

1.1 Problem definition

The problem which is addressed in this thesis is the complexity of ensuring the security of the virtualisation layer in the cloud platform. The complexity is due to the many different components the virtualisation layer consists of. These components also relate to each other in different ways and bring their vulnerabilities to the system. Due to this complexity, security is problematic to ensure.

This thesis project will investigate the possibility of representing the compute node provided by OpenStack in a domain-specific language HypervisorLang to simulate cyber attacks as an aid to simplify the process of assessing security in the virtualisation layer. To create a model of the compute node a domain-specific language will be created, and to do this the Meta Attack Language (MAL)[12] will be used. The codified threat model will contain vulnerabilities tied to the identified assets and will be used for simulating virtual attacks.

1.1.1 Research Question

This thesis aims to examine and answer the following question:

Req 1: How can a domain-specific language be designed to accurately simulate cyber-attacks on a QEMU host setup?

1.1.2 Delimitations and scope

OpenStack provides an open-source cloud operating system which consists of multiple components providing a wide range of services. This thesis project will be limited to the Nova compute module and more specifically to a QEMU/KVM hypervisor setup with Ephemeral storage. In this setup, the host hypervisor is at focus together with the host, instances and the Novaservice command-line interface (CLI). Apart from this, related components from Core lang will be added, such as Data and System(the host). The Libvirt API will not be covered, since the Novaservice CLI communicates with libvirt which then, in

turn, sends the information to QEMU, but in this domain-specific language, the NovaService CLI talks directly to QEMU. This is a simplification, which also removes the ability to describe attack vectors tied to libvirt. No storage modules will be added due to time constraints, therefore only Ephemeral storage will be described, this further implies that persistent storage is missing and if an instance were to be removed the data is lost.

1.1.3 Ethics and sustainability

Today's digital society sees increased threats in the form of cyberattacks, in recent years cyber attacks have increased and pose a greater problem for companies and individuals who operate digitally. As the use of cloud services is steadily increasing, tools are also needed to easily assess how secure service is, this to ensure that the service is trusted by both the user and the provider. The creation of a domain-specific language (DSL) that analyzes the security of a cloud service can advantageously result in the cloud service becoming more secure. It can also result in the user of DSL gaining an increased understanding of Cybersecurity and how one with simple means can improve this. On the downside, in the same way, a user can access the information and improve the security of a system, the methodology can be used to identify weaknesses in a system to possibly use them to exploit the system. As for sustainability, developing secure software can help expand the lifetime of a software, whereas security flaws can cause the users mitigate to a different software. As for the writer, I would argue that the ethics of providing means of assessing the security of a system for a good cause does out-weight the downside of someone using the tool for identifying the flaws and misuse them.

1.1.4 Thesis outline

The thesis is organized in the following outline:

Chapter 2 Contains related work which is relevant for the thesis and its covered subjects.

Chapter 3 presents the background, and covers cloud computing, deployment models probabilistic threat modelling, the mathematical formalism of MAL and the meta language MAL.

Chapter 4 Contains the methodology which the thesis has used and explains

the different phases of the method.

Chapter 5 Consists of the Domain Survey and display the results of the survey.

Chapter 6 presents the results of the thesis, the feature matrix and the discovered attacks and defenses.

Chapter 7 presents the discussion.

Chapter 8 contains evaluation of the test cases and presents three examples of cases.

Chapter 9 presents conclusions and future work.

Chapter 2

Related work

The Meta Attack Language or MAL is part of the field of model-driven security and attack graphs. Two approaches in this field are UMLSec and SecureUML which emerged to further aid developers creating security-critical systems, they are both based on the Unified Modelling Language (UML) that is a standardised modelling language used in object-oriented programming projects to provide system specifications. UMLSec was developed as an extension to UML and provides a tool to add expressions regarding security in the UML-diagrams of a system specification[14]. The focus of SecureUML is modelling access control policies based on Role-based access Control extended with constraints and integrates into a model-driven software development process[15].

MAL supports probabilistic threat modelling, threat modelling which is a process of finding potential threats to a system or lack of safeguards in certain parts of a system. Relevant to this thesis is the survey conducted by Rajendra Patil and Chirag Modi[11], which is an in-depth survey covering various vulnerabilities, security threats and attacks related to the two hypervisors XEN and KVM+QEMU[11]. The vulnerabilities are also classified to the components of the virtualisation infrastructure they are tied to. Also the work[16] by Diego Perez-Botero, Jakub Szefer, and Ruby Lee has been relevant to this thesis. In this paper vulnerabilities tied to the Hypervisor are characterized to the affected component[16].

MAL uses the concept of attack graphs to visualise the different attack scenarios and many different approaches which are based on graphs has been proposed[17], and several tools based on attack graphs have been developed. The

purpose of these tools has been to collect information regarding the structure of a system or infrastructure, and automatically generate attack-graphs based on the data. Attack trees is a different concept which is often mentioned with attack-graphs and were popularized in 1999 when the paper written by Bruce Schneier regarding modelling security threats in computer systems by using Attack Trees[18] was released. The paper was later on extended by Kordy et al.[19] who further added defences to the Attack Trees. One example of such a tool that uses Attack-graphs is MulVAL, which is a framework for determining the security impacts of software exploits on a network. MulVAL uses information from a vulnerability database together with configuration data from each machine on the scanned network, then derives logical attack graphs from the combined data[20]. Another tool is NetSPA - a Network Security Planning Architecture, which outputs worst-case attack-graphs from the use of network configuration information[21].

A sub-domain in the field of attack graphs are Probabilistic attack graphs where the steps involved in the graph are assigned with a probability, thus creating a Bayesian network. In [22] the authors use the Bayesian attack graphs to assess the security risks of a network system at various levels, the information can then be used to form a security mitigation and management plan. Most relevant for this thesis project is the works of Johnson et al. in creating the Meta Attack Language[12], which support the creation of domain-specific languages for probabilistic threat modelling and attack simulations. MAL has previously been used to create domain-specific languages which also is relevant for the creation of HypervisorLang, Corelang[9] which models a large abstract domain of IT and AWSlang[23] which is a domain-specific language to the AWS cloud platform.

Chapter 3

Background

This chapter will cover relevant background information connected to HypervisorLang. 3.1 Covers cloud computing and Deployment models for cloud applications. 3.2 Covers Probabilistic Threat modelling and mathematical formalism of MAL, in 3.3 MAL is introduced.

3.1 Cloud computing

Cloud computing can be described as a resource on demand, or pay as you go for the consumer. To the provider cloud computing can be used to maximise resource utilisation, since multiple tenants can share the resources.

The National Institute of Standards and Technology (NIST) definition of cloud computing: *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The cloud model is composed into five essential characteristics, three service models and four deployment models*[p.2, 24].

The five essential characteristics are:

- *On-demand self-service.* The consumer can independently provision computing capabilities, such as server time and network storage. This can be done without requiring human interaction with each service provider[24].
- *Broad network access.* The resources hosted by the provider are accessible from a broad range of devices, such as mobile phones, tablets, laptops, and workstations[24].

- *Resource pooling.* The providers computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to a consumers demand[24]. Examples of resources include storage, network bandwidth, memory and processing power. The resource pooling enables optimisation of resource usage since many consumers can make use of the same resources.
- *Rapid elasticity.* Capabilities can be elastically provisioned and released, in some cases automatically[24].
To the consumer, resources can appear unlimited or infinite[24], unlike the era before cloud computing when storage and computing power limits were visible to the consumer.
- *Measured service.* Cloud systems automatically control and optimise resources by leveraging a metering capability at some level of abstraction appropriate of the type of service provided (storage, processing power, network bandwidth, and active user accounts)[24]. The measurement tools can provide both the consumer and provider with an account on the utilisation of the services.

The three different types of service models described by NIST:

- Software as a Service (SaaS) is a software delivery model which can be described as a software licensing model or "software on demand", instead of the traditional way where the consumer installs the software on their computer.
With SaaS, the consumer makes use of the provider's software which is executed on a cloud infrastructure[24]. The access is usually provided via the web-browser or a client program. The provider takes care of configurations, updates, cloud infrastructure and since the software is hosted this way it is also possible to provide the service via a subscription model.
- Platform as a Service (PaaS) is a step further to the left, as shown in Figure 3.1. PaaS lets the consumer deploy their software on a cloud service. The consumer takes part in managing the code and configurations of the application-hosting environment, the programming languages, libraries, services, tools, and all the underlying infrastructure such as network, operating system, storage and servers are all provided by the cloud provider[24].

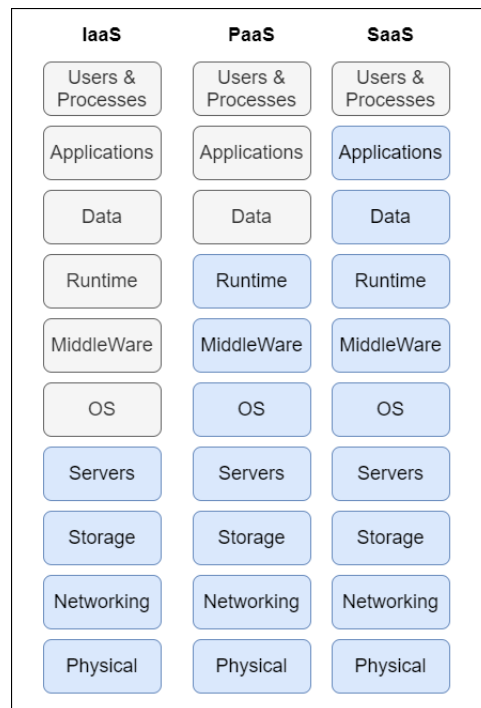


Figure 3.1: Depiction of service models, the blue colour describes the modules managed by the cloud-vendor. The gray areas are the parts managed by the consumer.

- With Infrastructure as a Service (IaaS), the cloud provider offers 'raw' access to the infrastructure which consist of processing, storage, network and other fundamental computing resources [24]. The consumer is responsible for the application software, operating system and has control over storage[24].

3.1.1 Deployments models

The way of deploying a cloud infrastructure can be done in various ways, and could be described as the 'configuration' of the infrastructure and are based on the needs of a consumer.

Private cloud

Private cloud is when the infrastructure is operated by one exclusive organisation, which also may be divided into several different business units using

the same infrastructure. The infrastructure may be owned, managed and operated by the organisation, or by a third party[24]. It may also reside on or off premises[24]. Traditionally private clouds ran on-premise, but currently organizations are building private clouds on rented, vendor owned data centers which are resided off-premise.

Community cloud

The community cloud infrastructure is provisioned and shared by the members of an exclusive community[24]. The infrastructure might be owned, managed, and operated by one or more of the members in the community[24].

Public cloud

The public cloud infrastructure is managed by the cloud vendor and resides on the premises of the same vendor[25]. The pooling of resources is done automatically by a self-service interface[25]. The infrastructure is provisioned for the use of the general public[24].

Hybrid cloud

The hybrid cloud is a mix between the former deployment models (public, private or community cloud)[24], for example, some parts may reside on the private infrastructure, while other parts are deployed in the public infrastructure[26].

3.2 Probabilistic Threat modelling

Threat modelling is a process of assessing the security of a system or an infrastructure in a structured way[27] and help identify threats, vulnerabilities and countermeasures in the evaluated system. Preferably security engineering should be integrated as early as possible in the software development process[27], it is also possible to implement threat modelling on an existing system, but this is generally more time consuming and costly when incorporating security fixes at a later stage[27]. When the security requirements are formed, the threats are analysed concerning likelihood and criticality, based on the outcome the threat could either be mitigated or the risk is accepted[27]. The benefits of using threat modelling when developing security requirements are that the process helps define realistic and meaningful requirements[27] by looking

at the system as a whole and taking threats and vulnerabilities into consideration. This is especially important since the security definition of the system would be flawed if the security requirements were to be picked at random.

Myagmar et al[27] describe the process of threat modelling in three steps.

Characterizing the System, describing dataflows in a software or a network model. Which could be a description of computers in a network that are connected. This step emphasising extracting the main characteristics of a system. The second step is to **Identify assets and access points**, an asset is an abstract description of a resource in the system that needs to be protected from an adversary. One example of an asset would be customer-related data or a system used to provide services, in the threat model the assets are targeted by an adversary. Access points are the ways into the systems that an attacker could exploit, such as ports, web-services or an SSH connection. When these two steps have been completed the third step is to **identify the threats** to the system. Depending on what type of system that is modelled, the threat would differ, examples of threats are denial of service, information disclosure or elevation of privileges.

Attack-graphs is a valuable tool in the threat modelling process and visualise the path an attacker could perform through the system. Thus the Attack-graphs offer both detection of vulnerabilities in a system and mitigations[28]. The attack tree represents the identified vulnerabilities and each node is one possible attack designed according to the attacker's perspective[28]. The edges connecting the nodes represent relations between attacks, where one attack is connected to another succeeding attack. The constructed attack tree can then be analysed by the assessor, according to parameters specified by the said assessor. The chosen parameters could differ depending on the system, but one parameter, being used in this thesis, is *time to compromise* where each attack step would have a local time to compromise, and the tree as a whole would have a global time to compromise[12].

3.2.1 Mathematical formalism of the Meta attack language

Let X express an object or domain entity. For example, an object could be a Laptop or an Application. Objects are divided into a set of classes

$X = X_1, \dots, X_n$, e.g.

$Laptop \in Machine$ and $Application \in Software$

Each class is linked with a set of attack steps $A(X_i)$, $X.A$ is used to denote the attack step from an object A in class X . For example, an application could be infected with malicious code which performs some sort of code-execution. Examples of attack steps of the previously mentioned class could be *Application.codeExecution* or *Laptop.fullaccess*.

The representation of relationships between objects are denoted by *links* and *associations* in MAL[12]. A *link* relationship is denoted by λ , and consist of a binary tuple of objects, each taken from a class, such that $\lambda = (X_i, X_j)$. For instance, an application needs to be installed on a laptop.

Links are partitioned into associations such that $\Lambda = \{\Lambda_1, \dots, \Lambda_n\}$, that relate classes to each other in the following manner[12]:

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

Classes also play *roles* in associations, $\Psi(X_i, \Lambda)$, For example, an Installed association between Application and Machine would define a role, say, Host-System for Laptop which would define that the application resides on the Machine when installed.

As mentioned earlier, the attack steps are connected through directed edges, $e \in E$,

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

One example of such a connection could be that accessing an application on the laptop could lead to request access to data in said application. And implies that the first attack-step leads to the second step.

$$e = (Laptop.access, Application.data.requestAccess).$$

To further enhance the threat modelling a proposed addition is to use a probabilistic relational model, which is useful in the creation and analysis of attack graphs[29]. The probabilistic relational model allows for the creation of a Bayesian network, where the attack steps in the attack graph can be associated with a probability. The addition of probability to each attack step enhances the model and instead of displaying attack steps as "if this step is breached, then these steps are available", the model instead expresses how likely a cer-

tain path is to occur. The Probabilistic relational model also contains classes, class-relationships and attributes, which makes it possible to associate a probabilistic dependency model to the attributes of classes in the architectural meta-model[29].

3.3 The Meta attack language

The following text presents the Meta attack language (MAL), which is a meta-language based on the formalism presented earlier. MAL is also the foundation for HypervisorLang, which this thesis project is based upon. Only the core entities of the MAL specification will be presented, and further information can be found at the original publication[12] and the mal-lang GitHub account[30].

```
asset System {
  | connect
    -> attemptGainFullAccess
  | authenticate
    -> attemptGainFullAccess
  & attemptGainFullAccess
    -> fullAccess
  | fullAccess
    -> attemptAccessToData
  | attemptAccessToData
    -> sysData.attemptAccess
}
```

Figure 3.2: Depiction of the System asset and included attack steps

```
asset Data {
  | attemptAccess
    -> access
}
```

Figure 3.3: Depiction of the data asset and included attack step

The name of the asset in fig 3.2 is System and includes attack steps, there are four OR-step denoted by | named connect, authenticate, fullAccess and attemptAccessToData. The OR-step implies that only one parent attack step is

```

associations{
  System [system] 1..* <-- DataHosting --
> * [sysData] Data
}

```

Figure 3.4: Depiction of associations between assets

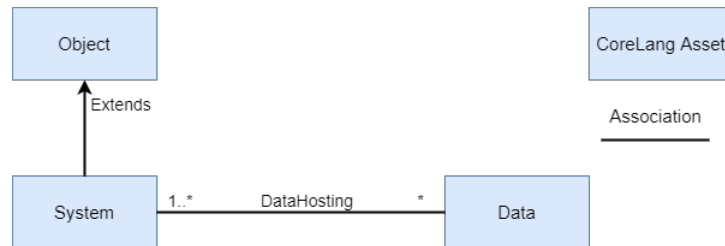


Figure 3.5: Depiction of two assets in coreLang

needed to reach this node, for example the `attemptAccessToData`-step can be reached from `fullAccess`. The third attack step `attemptGainFullAccess` of the type AND is denoted by an `&`, this implies that both of the parent attack-steps are needed to reach this step (connect and authenticate). Defences in the language can be defined by `#`. The arrow `->` represents the next step if connect is compromised, and the attack would proceed to `attemptGainFullAccess` when both OR steps are compromised. `sysData` in the `attemptAccessToData` step is an association-role and are denoted in MAL by specifying roles in the following manner seen in figure 3.4. This connects the attack step `attemptAccessToData` in figure 3.2 with `attemptAccess` in figure 3.3. The roles "sysData" and "system" represent the relation step between System and Data, where one Machine can host the Data. The representation from System to Data is through the role "sysData". The cardinality of the association is declared after the role, "1..*" which translates to "one or many (*)" computers can host data. One attack step may be described such as `sysData.attemptAccess` and describes an attempt to access any of the data which resides on the System.

MAL also supports class inheritances in the same manner as other object-oriented languages. Classes which are intended for specialisation and never meant to be instantiated may be defined as an abstract class (asset). The associations are depicted in figure 3.5 to visualise how the assets are connected.

As seen above in figure 3.6, the abstract class `Object` defines the attack step "attemptUseVulnerability" which is then extended to the class `System`, it is also possible to override the `attemptUseVulnerability`-step. In the above

```

abstract asset Object{
    | attemptUseVulnerability
        -> ...
}

asset System extends Object{
    | connect
        -> attemptGainFullAccess
    | authenticate
        -> attemptGainFullAccess
    & attemptGainFullAccess
        -> fullAccess
    | attemptUseVulnerability
        -> fullAccess
}

```

Figure 3.6: Depiction of extending an asset in MAL

attack, the "fullAccess" is compromised instantaneously via the "attemptUseVulnerability", there may be situations where a compromise of a certain attack step requires a certain amount of time. Below in figure 3.7 is such an example where a dictionary attack against an Instance would result in access to the Instance. The dictionary attack in this example is specified to take 18 hours in the following way: As mentioned before defences in MAL are denoted by the #-sign, and BOOLEAN values indicate if the defence is active or not. In the above example, 2FA is added and assigned to the attemptGainFullAccess-step. This leads to that the attack step "attemptGainFullAccess" is unreachable if the 2FA-protection is set to true.

It is also possible to use probability distributions to better describe uncertainties tied to an attack. For example, the choice of password, or how early in the employed dictionary the aforementioned password is listed. In the example depicted in figure 3.8, the deterministic time of 18 is instead a probabilistic distribution, indicating that on average, the dictionary attack would be expected to take 18 hours but with an added uncertainty.

3.3.1 coreLang

coreLang is a domain-specific language (DSL) created with MAL and models an abstract IT-infrastructure. The modelled components are core structures of

```

class Instance{
  | connect
    -> dictionaryAttack
  | dictionaryAttack [18.0]
    -> attemptGainFullAccess
  | attemptGainFullAccess
    -> fullAccess
  # 2FA
    -> attemptGainFullAccess
}

```

Figure 3.7: Depiction of specifying the time consumption of a specific attack

```

class Instance{
  | connect
    -> attemptGainFullAccess

  | dictionaryAttack [GammaDistribution(1.5, 15)]
    -> attemptGainFullAccess

  & attemptGainFullAccess
    -> fullAccess
}

```

Figure 3.8: Depiction of a GammaDistribution connected to a attack step

software systems and IT infrastructure[10]. coreLang is meant to be a foundation for future DSL, where the DSL's can make use of the basic structures captured in coreLang[10]. The assets covered by coreLang are *System* which are Compute instances, *Vulnerability* a set of vulnerabilities and exploits, *User* that covers exploits connected to the users, *IAM* Identity and access management, *Data resources* which models data, *Network* that models the OSI model in a compact way[10].

Chapter 4

Methodology

The *design science research methodology* (DSRM) by Preffers et al [31] was used during the thesis project, since it offers an systematic process for the development of an artefact involving principles and procedures. The DSRM methodology consists of six activities, the first part is *problem identification and definition* where the problem definition is connected to the creation HypervisorLang which is part of the problem solution. The second and third activity is to *Define the objectives for a solution* and *design and development*. The second activity connects to the focus of HypervisorLang, since the OpenStack platform is broad and contains numerous different services the scope had to be narrowed down to only focus on the Virtualisation layer in the computing node, and the third activity of development is when developing the actual language in MAL. This is followed by *Demonstration, Evaluation and Communication* where communication is the part of this thesis to communicate the works of HypervisorLang, the steps are depicted in figure 4.1.

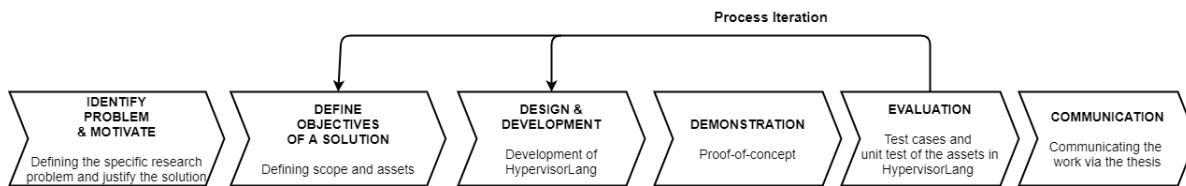


Figure 4.1: Depiction of the DSRM process model

4.1 Domain Survey

During this thesis, an artefact was created, and to fully understand the assets included in the artefact, an extensive survey was needed. The first part of the study was to limit the thesis project and identifying main assets which were to be included in the artefact. The second part was to find studies covering threat models regarding hypervisors. Also, earlier creations of domain-specific languages in MAL was taken into consideration when creating HypervisorLang, such as AWSLang[23] and corLang[9].

4.1.1 Attack Lists

The following phase was to create Attack Lists tied to the assets identified in the earlier phase. The threat models by Rajendra Patil and Chirag Modi[11] was a great help, also the CVE[32] database was consulted. Further, installation of OpenStack was used together with the OpenStack documentation to understand how the Cloud platform could be managed.

4.1.2 Creation of Domain-specific language

The creation of the artefact was done during the domain survey and creating the Attack Lists. The first part of creating the domain-specific language was to identify main assets for the language and combine the findings into the model. Also, many features have been collected from coreLang[9] and AWSlang[23], however, the main parts from these two are based on coreLang, since it is updated and also has some parts adapted from AWSlang.

4.1.3 Evaluation

The DRSM process model by Preffers et al [31] proposes testing of the artefact as evaluation. Testing has also been applied to the HypervisorLang since it is supported by MAL, and in the case of evaluating HypervisorLang both unit testing and integration testing will be performed. The Unit test used for evaluating that each asset works as intended by asserting the outcome from specific attack steps or a whole attack chain. The integration testing is used to perform traversing via the associated assets and testing that the combined functions do work as intended.

Chapter 5

Domain Survey

This chapter covers the outcome of the domain survey. In section 5.1 the OpenStack Nova is covered. A background regarding Hypervisors are covered in section 5.2 and in 5.3 SELinux and sVirt are covered.

5.1 OpenStack

OpenStack is an openSource cloud operating system which can be used to deploy public or private clouds. It is an IaaS cloud platform that also has further possibilities to add additional components that provides orchestration, service management and other services.

- **Nova** Nova is the computational part in OpenStack, it consists of five different parts working together to provide a way to provision the computational resources to users (virtual servers). These parts are depicted in 5.1 and are:
 - **Nova-compute** Nova-compute is primarily a worker daemon, which has the function to create or terminate virtual machines. This is done via the hypervisor API, for KVM libvirt is used.
 - **Nova-Scheduler** The Nova-Scheduler determines how a compute-request will be dispatched[33]. If a compute-request arrives, the scheduler decides which host that will launch the VM (Virtual machine). A host is a physical node which has a nova-compute service running.
 - **api** The Nova-API handles the API calls from end-users. The calls can be made directly via API-request, HTTP requests via the dashboard or CLI-tools (Command line)[34].

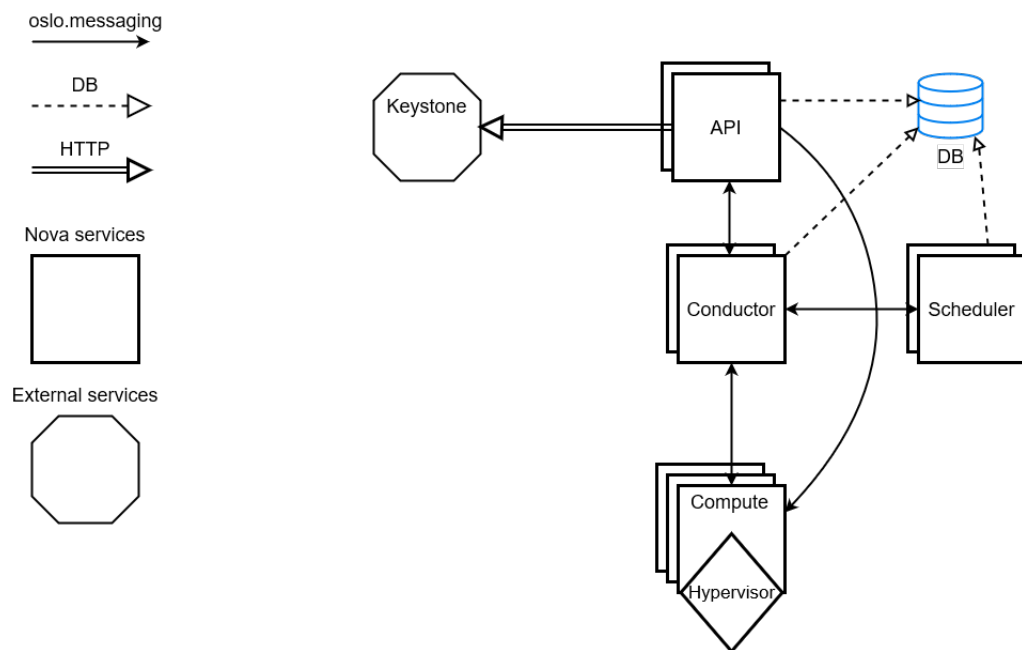


Figure 5.1: Key components of a Nova deployment

- **Nova-Conductor** The Nova-Conductor handles request that needs coordination (build/resize), acts as a database proxy, or handles object conversions[35]. Since it acts as a database proxy it does not reside on a Nova-Compute node, since that would negate the security properties of removing database access from the compute-node and moving these to the Conductor[36].

5.2 Hypervisor

Before the hypervisor computers could mainly operate one singular operating system (OS) at a time, this made them stable since the hardware only had to handle request from one operating system[37]. However this approach has the downside of wasted resources, this is mainly due to usage of one OS does not make use of all the resources available[37].

This is solved by using a hypervisor, which is a software layer that adds support of running multiple OS along with each other on one single machine (Virtual machines, VM)[37]. They all share the resources of the physical machine and the OS does function in the same way as it would if not run on top of a hyper-

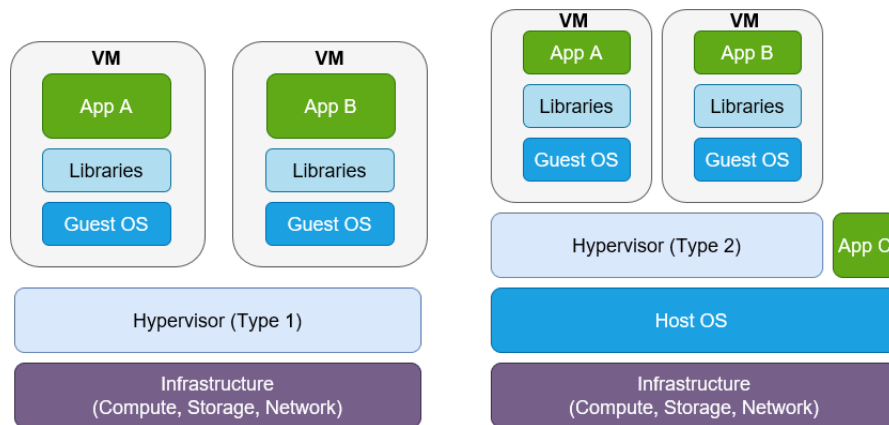


Figure 5.2: Type 1 and 2 hypervisors

visor. Since the OS are running on the same machine at the same time there is a need for isolation, this is handled by the hypervisor, it separates the VM from interfering with each other[37].

There are two different types of hypervisors, often referred to as Type 1 & Type 2. Type 1 is the kind of hypervisor that runs directly on top of the hardware, this means that it interacts directly with the CPU, storage and memory[37]. This is depicted in figure 5.2 to the left. Type 1 hypervisors are common in an enterprise data centre or other server-based environments[38].

The Type 2 hypervisor needs an underlying OS to function and it runs as an application on top of this OS. This is suitable for individual PC users that have the need for running multiple operating systems on their machine [37] and is depicted in figure 5.2 to the right.

The hypervisors are central for the cloud-based industry since it lets the cloud provider utilize the same physical resources (CPU, memory, network, and storage) for multiple users. The hypervisors provide an “abstraction of computer resources” where the resources are pooled together and enable users to obtain the resources on demand.

KVM and QEMU

Kernel-based Virtual Machine (KVM) is a virtualisation module in the Linux kernel which allows the kernel to function as a hypervisor. All hypervisors need some operating system-level components (memory manager, I/O stack, device drivers, and more) to function, and since KVM is part of the Linux

kernel it has all these parts[38]. KVM is a Type 1 hypervisor and was merged with the mainline Linux kernel in 2007[38]. With KVM it is possible to run multiple virtual machines, running unmodified Windows or Linux images[39]. Each virtual machine gets private virtualised hardware such as network-card, storage disk, graphics adaptors etc[39]. KVM functions together with QEMU which provides device emulation and virtualizer. QEMU takes on the part of emulating hardware, which means that you can run an operating system build for a certain type of machine (e.g. an ARM board) on top of your machine of choice (e.g. your X86 PC) but also device emulation such as network-interfaces, soundcard or videocards are emulated. The device emulation in QEMU is, however, software-based, this means that every CPU instruction needs to be translated from ARM to X86 instructions. This is done in software and has a certain performance penalty to it.

However together with KVM, QEMU can make use of CPU extensions (HVM), for the virtualisation and this lets the emulation reach near-native performance when executing guest-OS code directly on the host machine when the target architecture is the same as the host architecture.

5.2.1 Instances

An instance is the same as a virtual machine (VM) and is used for reducing overhead. With the Hypervisor, it is possible to run multiple instances on one server. Use cases could be to run outdated software or the fact that the instance is encapsulated and if anything were to go wrong is easy to perform recovery. This also means that if one instance is to go down, this does not affect the other running instances.

5.3 SELinux & sVirt

Security enhanced Linux (SELinux) was introduced to the open-source community in late 2000 and was integrated into the Linux kernel in 2003[40]. It was originally developed by the NSA to demonstrate how mandatory access control (MAC) could be added to an operating system and the value of a flexible MAC[41].

SELinux applies the principle of least-privilege on both users and processes. By default, everything is denied and policies are added to provide each element of a system (service, users or program) only access to the specific parts needed for the specific element to function[42], and if the element were to try

and access or modify files outside of its policy, the access is denied and action is logged[42].

The default policy is the **targeted** policy which follows the model of least-privilege model. This policy 'targets' a selected system process and confines it[43]. By default a logged-in user would be run in the *unconfined_t* domain and a system process which is started at init would be run in a *initrc_t* domain. Both of the domains *unconfined_t* and *initrc_t* are unconfined which results in the members of these domains are unable to allocate writable memory and execute it by default. This reduces vulnerabilities such as buffer overflow attacks[43]. The processes that are confined runs their own domain, for example the *httpd_t* domain is connected to the *httpd* process. The reason for this is that the domain specifies what a certain process is allowed to do, and if a confined process would be compromised by an attacker, the possible damage an attacker could do would be limited[43]. SELinux does also implement role-based access control (RBAC).

When it comes to securing virtualisation, sVirt integrates SELinux and the virtualisation. sVirt uses process-based mechanisms and restrictions to provide extra security for the virtualized guest instances, in other words, it uses the security framework provided by SELinux to add MAC to the host, this secures the system from bugs in the hypervisor which could be used to target the host or another virtual guest instance. OpenStack recommends using sVirt to harden QEMU[44].

5.3.1 Secure Virtualisation

AMD Secure Encrypted Virtualisation (SEV), is a protection against the hypervisor snooping of the guest memory. It uses encryption to separate the guest instances and the hypervisor from each other[45] so neither has access to the resources of the other part[46]. If an attacker has admin privileges on the host and would try to read the data from one of the instances, the data is encrypted[45].

Chapter 6

Results

In this chapter, the feature matrix and attack list will be presented in section 6.1 and in section 6.2 HypervisorLang be presented together with coreLang.

6.1 Feature Matrix and Attack Lists

6.1.1 Feature Matrix

In the Feature matrix, assets tied to HypervisorLang are presented. Covered in this matrix is also which category each asset belongs to, and also if the asset is tied to either coreLang, AwsLang (marked with A) or new by HypervisorLang. The changes described in the matrix are connected to any new code in the said asset.

Category	Asset	Adopted from coreLang without changes	Adopted from coreLang with changes	New asset for HypervisorLang	
Core	Object	X	X		
	System				
	Application	X	X		
	Information	X			
HypervisorLang	Data	X	X	X	
	Network				
HypervisorLang	NovaService		X(A)		X
	HMEnc			X	
	LSM			X	
	SELinux			X	
	QemuKVM			X	
	Instance				

Table 6.1: Matrix regarding the assets in HypervisorLang

6.1.2 Attack and defence lists

In this section, the attack lists and defences for each asset are presented. Also, a brief description of the attacks is added at the end.

QemuKVM

The following attacks are connected to the QemuKVM asset:

- A1.1 BufferOverFlow
- A1.2 OutOfBoundsReadOR-Write
- A1.3 NullPointerDereference
- A1.4 GuestInstanceDOS
- A1.5 Stop
- A1.6 Delete
- A1.7 fd_CMD_READ_ID
- A1.8 AttemptVenomFDC
- A1.9 VenomExploit

- A1.10 PatchStatus

The QemuKVM exploits are mainly reached via the Instance. The BufferOverflow[47] attack comes from improper restrictions of overwriting the memory of an application, OutOfBounds[48][49] read or write corresponds to writing outside of a designated area and nullpointerDereference[50] is tied to dereferencing a pointer which is expected to be valid. The GuestInstanceDOS step is a followup if any of A1.1 - A1.4 are executed. A1.5 Stop[51] and A1.6 Delete[52] are functions if the attacker has gained access to the CLI. The attacks A1.7-9 corresponds to the Venom attack[53], which were a vulnerability in the floppy disk controller. The outcome of this attack could lead to VM Escape and executing code on the host[54]. A1.10 is protection which describes if the system is patched correctly. Further explanations of the attack-steps can be found in Appendix tableA.1. The code for HypervisorLang can be found in appendix B and C, and also at its repository[55].

Instance

Attacks connected to the Instance asset:

- A2.1 Connect
- A2.2 Authenticate
- A2.3 FullAccess
- A2.4 Stop
- A2.5 Delete
- A2.6 DeviceEmulationExploit
- A2.7 ImproperMemoryBounds
- A2.8 OutOfBoundsRead
- A2.9 NullPointerDereference
- A2.10 Deny

An instance running on top of the hypervisor behaves similar to an instance running in AWSLang[23], therefore the attack-steps A2.1-A2.5 and A2.10 has been adopted from AWSLang. The attack step A2.6 can be reached when the attacker has compromised A2.3 and is a first step for reaching A2.7-A2.9, these

attacks correspond to an attacker exploiting the hypervisor via the Instance. Further information regarding the attacks can be found in Appendix A.2 and a graph visualising the steps are depicted in figures 6.3 and 6.4.

NovaService

- A3.1 Attempt Use CLI

With Nova CLI it is possible for an attacker to shut-down or delete instances[52][51].

HardwareMemoryEncryption (HMEnc)

Defence list tied to host hardware memory encryption:

- A4.1 read

If the host machines support memory encryption via the hardware, it can prevent an attacker with admin rights to read data which resides on the instances[45][46].

LSM sVirt

Defence list tied to sVirt:

- A5.1 fd_CMD_READ_ID

If the host were to have enabled sVirt, it would prevent further escalation outside the boundaries of the process running the Instance. In this case, the fd_CMD_READ_ID corresponds to the Venom attack and could be prevented by sVirt.

6.2 HypervisorLang

This section presents the QEMU specific model-language based on MAL. The first part presents the coreLang which models core IT-infrastructure which HypervisorLang borrows implementations from.

6.2.1 coreLang

A few coreLang assets are included in HypervisorLang and work in the following way. `Object` represents the simplest form of an asset, and can be exploited with two attack-steps: `attemptUseVulnerability`, which is the use of a Vulnerability and `deny` which represents a denial of service (DOS) attack, `Object` is also defined as abstract and can therefore only be extended and not instantiated.

`System` represents a runnable system and can host `Instances` and `Data`. It extends `Object` and to gain access to `System`, the attacker first has to connect and authenticate to said system. With access, the user can perform Denial of service attacks against the hosted instances on the system, access to data contained in the system and also attempt to connect to the command-line interface (CLI) which is connected with the Nova services orchestrating the instances. In the connection between HypervisorLang and `System`, it can be described as the "compute-node" in the OpenStack system, which hosts the instances.

The asset `Application` is the representation of runnable software which resides on the `Instances`. The application is also recursive which means that one application can host another application.

The main attack-points are `localConnect` which is a local connection to one or more applications and can be used to further gain full access to the host application. `NetworkAccess` which is a connection which can be executed via a network connection or via `codeExecution` that describes a code injection into the application. With full access to the application comes the attack-steps `read`, `modify` or `deny` that translates to the CIA triad of exposures. Further the `fullaccess` could lead to exposure of data connected with the exploited application. coreLang also models data resources such as information which resides in the `Data` asset. The asset `Data` is also associated in a recursive way, that is the modelling of data that contains data, and is also further associated with the asset `Information`. Attack-steps of the information asset is `attemptAccess` to the contained data. The `Data` asset main attack-steps are related to `deny`, `read`, `write` and `delete`.

The last asset which is included in HypervisorLang is the `Network` asset, it connects the applications to a network thus modelling network exposure connected to the running applications. The attack-steps are `physicalAccess`, which are direct access to the network, `access` that describes the possibility

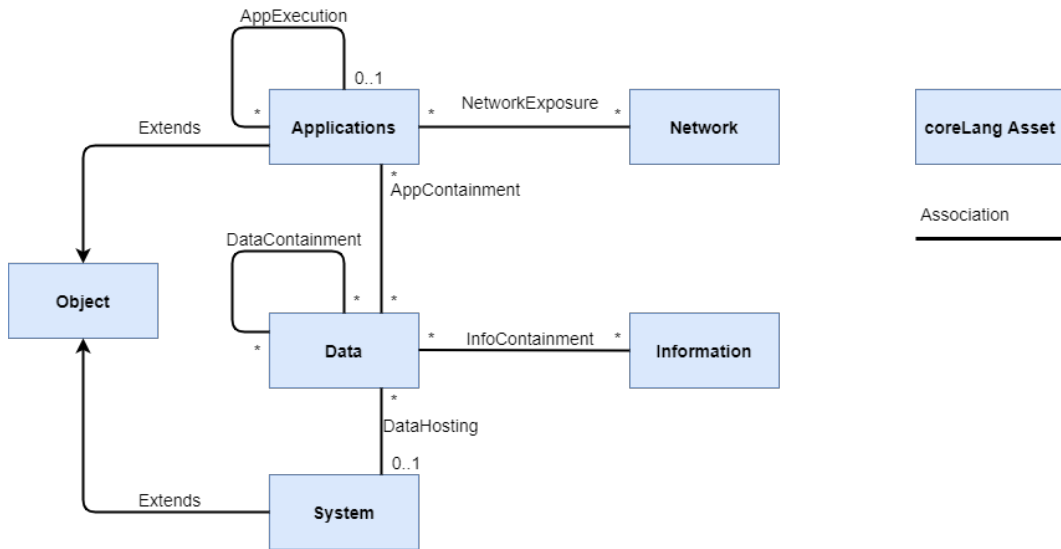


Figure 6.1: Associations between the assets used in coreLang

to connect to a network-connected-application, and at last `denialOfService`, which is a denial of network communication for an application that uses the network.

The associations between the assets are depicted in 6.1. `Object` extends to both `Application` and `System`, which leads to the inheritance of the two attack-steps mentioned earlier. `Data` is recursively associated with itself to model data which contains data, and with `Information` which leads to that data can contain information. `System` is also associated with `Data`, thus describing that `System` can hold data. `Application` is also recursively associated to itself describing that one `Application` can execute one or many `Applications`, `Data` is also associated to `Application` since an application or program can contain data. The `Network` is associated with `Application` which is described as network exposure, meaning that an application can be connected to a network.

6.2.2 HypervisorLang

The `HypervisorLang` is an extension of the `coreLang` with the additional assets, `LSM` (Linux security modules), `NovaService`, `Instance` etc. These modules are key components for modelling the compute-node. This section describes the different assets and their associations.

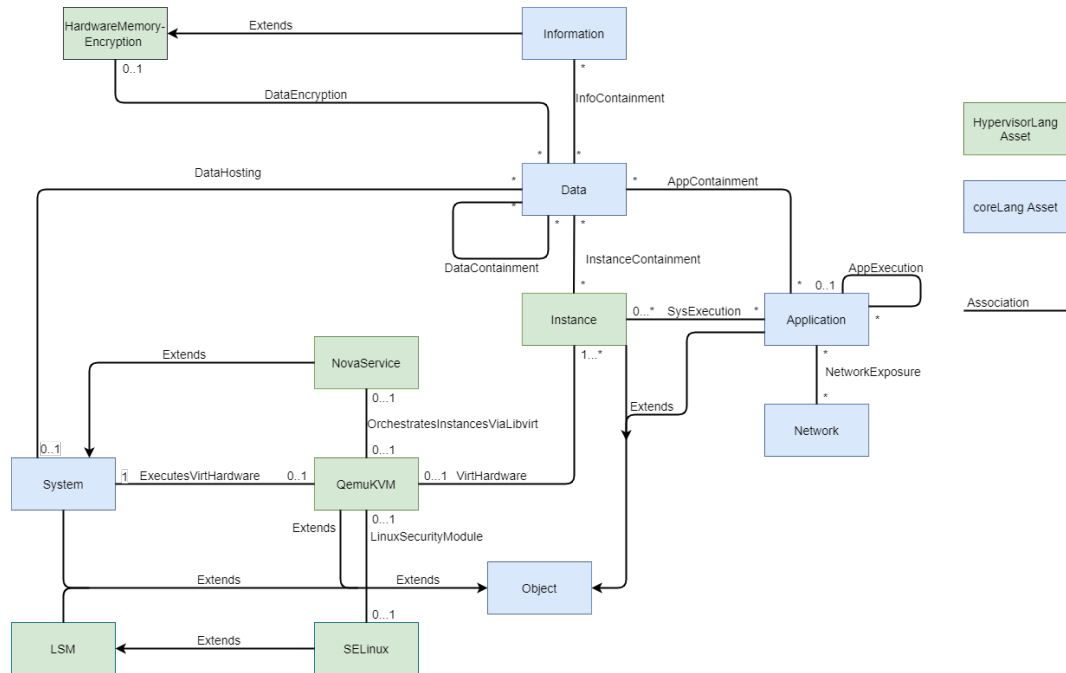


Figure 6.2: Associations between the assets used in HypervisorLang

Associations

The system executes one QEMU process which in turn can run multiple Instances. The Instance can host data and are therefore associated with the Data asset. NovaService controls QEMU, however, the model is simplified and therefore Libvirt is not described in the model. SELinux is associated with QEMU to provide sVirt (MAC). This is depicted in figure 6.2.

HardwareMemoryEncryption

The HardwareMemoryEncryption asset is a hardware related defence feature and added if the server processor support techniques such as AMD-SEV. The defence connected to this asset is shown in table 6.2 and the associations in figure 6.2.

NovaService

The NovaService is the helper daemon running on a compute-node, it can create or terminate instances. In this case, since it resides on a System, the

Prevents Attack	Attacked Asset	Description
A4.1 read	Data	The hardware memory encryption prevents snooping via the hypervisor

Table 6.2: Defenses tied to hardware encryption.

Attack Name	Attack Type	Description
A3.1 attemptUseCLI	AND	Using the command line, an attacker can stop or delete an instance

Table 6.3: NovaService Attacks.

attack-steps are similar to the System-attack-steps. Therefore the `NovaService` extends the system. To make use of the command-line interface (CLI) connected to the `NovaService`, the attacker first needs to have admin or root privileges in the system, and before this can be accomplished the attacker would have been connected and authenticated to the system. The attack is listed in table 6.3 and the associations are depicted in figure 6.2.

LSM & SELinux

The LSM asset is added in the model to further include software-related features. In this case, SELinux extends LSM, to provide the possibility to model activated SELinux on the host. `sVirt` is a framework used to harden the virtualized instances and is modelled via the association between LSM and `QemuKVM` in `HypervisorLang`, this is depicted in figure 6.2. In `HypervisorLang` `sVirt` is used to prevent break-out-attacks from the instances listed in table 6.4.

QemuKVM

The QEMU KVM asset is two of the core parts in the compute-node. As mentioned earlier QEMU handles device emulation to the Instances. For this

Prevents Attack	Attacked Asset	Description
A5.1 fd_CMD_READ_ID	QemuKVM	Enabled <code>sVirt</code> would prevent further escalation outside the boundaries of the process running the Instance.

Table 6.4: SELinux Defense.

DSL the attack surface from the instances towards the QEMU module. The attacks are listed in table A.1 found in appendix A and are depicted in figure 6.3. The associations are depicted in figure 6.2. The attacks are collected via the survey conducted by Rajendra Patil and Chirag Modi[11]. Mainly tied to the HypervisorLang the attacks in the QemuKVM assets are depicted in figure 6.3 which presents the steps from the Instance to the host. The Venom attack could be used to break out of an Instance[54][53]. In figure 6.3 the defences sVirt and patchStatus depicted to prevent the Venom attack. As for the other attacks patchStatus is the only defence.

Instance

The instance is a guest virtual machine running on the host. It extends object and the modelled attacks are depicted in figure 6.3 and 6.4. It is also associated with Data since it can hold data, and QEMU which handles the emulation for each instance, and it is also associated with Application since an instance can host applications, all the associations are depicted in figure 6.2. The first attack step depicted in figure 6.3 is A2.3 or FullAccess, the earlier steps Connect (A2.1) and Authenticate (A2.2) is not depicted in this model due to keeping the size down. Also the coreLang and AWSLang specific attacks steps such as read and write data is also not depicted. When an attacker has gained FullAccess to the Instance, the attacker can attempt to exploit the device emulation via BufferOverflow[47], Out-Of-Bounds-Read[48][49], Null-Pointer Dereference[50] or the Venom Attack[54][53]. As depicted in 6.3 the first three attacks lead to denial of service, which in turn affects the Instance locally with the Deny(DOS) step. In figure 6.4 the attack steps mainly come from coreLang and AWSLang, with fullAccess it is possible to reach the Data asset since the assets are associated, this leads to the possibility to further read, write or delete the Data. As for protection, the defence dataNotExist refers to if the data does not exist. The attack step write in figure 6.4 leads to delete since altering the data can lead to its corruption. When read is reached this leads to the possibility to access information contained in the data.

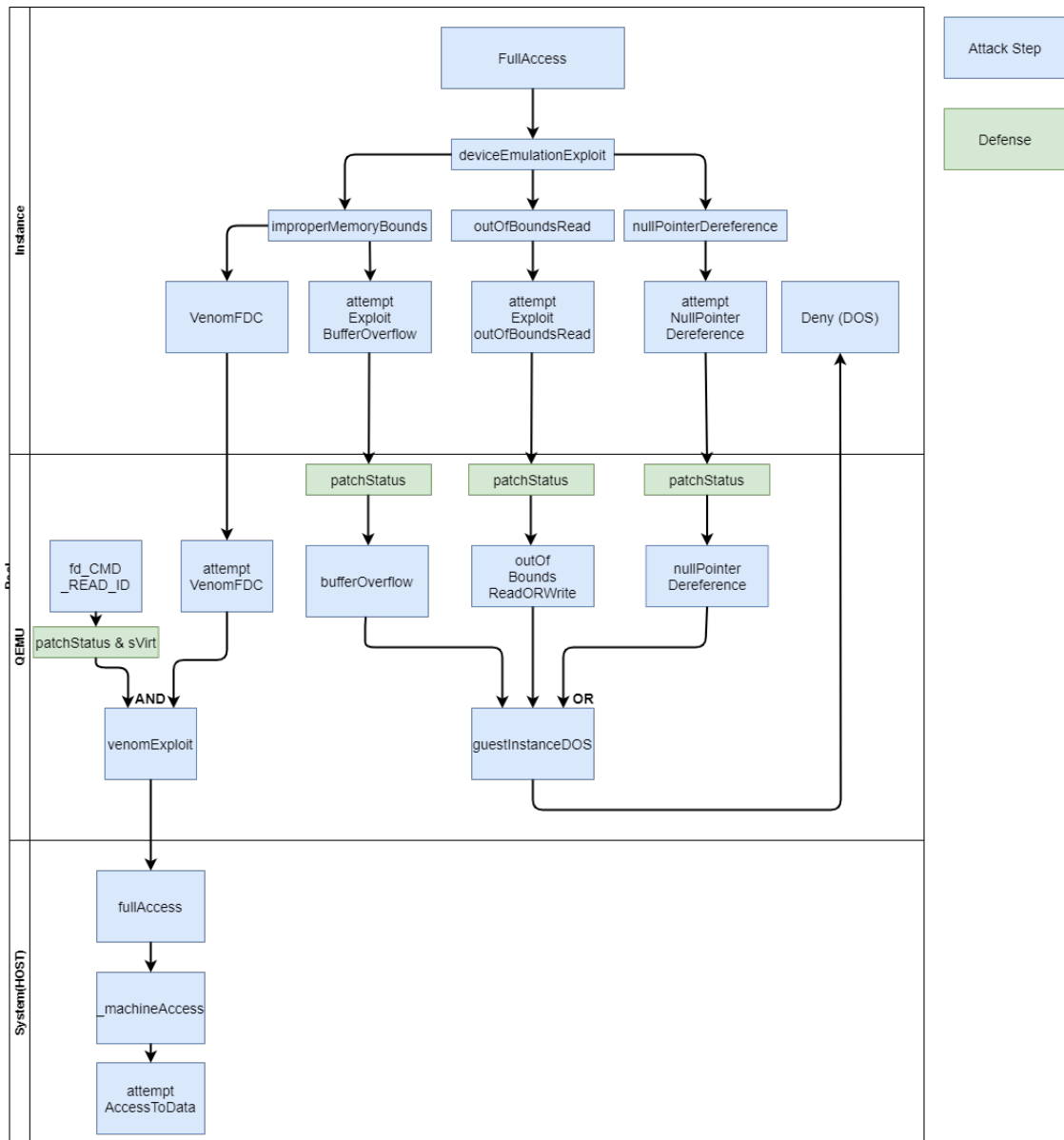


Figure 6.3: Depiction of attack-steps from instance to host in HypervisorLang

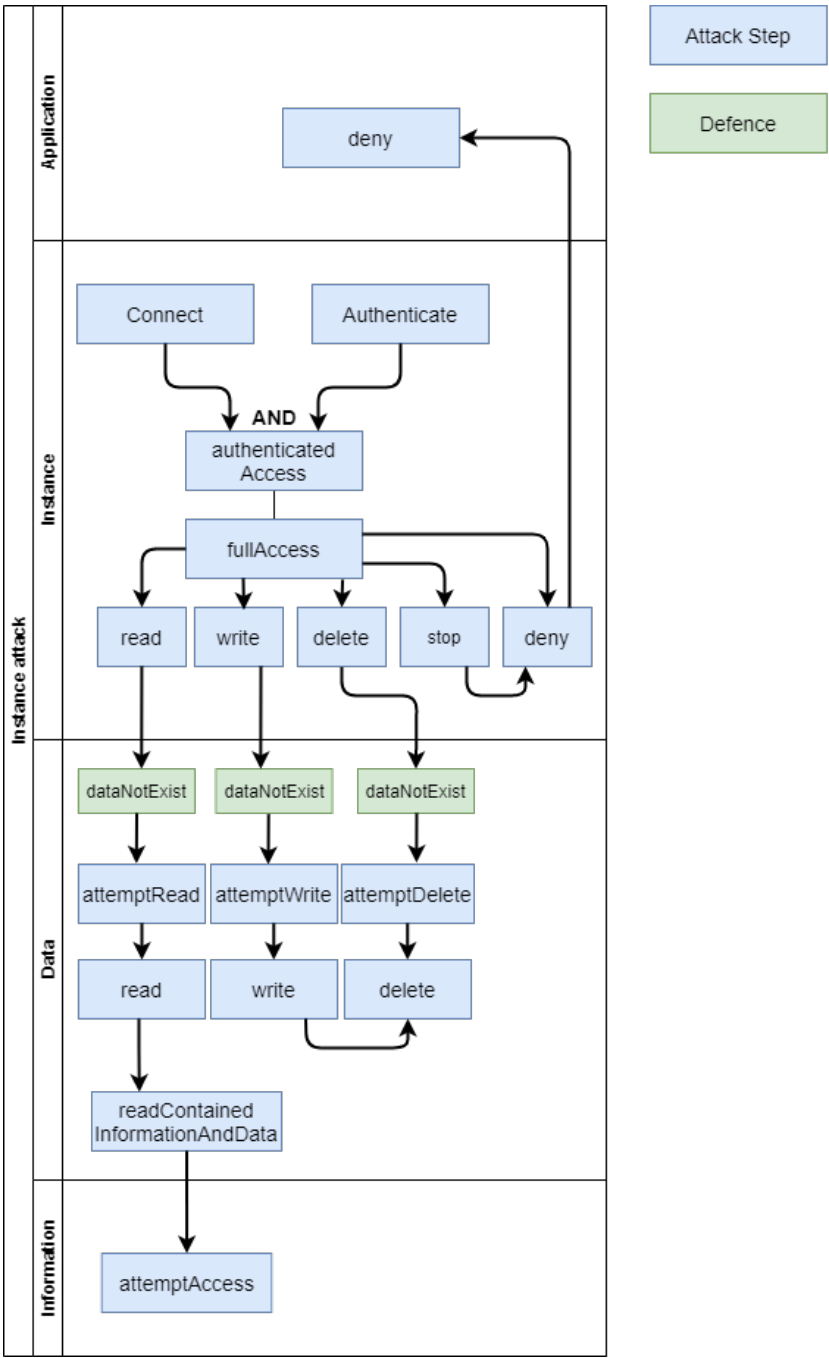


Figure 6.4: Depiction of attack-steps from instance to data in HypervisorLang

Chapter 7

Evaluation

This chapter presents how the evaluation was performed in section 7.1 together with three test cases used to evaluate HypervisorLang.

7.1 Evaluation of Testcases

For the evaluation of HypervisorLang, the DSRM methodology proposes some different ways of evaluating the artefact. As for creating the HypervisorLang, the artefact is similar to the creation of source code and therefore evaluation of HypervisorLang can be done through testing. As for the test cases, they are divided into two parts. The first part of tests is unit testing, created to evaluate the individual assets in such a way showing that they perform as designed. The second part of testing is to create test cases which are tied to the attack-steps in the model and can traverse through different assets. A complete list of test cases can be found in appendix A with a description of the scenario tied to each case and the code for each case. The code for the three test cases in this chapter can be found in appendix D. The full repository with test cases can be found at[55].

7.1.1 Example test case 1 - Access to Instance

The first test case assumes an attacker gains access to one running Instance via the attack steps A2.1, A2.2 and A2.3. Once access has been made, it is assumed that the attacker has full privileges on the instance which results in the possibility for the attacker to make use of the attack steps A2.4, A2.5 which are connected to stopping or removing the instance. Also, coreLang specific attack steps are available which are tied to the data on the instance, the fullAccess

gives a possibility to perform read, write and deletion of data on the instance. The test case is depicted in figure 7.1, and the full list of attacks tied to the instance can be found in Appendix A.2. The consequences of this attack are:

- Access to data contained on the instance.
- Stopping applications running on top of the instance, causing a denial of service.
- Deletion of data on the instance, resulting in a denial of service and loss of data.

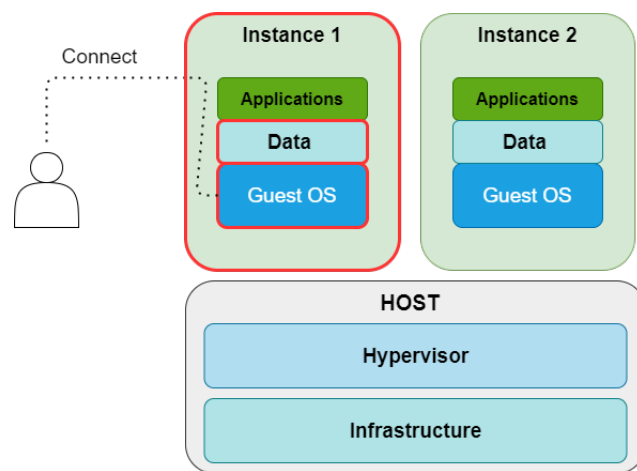


Figure 7.1: Depiction of test case 1. Where the attacker targets the Instance.

7.1.2 Example test case 2 - Breakout from Instance

In this test case, the attacker has access to the instance(attack steps A2.1-A2.3) which implies that the attacker has full access(root), and attempts to break out of the instance(attack steps A1.7 & A1.8 Together with A2.6), this would result in gaining access to the process running the instance on the host, which also means that an attacker can gain full privileges on the host. The consequences of the attack are that the host is compromised. This is depicted in figure 7.2.

- Access to data contained on the instance.
- Gaining high privilege on the host.
- Gain access to data tied to the instances running on the host.
- Stop or remove instances running on host causing a denial of service, loss of data.

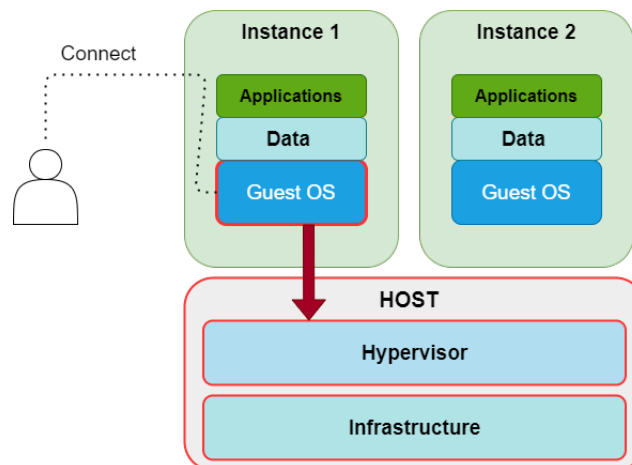


Figure 7.2: Depiction of test case 2. The attacker targets the hypervisor from the instance.

Preventions and defences added to the model which could prevent such attack are sVirt and Patches. With these defences active the breakout procedure via the Venom attack is prevented in this model.

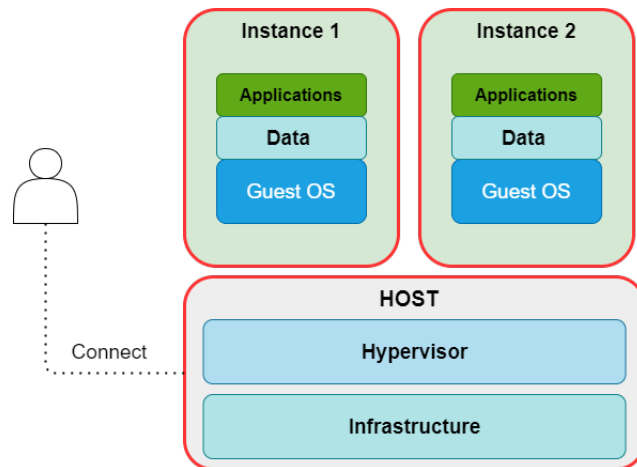


Figure 7.3: Depiction of test case 3. Where the attacker targets the Host

7.1.3 Example test case 3 - Access to host

The third test case is if an attacker or malicious admin where to gain access to the host. With such access, the consequences could lead to:

- Access to data on the host.
- Access to data tied to instances running on the host.
- Access to the NovaCli.

In test case 3 the system is compromised, in the test case data is associated with both the system and instances. However, when using encryption as a defence, the attacker does not gain the credentials to the encrypted data. Since this is handled by the hardware. The attack is depicted in figure 7.3.

Chapter 8

Discussion

This chapter covers a discussion regarding the results of the previous chapter. In section 8.1 the completeness of HypervisorLang is covered, in 8.2 the completeness of attack lists and defences are discussed. In 8.3 a discussion regarding the completeness of testing is discussed and section 8.4 covers a reflection regarding the usage of MAL as a base for HypervisorLang. 8.5 covers the discussion tied to the works of Hypervisorlang

8.1 Completeness of HypervisorLang

During the initial phase of the study, it was needed to limit the scope of the study due to the many components in the compute node. However, the main components of the virtualisation are covered on a high level. But since the thesis is limited to a few core parts there are still important components such as the Scheduler, Conductor, API-service and also Keystone which is an external service provided by OpenStack used for authentication. The model would also benefit from modelling storage units thus abling the use of persistent storage. Further, the Network module is also limited and could be expanded to describe a complete network, covering access between different host. If these components were to be added it would benefit the model by providing services such as moving an instance to another host, thus changing the data from 'at use' to an 'at rest' state.

8.2 Completeness of Attack Lists and Defenses

During the literature phase, the works by Rajendra Patil and Chirag Modi[11] was at great help, not only was the vulnerabilities sorted by the affected compo-

nent but also source and type of attack. However, the works of implementing these vulnerabilities were performed by examining literature. The work regarding the attack-lists would benefit by being evaluated further by an expert in this domain to validate how accurate the model performs. Regarding the defences there exists more security systems which could be added to the model, such as AppArmor could be added to visualise that different protections exist.

8.3 Completeness of testing

The included attack scenarios in HypervisorLang have been evaluated through testing to show that the model performs as intended. However, to further validate the model an interview with an expert in virtualisation field could help verify the correctness of the scenarios described in the model, this would also add a stronger confirmation to the cases implemented in HypervisorLang.

8.4 Using MAL as a base for HypervisorLang

HypervisorLang is based on MAL and borrows key parts from coreLang to model some of the modules needed in a cloud setup. One issue when creating the model is that previous coding experience is needed, in my opinion, the MAL syntax is easy to understand and apply, but could be difficult for inexperienced users. A different approach to solve this part could be to use securiCAD[56], which is a modelling tool made to visualise the process of describing the infrastructure.

8.5 Regarding the works of HypervisorLang

As for the creation of HypervisorLang the works can be used as a conceptual model to ‘visualise’ the components of a virtualisation layer and how the parts interact. Because of this, we believe that the conducted work does simplify the complexity of a cloud system by creating this model and therefore it might help a service provider in some extent to understand how the parts interact and how the system works. Further HypervisorLang also models vulnerabilities and defences which can supply the audience of the created model with understanding on why patching and applying security features to a cloud setup is needed and what could happen if an end-of-life situation of a component were to occur. As for accuracy, when creating a Domain-specific language with MAL the model can be tested with unit tests and integration test to validate the model. In some

extent, we would argue that security knowledge is one of the most important parts, but it is as important to have the tools to explain such information in a simplified way to the audience. Using MAL to build the model has some positive effects when the first model is deployed and if the source-system collects an update or new parts are added to the system, it is possible to add these components to the model without redoing the earlier work. Also, the language is written in such a way that it is easy to understand the code.

Chapter 9

Conclusion and future work

9.1 Conclusion

As concluded earlier HypervisorLang has some flaws regarding testing, validation and completeness. Technical work is needed to validate the model and further add functionality to reach a more complete phase of the domain-specific language. As for now, HypervisorLang can be used to visualise some virtualisation components together with attacks and defences in such way it can aid the audience to understand the virtualisation parts of the cloud system, even though the system is complex. The usage of MAL for the model is in our view a good choice since it has a wide variety of functions and the syntax is easy to follow.

9.2 Future work

As for the model, it is only a description of the virtualisation layer. A more complete model would also incorporate how compute nodes work together and also include the OpenStack services Nova-Scheduler, Nova-API and Nova-Conductor.

At the moment the model focus on "data-at-use", however, there are functions in the Nova-Compute to snapshot, stop, or migrate instances to different nodes, these steps transform the "data-at-use" to "data-at-rest" and modelling the "data-at-rest" would possibly uncover further vulnerabilities.

Also further adding the OpenStack components such as Keystone which is used for authentication, Cinder & glance used for storage, Neutron for provisioning networks and Placement which keeps tracking inventory. Adding these components could add up to a more complete model.

Regarding the Network Asset, it is only supporting connection to applications, however, the implementation of the network on an OpenStack cloud deployment would be more advanced consisting of multiple networks and subnets.

Another DSL of the OpenStack cloud environment has also been created[57], modelling the different services which are supported. As for completing the OpenStack model, merging the modelling of the hypervisor to a broader DSL could further complete the language.

Bibliography

- [1] *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17% in 2020*. <https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020>. Accessed: 2020-06-07.
- [2] Kanchan Kamila. “Role of cloud computing in modern libraries: A critical appraisal”. In: *Int. J. Inf. Libr. Soc* 2.1 (2013).
- [3] *Rackspace Private Cloud Powered by OpenStack*. <https://www.rackspace.com/openstack/private/openstack>. Accessed: 2020-05-16.
- [4] *Web Solutions Inspire Cloud Computing Software*. https://spinoff.nasa.gov/Spinoff2012/it_2.html. Accessed: 2020-05-16.
- [5] *What is OpenStack*. <https://www.openstack.org/software/>. Accessed: 2020-05-16.
- [6] *Red Hat OpenStack Platform*. <https://www.redhat.com/en/technologies/linux-platforms/openstack-platform>. Accessed: 2020-05-16.
- [7] Wenjun Xiong and Lagerström Robert. “Threat Modeling – A Systematic Literature Review”. In: *Computers & Security* 84 (Mar. 2019), pp. 53–69. DOI: 10.1016/j.cose.2019.03.010.
- [8] A.O. Baquero, Andrew Kornecki, and Janusz Zalewski. “Threat modeling for aviation computer security”. In: *CrossTalk* 28 (Jan. 2015), pp. 21–27.
- [9] Ekstedt. M et al. *coreLang*. 2019. URL: <https://mal-lang.org/coreLang/index.html> (visited on 06/18/2020).

- [10] Robert Lagerström et al. “A probabilistic attack simulation language for the IT domain”. In: (). URL: https://www.gramsec.uni.lu/preproceedings/GramSec_2020_paper_7.pdf.
- [11] Rajendra Patil and Chirag Modi. “An Exhaustive Survey on Security Concerns and Solutions at Different Components of Virtualization”. In: *ACM Comput. Surv.* 52.1 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3287306. URL: <https://doi.org/10.1145/3287306>.
- [12] Pontus Johnson, Robert Lagerström, and Mathias Ekstedt. “A Meta Language for Threat Modeling and Attack Simulations”. In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ARES 2018. Hamburg, Germany: Association for Computing Machinery, 2018. ISBN: 9781450364485. DOI: 10.1145/3230833.3232799. URL: <https://doi.org/10.1145/3230833.3232799>.
- [13] *Choosing a hypervisor*. <https://docs.openstack.org/arch-design/design-compute/design-compute-hypervisor.html>. Accessed: 2020-06-10.
- [14] Jan Jürjens. “UMLsec: Extending UML for Secure Systems Development”. In: *Proceedings of the 5th International Conference on The Unified Modeling Language*. UML ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 412–425. ISBN: 3540442545.
- [15] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *Proceedings of the 5th International Conference on The Unified Modeling Language*. UML ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 426–441. ISBN: 3540442545.
- [16] Diego Perez-Botero, Jakub Szefer, and Ruby Lee. “Characterizing hypervisor vulnerabilities in cloud computing servers”. In: May 2013, pp. 3–10. DOI: 10.1145/2484402.2484406.
- [17] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. “DAG-Based Attack and Defense Modeling: Don’t Miss the Forest for the Attack Trees”. In: *Computer Science Review* 13 (Mar. 2013). DOI: 10.1016/j.cosrev.2014.07.001.
- [18] Bruce Schneier. “Attack Trees”. In: *Dr. Dobb’s Journal* 24.12 (1999). Accessed: 2020-06-14, pp. 21–29.
- [19] Barbara Kordy et al. “Foundations of Attack–Defense Trees”. In: vol. 6561. Sept. 2010, pp. 80–95. DOI: 10.1007/978-3-642-19751-2_6.

- [20] Xinming Ou, Sudhakar Govindavajhala, and Andrew Appel. “MulVAL: A logic-based network security analyzer”. In: (July 2005), pp. 8–8.
- [21] Artz Lyle. “NetSPA : a Network Security Planning Architecture”. In: (Mar. 2006).
- [22] N. Poolsappasit, R. Dewri, and I. Ray. “Dynamic Security Risk Management Using Bayesian Attack Graphs”. In: *IEEE Transactions on Dependable and Secure Computing* 9.1 (2012), pp. 61–74.
- [23] Amandeep Singh Viridi. “AWSLang: Probabilistic Threat Modelling of the Amazon Web Services environment”. In: (2018). URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-254329>.
- [24] Peter Mell and Timothy Grance. “The NIST Definition of Cloud Computing”. In: (2011). doi: <https://doi.org/10.6028/NIST.SP.800-145>.
- [25] *Red hat What is public cloud?* <https://www.redhat.com/en/topics/cloud-computing/what-is-public-cloud>. Accessed: 2020-03-19.
- [26] *Red hat What is private cloud?* <https://www.redhat.com/en/topics/cloud-computing/what-is-private-cloud>. Accessed: 2020-03-19.
- [27] Suvda Myagmar, Adam Lee, and William Yurcik. “Threat Modeling as a Basis for Security Requirements”. In: (Aug. 2005).
- [28] O. Sheyner et al. “Automated generation and analysis of attack graphs”. In: *Proceedings 2002 IEEE Symposium on Security and Privacy*. 2002, pp. 273–284.
- [29] Teodor Sommestad, Mathias Ekstedt, and Pontus Johnson. “A probabilistic relational model for security risk analysis”. In: *Computers & Security* 29 (Sept. 2010), pp. 659–679. doi: 10.1016/j.cose.2010.02.002.
- [30] *MAL-documentation*. URL: <https://github.com/mal-lang>.
- [31] K. Peffers et al. “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3 (2007). Accessed: 2020-02-26, pp. 45–78.
- [32] *Common Vulnerabilities and Exposures (CVE)*. URL: <https://cve.mitre.org/>.

- [33] *Compute scheduler*. <https://docs.openstack.org/newton/config-reference/compute/schedulers.html>. Accessed: 2020-04-02.
- [34] *Compute API*. <https://docs.openstack.org/newton/config-reference/compute/api.html>. Accessed: 2020-04-02.
- [35] *Nova System Architecture*. <https://docs.openstack.org/nova/latest/user/architecture.html>. Accessed: 2020-04-02.
- [36] *Compute Conductor*. <https://docs.openstack.org/newton/config-reference/compute/conductor.html>. Accessed: 2020-04-02.
- [37] *Hypervisors What are hypervisors?* <https://www.ibm.com/cloud/learn/hypervisors>. Accessed: 2020-03-30.
- [38] *Red hat what is a hypervisor?* <https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor>. Accessed: 2020-04-14.
- [39] KVM. *Main Page — KVM*. [Online; accessed 19-April-2020]. 2016. URL: https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792.
- [40] *Red hat What is SELinux?* <https://www.redhat.com/en/topics/linux/what-is-selinux>. Accessed: 2020-05-22.
- [41] *NSA Security-Enhanced Linux*. <https://www.nsa.gov/what-we-do/research/selinux/>. Accessed: 2020-05-22.
- [42] *CentOS SELinux*. <https://wiki.centos.org/HowTos/SELinux>. Accessed: 2020-05-22.
- [43] *Targeted Policy*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/chap-security-enhanced_linux-targeted_policy. Accessed: 2020-05-26.
- [44] *OpenStack Security Guide*. 2020. URL: <https://docs.openstack.org/security-guide/compute/hardening-the-virtualization-layers.html> (visited on 07/12/2020).

- [45] Garry McCracken and Brent Hollingsworth. *Solving the Cloud Trust Problem with WinMagic and AMD EPYC™ Hardware Memory Encryption*. Oct. 2018. URL: <https://www.amd.com/system/files/documents/trusted-cloud-winmagic-amd-epyc-hardware-memory-encryption.pdf>.
- [46] David Kaplan, Jeremy Powell, and Tom Woller. *AMD MEMORY ENCRYPTION*. Apr. 2016. URL: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.
- [47] *CVE ID: cve-2015-5158*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5158>.
- [48] *CVE ID: cve-2017-11334*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11334>.
- [49] *CVE ID: cve-2017-13672*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13672>.
- [50] *CVE ID: cve-2017-12809*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12809>.
- [51] *Stop and start an instance*. URL: <https://docs.openstack.org/newton/user-guide/cli-stop-and-start-an-instance.html>.
- [52] *Delete an instance*. URL: <https://docs.openstack.org/ocata/user-guide/cli-delete-an-instance.html>.
- [53] *CVE ID: cve-2015-3456*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456>.
- [54] Pierluigi Paganini. *VENOM Vulnerability Opens Millions of Virtual Machines to Attack*. URL: <https://resources.infosecinstitute.com/topic/venom-vulnerability-opens-millions-of-virtual-machines-to-attack/>.
- [55] Freddy Aasberg. *HypervisorLang*. URL: <https://gits-15.sys.kth.se/pipirs/HypervisorLang>.
- [56] Mathias Ekstedt et al. “Securi CAD by Foreseeti: A CAD Tool for Enterprise Cyber Security Management”. In: Sept. 2015, pp. 152–155. doi: 10.1109/EDOCW.2015.40.
- [57] Rosander Sara. “StackLang: Automatic Attack Simulations Against the OpenStack Cloud Environment”. unpublished. N.D.

Appendix A

Test-Cases

TestCase:	TC description	Expected outcome	Status:
TC1	Attacker Gains full access to Host(System) and reads data tied to the system.	//Access to system model.system1.fullAccess.assertCompromisedInstantaneously(); model.system1._machineAccess.assertCompromisedInstantaneously(); model.system1.denialOfService.assertCompromisedInstantaneously(); //Access to data tied to system model.data1.read.assertCompromisedInstantaneously(); model.data2.read.assertCompromisedInstantaneously();	passed
TC2	Attacker Gains full access to Host(System) and NovaCli. This leads to the possibility to stop or delete Instances.	//Access to Model1 System and also to the CLI model.system1.fullAccess.assertCompromisedInstantaneously(); model.system1._machineAccess.assertCompromisedInstantaneously(); model.system1.denialOfService.assertCompromisedInstantaneously(); //Since full access, its possible to read data. model.data1.read.assertCompromisedInstantaneously(); model.data2.read.assertCompromisedInstantaneously(); //Reaching CLI model.novaCLI1.attemptUseCLI.assertCompromisedInstantaneously(); //Reaching hypervisor model.hypervisor1.delete.assertCompromisedInstantaneously(); model.hypervisor1.stop.assertCompromisedInstantaneously(); // Reaching instance1 model.instance1.deny.assertCompromisedInstantaneously(); // Reaching instance2 model.instance2.delete.assertCompromisedInstantaneously(); //Since Instances are stopped/removed following attacksteps are tied to applications or data //Instance1 -> deny-> application ->Deny. model.application1.deny.assertCompromisedInstantaneously(); //Instance2 -> delete-> data ->attemptDelete. model.data2.attemptDelete.assertCompromisedInstantaneously(); model.data2.delete.assertCompromisedInstantaneously();	passed
TC3	Attacker Gains full access to Host(System) and reads data tied to the system. However, hardware encryption is active resulting in failure to read data	//Access to system model.system2.fullAccess.assertCompromisedInstantaneously(); model.system2._machineAccess.assertCompromisedInstantaneously(); model.system2.denialOfService.assertCompromisedInstantaneously(); //Data is encrypted which means the attacker cannot access the data model.encData1.read.assertUncompromised(); model.encData2.read.assertUncompromised();	passed
TC4	Attacker gains access to Instance, hardware memory encryption is active. Since the attacker has access to the instance, data is unencrypted on said instance.	model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.encdata1.read.assertCompromisedInstantaneously();	passed

TC5	Attacker gains access to Instance, hardware memory encryption is active. The attacker breaks out of the instance via Venom attack.	<pre> model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.instance1.fullAccess.assertCompromisedInstantaneously(); model.encdata1.read.assertCompromisedInstantaneously(); model.instance1.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance1.improperMemoryBounds.assertCompromisedInstantaneously(); model.instance1.venomFDC.assertCompromisedInstantaneously(); //Hypervisor traverse model.hypervisor.attemptVenomFDC.assertCompromisedInstantaneously(); model.hypervisor.venomExploit.assertCompromisedInstantaneously(); //SystemTraverse model.system.fullAccess.assertCompromisedInstantaneously(); model.system._machineAccess.assertCompromisedInstantaneously(); /**Breakout Complete */ //Data from first instance is compromised(Access to instance, however the data to the second instance is uncompromised). model.encdata2.read.assertUncompromised(); </pre>	passed
TC6	Attacker gains access to Instance, hardware memory encryption is active together with sVirt and patches. The breakout attack is evaded.	<pre> //Instance traverse model.instance3.authenticatedAccess.assertCompromisedInstantaneously(); model.instance3.fullAccess.assertCompromisedInstantaneously(); model.encdata3.read.assertCompromisedInstantaneously(); model.instance3.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance3.improperMemoryBounds.assertCompromisedInstantaneously(); model.instance3.venomFDC.assertCompromisedInstantaneously(); //Hypervisor traverse model.hypervisor2.attemptVenomFDC.assertCompromisedInstantaneously(); model.hypervisor2.venomExploit.assertUncompromised(); //SystemTraverse model.system2.fullAccess.assertUncompromised(); model.system2._machineAccess.assertUncompromised(); /**Breakout Failed */ //Data from first instance is compromised(Access to instance, however the data to the second instance is uncompromised). model.encdata4.read.assertUncompromised(); </pre>	passed
TC7	Attacker gains access to Instance, no defenses are active.	<pre> //Instance traverse model.instance5.authenticatedAccess.assertCompromisedInstantaneously(); model.instance5.fullAccess.assertCompromisedInstantaneously(); model.data5.read.assertCompromisedInstantaneously(); model.instance5.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance5.improperMemoryBounds.assertCompromisedInstantaneously(); model.instance5.venomFDC.assertCompromisedInstantaneously(); //Hypervisor traverse model.hypervisor3.attemptVenomFDC.assertCompromisedInstantaneously(); model.hypervisor3.venomExploit.assertCompromisedInstantaneously(); //SystemTraverse model.system3.fullAccess.assertCompromisedInstantaneously(); model.system3._machineAccess.assertCompromisedInstantaneously(); /**Breakout Complete */ //Data from first instance is compromised(Access to instance, however the data to the second instance is also compromised due to Data is not encrypted). model.data6.read.assertCompromisedInstantaneously(); </pre>	passed
TC8	Access to Instance, ExploitBufferOverflow case is performed.	<pre> //Access to instance model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.instance1.fullAccess.assertCompromisedInstantaneously(); //Attempt Buffer overflow model.instance1.attemptExploitBufferOverflow.assertCompromisedInstantaneously(); model.instance1.attemptExploitOutOfBoundsRead.assertCompromisedInstantaneously(); model.hypervisor1.outOfBoundsReadORWrite.assertCompromisedInstantaneously(); model.hypervisor1.guestInstanceDOS.assertCompromisedInstantaneously(); //Successful attempt leads back to deny on instance. model.instance1.deny.assertCompromisedInstantaneously(); //Application is denied, leads to denial of data. model.application1.deny.assertCompromisedInstantaneously(); model.data2.deny.assertCompromisedInstantaneously(); </pre>	passed

TC9	Access to Instance, OutOfBoundsRead attack is performed.	<pre> model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.instance1.fullAccess.assertCompromisedInstantaneously(); //Attempt OutOfBoundsRead model.instance1.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance1.outOfBoundsRead.assertCompromisedInstantaneously(); model.instance1.attemptExploitOutOfBoundsRead.assertCompromisedInstantaneously(); model.hypervisor1.outOfBoundsReadORWrite.assertCompromisedInstantaneously(); model.hypervisor1.guestInstanceDOS.assertCompromisedInstantaneously(); //Successful attempt leads back to deny on instance. model.instance1.deny.assertCompromisedInstantaneously(); //Application is denied, leads to denial of data. model.application1.deny.assertCompromisedInstantaneously(); model.data2.deny.assertCompromisedInstantaneously(); </pre>	passed
TC10	Access to Instance, NullPointerDereference attack is performed.	<pre> //Access to instance model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.instance1.fullAccess.assertCompromisedInstantaneously(); //Attempt OutOfBoundsRead model.instance1.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance1.nullPointerDereference.assertCompromisedInstantaneously(); model.instance1.attemptNullPointerDereference.assertCompromisedInstantaneously(); model.hypervisor1.nullPointerDereference.assertCompromisedInstantaneously(); model.hypervisor1.guestInstanceDOS.assertCompromisedInstantaneously(); //Successful attempt leads back to deny on instance. model.instance1.deny.assertCompromisedInstantaneously(); //Application is denied, leads to denial of data. model.application1.deny.assertCompromisedInstantaneously(); model.data2.deny.assertCompromisedInstantaneously(); </pre>	passed
TC11	Access to Instance and defense "Patch" is active. Attacks are evaded	<pre> //Access to instance model.instance2.authenticatedAccess.assertCompromisedInstantaneously(); model.instance2.fullAccess.assertCompromisedInstantaneously(); //Attempt attempt attack hypervisor via instance model.instance2.deviceEmulationExploit.assertCompromisedInstantaneously(); model.instance2.nullPointerDereference.assertCompromisedInstantaneously(); model.instance2.attemptNullPointerDereference.assertCompromisedInstantaneously(); //Attempts to reach attacksurface on hypervisor is evaded. model.hypervisor2.bufferOverflow.assertUncompromised(); model.hypervisor2.outOfBoundsReadORWrite.assertUncompromised(); model.hypervisor2.nullPointerDereference.assertUncompromised(); model.hypervisor2.guestInstanceDOS.assertUncompromised(); </pre>	passed
TC12	Access to Instance. Attack on hosted application.	<pre> //Access to instance model.instance1.authenticatedAccess.assertCompromisedInstantaneously(); model.instance1.fullAccess.assertCompromisedInstantaneously(); //Attempt DOS on application via instance model.instance1.deny.assertCompromisedInstantaneously(); model.application1.deny.assertCompromisedInstantaneously(); model.data2.deny.assertCompromisedInstantaneously(); </pre>	passed
TC13	Access to physical network. Result in denial of service of connected application.	<pre> //Access to physical network model.netD.denialOfService.assertCompromisedInstantaneously(); //Denial of service on networkconnected application model.app1.deny.assertCompromisedInstantaneously(); model.data1.deny.assertCompromisedInstantaneously(); </pre>	passed

TC14	Access to network and has authentication to application running on network which then provide fullAccess to the application.	<pre>// Access to network application model.app1.networkConnect.assertCompromisedInstantaneously(); model.app1.networkAccess.assertCompromisedInstantaneously(); model.app1.fullAccess.assertCompromisedInstantaneously(); //Access to data model.data1.attemptRead.assertCompromisedInstantaneously(); model.data1.attemptAccess.assertCompromisedInstantaneously(); model.data1.deny.assertCompromisedInstantaneously();</pre>	passed
TC15	Access to Instance, data is missing.	<pre>//Access to instance model.instance2.authenticatedAccess.assertCompromisedInstantaneously(); model.instance2.fullAccess.assertCompromisedInstantaneously(); //Attempt attempt find missing data model.data5.read.assertUncompromised(); model.data5.delete.assertUncompromised(); model.data5.readContainedInformationAndData.assertUncompromised(); model.data5.write.assertUncompromised();</pre>	passed

Attack Name	Attack Type	Description
A1.1 bufferOverflow	OR	CWE-119: Exploiting improper restrictions of overwriting the memory of an application
A1.2 outOfBoundsReadOR-Write	OR	CWE-125: Exploits reading outside of the intended area
A1.3nullPointerDereference	OR	CWE-476: occurs when a application dereferences a pointer which is expected to be valid, but is null
A1.4 guestInstanceDOS	OR	If an instance is to go down, this would lead to DenialOfService.
A1.5 stop	OR	If an instance is to be stopped, this would lead to DenialOfService
A1.6 delete	OR	Removal of an instance would result in DOS and loss of data
A1.7 fd_CMD_READ_ID	Exists	CVE-2015-3456 or VENOM. Makes it possible to execute arbitrary code via FD_CMD_READ_ID. However this attack only exist if sVirt is unavailable
A1.8 attemptVenomFDC	OR	VENOM. This is a step to attempt use the Venom Exploit.
A1.9 venomExploit	AND	This is the venom attack step. Both attemptVenomFDC and fd_CMD_READ_ID needs to be successful for this step to be available. A successful exploit would lead to executor.fullAccess, which is admin privileges on the host
A1.10 patchStatus	Defense	Ensure the QEMU/KVM is patch & up-to-date, and the patch is installed correctly

Table A.1: QEMUKVM attacks.

Attack Name	Attack Type	Description
A2.1 connect	OR	connect to an instance, for example via SSH
authenticate	OR	A2.2 authenticate to an instance, for example via SSH
A2.3 authenticated-Access	AND	If the authentication and connection steps are both successfull, it is possible to get authenticated access on an instance
A2.3 fullAccess	OR	If an attacker has authenticated access, but needs elevated privileges, fullaccess is the next step to gain more privileges
A2.4 stop	OR	Stopping the instance would lead to denial of service of the running applications
A2.5 delete	OR	Deleting the instance would lead to denial of service and loss of data. Deletion can also refer to the removal of data kept on the Instance, since the model data is on ephemeral storage
A2.6 deviceEmulation-Exploit	OR	This is a first attack step for reaching improperMemoryBounds, outOfBoundsRead or nullPointerDereference
A2.7 improperMemory-Bounds	OR	CWE-119: Exploiting improper restrictions of overwriting the memory of an application
A2.8outOfBounds-Read	OR	CWE-125: Exploits reading outside of the intended memory area
A2.9 nullPointer-Dereference	OR	CWE-476: Occurs when a application dereferences a pointer which is expected to be valid, but is null
A2.10 deny	OR	Denial of service attack of the executed applications of the instance
A2.11 read	OR	Reading data which is contained on the instance
A2.12 write	OR	Writing to data which is contained on the instance

Table A.2: Instances Attacks.

Appendix B

Core

```
category System {

abstract asset Object
  developer info: "An object is the simplest form of an
  asset that can be compromised by a vulnerability."
{
  | attemptUseVulnerability

  | deny {A}
  user info: "The Attacker can deny some or all
  functionality of an object"
}

asset System extends Object
developer info: "Adapted from AWSLang"
{
  | connect
  developer info: "Attempt connection to the eg
  via shell, but the attacker has yet to authenticate"
  -> attemptGainFullAccess

  | authenticate
  developer info: "Does the attacker have the
  credentials to an account?."
  -> attemptGainFullAccess
```

```

    & attemptGainFullAccess
      developer info: "One way to get access to the
        machine is through legitimate authentication."
      -> fullAccess

  | fullAccess
    -> _machineAccess

  | _machineAccess
    developer info: "if access to machine, its possible to
      read data from instances."
    -> denialOfService,
      attemptAccessToData,
      subSystems[NovaService].attemptUseCLI

  | attemptAccessToData
    developer info: "Access to data on system."
    -> sysData.attemptAccess

  | denialOfService
    -> deny

}

asset Application extends Object
  developer info: "Adopted from Corelang. An application
    specifies pretty much everything that is executed or
    can execute other applications."
  {
    //No changes from CoreLang, Applications run on VM's

  | localConnect
    user info: "An attacker with low-privilege access on
      the executing instance is assumed to be able to locally
      (on the same host i.e. using loopback) interact with the
      application."
    -> localAccess,
      specificAccessFromConnection
  }

```

```

    | specificAccessFromConnection @hidden
      developer info: "This intermediate step is used to
        represent that at least one type of connect has
        happened before being able to interact locally."
      -> specificAccess

| specificAccessFromIdentity @hidden
  developer info: "This intermediate step is
    needed because if no LowApplicationPrivileges
    Identity is associated then localInteraction
    would be instantly compromised after connect"
  -> specificAccess

& specificAccess
user info: "An attacker with low-privilege access on the
executing instance is assumed to be able to locally
(on the same host i.e. using loopback) interact with
the executed applications."
-> appExecutedApps.localConnect //
  But also achieve localConnect on all child applications
  (this is something that needs to be reviewed again at a
  later stage)

| networkConnect
user info: "An attacker can connect to any network exposed
application."
-> networkAccess,
  specificAccessFromConnection

| networkRequestConnect
  user info: "The attacker has successfully sent a request
    to the application."
  developer info: "Adopted from awsLang."
  -> networkConnect

```

```

| networkRespondConnect [Exponential(0.001)]
  user info: "An attacker may be able to respond to requests
submitted by an application."
  developer info: "Adopted from awsLang."
  -> networkConnect

| authenticate
  user info: "The attacker is able to authenticate with the
appropriate credentials."
  -> localAccess,
      networkAccess

& localAccess @hidden
  -> fullAccess

& networkAccess @hidden
  -> fullAccess

| fullAccess {C,I,A}
  user info: "Legitimate access, as user or as administrator."
  -> read,
      modify,
      deny,
      appExecutedApps.fullAccess, // Gain access on all applications
executed by this (host) application
      hostApp.localConnect // and localConnect on the host
application

| codeExecution
  developer info: "Adopted from awsLang."
  -> fullAccess,
      modify

| read {C}
  user info: "The attacker can read some or all of this
service's code and data."
  developer info: "Adopted from awsLang."
  -> containedData.attemptRead

```

```

| modify {I}
  user info: "The attacker can modify some or all of
this service's data. Adopted from awsLang."
  -> containedData.attemptAccess

| deny {A}
  user info: "The attacker can deny some or all functionality
and data pertaining to this service. Adopted from awsLang."
  -> containedData.deny
}

category DataResources{

asset Information
  user info: "Represents any type of information that
might be contained inside Data."
  {
  | attemptAccess
    user info: "The attacker is attempting to access
the information."
  }

asset Data
  developer info: "Adopted from AWSlang, Encryption:
https://docs.openstack.org/project-deploy-guide/
openstack-ansible/draft/overview-storage-arch.html ,
Storage Nova: https://docs.openstack.org/project-
deploy-guide/
openstack-ansible/draft/overview-storage-arch.html"

  {
  | attemptAccess
    user info: "Attempt to access the data, this might
fail if the dataNotExist defense is used."
    -> access

& access
  -> attemptRead,

```

```

        attemptWrite,
        attemptDelete

!E dataEncrypted @hidden
  user info: "If the data are encrypted then accessing them
  requires the associated encryption credentials/key."
  developer info: "Data will be considered as encrypted if
  there is at least one Credentials instance associated with it.
  Otherwise, 'accessUnencryptedData' is reached."
  <- secureVirtualization
  -> accessUnencryptedData

& accessUnencryptedData
  user info: "If data is unencrypted then access them."
  -> accessDecryptedData

| accessDecryptedData @hidden
  user info: "Intermediate attack step to only allow effects of
  'accessUnencryptedData' on data after compromising the encryption
  credentials or encryption is disabled."
  -> access,
      readContainedInformationAndData,
      read,
      write,
      delete

# dataNotExist
  user info: "It models the probability of data actually not
  existing on the connected container (i.e. System,
  Application, Connection, etc.)."
  -> access,
      readContainedInformationAndData,
      read,
      write,
      delete

& readContainedInformationAndData
  user info: "From the data, attempt to access also the contained
  information/data, if exists."

```

```

-> information.attemptAccess,
    containedData.attemptAccess

| attemptRead
  user info: "Attempt to read the data."
-> read

| attemptWrite
  user info: "Attempt to write on the data."
-> write

| attemptDelete
  user info: "Attempt to delete the data."
-> delete

& read {C}
  user info: "The attacker can read the data."
-> containedData.attemptRead,
    readContainedInformationAndData

& write {I}
  user info: "The attacker can write to the location
of the data, effectively deleting it."
-> containedData.attemptWrite,
    delete

& delete {I,A}
  user info: "The attacker can delete the data."
-> containedData.attemptDelete

| deny {A}
  user info: "if a DoS is performed data are denied,
it has the same effects as deleting the data."
-> containedData.deny
    }
}

category Networking {

```

```

asset Network
  user info: "A network zone is a set of network accessible
  applications."
  {
  | physicalAccess {A}
    developer info: "Attacker has physical access on the network.
    This means he can cut wires/fibers and also connect using iLOs."
    -> denialOfService

  | access
    user info: "Access provides connect to all reachable
    applications."
    -> applications.networkConnect,
        denialOfService

  | denialOfService {A}
    user info: "If a DoS is performed it affects, the applications
    communicating over the network as well as the connected
    application."
    -> applications.deny
  }
}

associations {

//-----### System/Application related associations

Application [hostApp]0..1 <-- AppExecution -->
    *[appExecutedApps] Application

//-----### Data related associations
Data [containingData]*<-- DataContainment -->
    *[containedData]Data
    user info: "Data can be contained inside other data."

Data[containedData]*<--AppContainment-->
    *[containingApp]Application
developer info:

```

"An application should be able to contain some data."

```

System [system]0..1 <-- DataHosting -->
    *[sysData]Data
user info: "A system can host data."

Data [containerData]*<--InfoContainment-->
    *[information]Information
user info: "Data can contain information, as for
example credentials."
//-----### Network related associations
Network [networks] *<--NetworkExposure-->
    *[applications] Application
    user info: "An application can communicate /
    be exposed on a network."
}

```

Appendix C

HyperVisorLang

```
#id: "org.mal-lang.kvmlang"
#version: "0.0.2"

include "core.mal"

category System {

  asset HardwareMemoryEncryption extends Information{
    //If the machine supports AMD-SEV or Intel-MKTME,
    the data in use is encrypted.
    | use {C}
    user info: "Someone is using the credentials to
      perform a legitimate authentication."
    -> encryptedData.accessDecryptedData

    | attemptAccess
    user info: "The attacker is attempting
      to access the credentials."
    -> use
  }

  asset NovaService extends System
  developer info: "This is the worker daemon
    that creates or terminates VM's
    through libVirt, which further controls QemuKVM"
```

```

user info: "This is when a user has access to the
Nova Comand Line interface (CLI) / Openstack CLI which
controls the VM's"

```

```
{
```

```
| fullAccess @hidden
```

```
-> attemptUseCLI
```

```
| _machineAccess @hidden
```

```
-> attemptUseCLI
```

```
& attemptUseCLI
```

```
user info: "the user is attempting to access the CLI,
goes via the hypervisor"
```

```
-> mgmtInstance.stop,
    mgmtInstance.delete
```

```
}
```

```
abstract asset LSM extends Object
```

```
developer info: "New asset for KVM-QEMU. Linux security
module."
```

```
{
```

```
}
```

```
asset SELinux extends LSM
```

```
developer info: "New asset for KVM-QEMU. Security-Enhanced
Linux restricts the privileges of the qemu process by
establishingsecurity boundaries, so if an attacker would
compromise the hypervisor, sVirt restricts the VM's access
outside of its boundaries"
```

```
{
```

```
//developer info: "If sVirt is enabled, MAC is enforced
to the VM's running on the host. The attack cannot
proceed outside of the VM-process boundaries."
```

```
}
```

```
asset QemuKVM extends Object
```

```

    developer info: "New asset for KVM-QEMU."
    user info: "Qemu emulates vCpu, SMP, Soft MMU, I &T.
    Mech, I/O Network, Paravirtualized I/O, VM Exits, Hypercalls"
{

& bufferOverflow
developer info: "CVE-2015-5158,
CVE-2015-7504, CVE-2017-10806"
    -> guestInstanceDOS

& outOfBoundsReadORWrite
developer info: "CVE-2017-11334, CVE-2017-13672,
CVE-2017-7718, CVE-2017-15289, CVE-2015-8619, CVE-2016-
10029"
    -> guestInstanceDOS

& nullPointerDereference
    developer info: "CVE-2017-12809"
    -> guestInstanceDOS

| guestInstanceDOS
    developer info: "If an instance goes down, this means
    DenialOfService"
    -> sysExecutedInstances.deny

| stop
    developer info: "If an instance is stopped, this means
    DenialOfService"
    -> sysExecutedInstances.deny

| delete
    developer info: "Removal of an instance would result in
    DOS and loss of data"
    -> sysExecutedInstances.delete

!E fd_CMD_READ_ID
    developer info: "CVE-2015-3456, VENOM, fd_CMD_READ_ID
    attack vector"

```

```

    user info: "If SELinux is disabled, it's possible to
    proceed with the an successful attack."
    <- svirt[SELinux]
    -> venomExploit

& venomExploit
  -> executor.fullAccess

| attemptVenomFDC @hidden
  -> venomExploit

# patchStatus
  developer info: "Ensure the Qemu/KVM is patch up-to-
date,
and it's patched correctly"
  -> bufferOverflow,
      fd_CMD_READ_ID,
      outOfBoundsReadORWrite,
      nullPointerDereference
}

asset Instance extends Object
developer info: "Adapted from AWSLang, with minor changes.
One instance is the  running on the machine"
{

| connect
  developer info: "Attempt connection to the eg via
  shell, but the attacker has yet to authenticate"
  -> authenticatedAccess

| authenticate
  developer info: "Does the attacker have the credentials
  to an account?."
  -> authenticatedAccess

& authenticatedAccess
  developer info: "One way to get access to the machine is

```

```

    through legitimate authentication."
    -> fullAccess

| fullAccess
    developer info: "privileged user access, can read/Write/delete
    data, and stop an instance"
    -> read,
        write,
        delete,
    deviceEmulationExploit,
        deny,
        stop

| stop
    developer info:"The instance is stopped /shutdown result is DOS"
    -> deny

| read
    developer info:"Access to the instance can lead to the
    attacker gains acces to the data"
    -> containedData.attemptRead

| write
    developer info:"Attempts to write."
    -> containedData.attemptWrite

| delete
    developer info:"the instance is removed, since the instance uses
    ephemeral storage, removal of instance result in loss of data."
    -> containedData.attemptDelete

| deviceEmulationExploit
    developer info: " The codebase in the QEMU quick emulator,
    stands for many of the exploits."
    -> improperMemoryBounds,
        outOfBoundsRead,
        nullPointerDereference

| improperMemoryBounds

```

```

    developer info: "CVE-2015-5158, CWE-119: Improper Restriction
    of Operations within the Bounds of a Memory Buffer"
    user info: "Exploits connected to: CVE-2015-5158,
    CVE-2015-7504, CVE-2017-10806, CVE-2017-10806, CVE-
2016-3710"
    //user info: "deviceEmulationExploit -> [Vulnerability] -
>
    [Exploit] -> hypervisor.<exploit>"
    -> attemptExploitBufferOverflow,
        venomFDC

| outOfBoundsRead
    developer info: "CWE-125: Out-of-bounds Read"
    -> attemptExploitOutOfBoundsRead

| nullPointerDereference
    developer info: "CWE-476: NULL Pointer Dereference"
    -> attemptNullPointerDereference

| attemptNullPointerDereference
    developer info: "Attack Step for CWE-476."
    -> hypervisor.nullPointerDereference

| attemptExploitBufferOverflow
    developer info: "Attack Step for CWE-119."
    -> hypervisor.bufferOverflow

| attemptExploitOutOfBoundsRead
    developer info: "Attack Step for CWE-125."
    -> hypervisor.outOfBoundsReadORWrite

| venomFDC
    developer info: "CVE-2015-3456, VENOM"
    -> hypervisor.attemptVenomFDC

| deny {A}
    -> guestSysExecutedApps.deny

```

```

    }
}

associations {

//----- ### Nova/orchestrator related associations

System [system] 1..* <-- ExecutesSubSystems --
>
    0..* [subSystems]System
    developer info: "Subsystems runs on the machine"

//----- ### qemuKVM/System/Instance related associations
System [executor] 1 <-- ExecutesVirtHardware -->
    0..1 [hypervisor] QemuKVM
    developer info: "The system executes the Qemu virtulizer
and the KVM converts the system kernel into a hypervisor."

QemuKVM [hypervisor] 0..1 <-- VirtHardware -->
    1..* [sysExecutedInstances] Instance
    developer info: "Qemu-KVM handles I/O emulation CPU
emulation and virtual hardware for each instance.
Each instance have their own Qemu-process tied to the instance."

NovaService [instanceMGMT] 0..1 <--
OrchestratesInstancesViaLibvirt --> 0..1 [mgmtInstance]QemuKVM
    developer info: "Nova compute orchestrates
the KVM-QEMU instances."

//----- ### Data related associations
Data [containedData] * <-- InstanceContainment -->
* [containingInstance] Instance
    developer info: "An instance should be able to contain some data."

Instance [guestExecutor] 1..* <-- SysExecution -->
* [guestSysExecutedApps] Application
    developer info: "The instance 'guestSystem' on which

```

Applications are running."

```
Data[encryptedData] * <-- DataEncryption -->
0..1[secureVirtualization]HardwareMemoryEncryption
    user info: "Encrypted data can be associated with
    the relevant encryption credentials."
```

```
//----- ### LSM related associations
```

```
LSM[svirt]0..1 <--LinuxSecurityModule-->
0..1[hypervisor]QemuKVM
    developer info: "SELinux provides MAC framework
    for the virtual machines"
}
```

Appendix D

Testmodel

D.1 TestCase1 in Result

```
package org.mal_lang.kvmlang.test;

import core.Attacker;
import core.AttackStep;
import org.junit.jupiter.api.Test;

public class InstanceExploitTest extends KVMLangTest{

    private static class InstanceExploitModel {
        /**First model has no defenses active.
         *
         *
         * */
        /** First model */
        public final Instance instancel = new Instance("instancel");
        public final QemuKVM hypervisor1 = new QemuKVM("hypervisor1", false);
        public final Data data1 = new Data("data1", false);
        public final Data data2 = new Data("data2", false);

        /**Application*/
        public final Application application1 =
            new Application("application1");

        public InstanceExploitModel() {
```

```

        hypervisor1.addSysExecutedInstances(instance1);
        instance1.addContainedData(data1);
        instance1.addGuestSysExecutedApps(application1);
        application1.addContainedData(data2);
    }
    public void addAttacker(Attacker attacker,
                           AttackStep attackpoint) {
        attacker.addAttackPoint(attackpoint);
    }
}
@Test
public void testApplicationDenialVia_Instance1_TC12() {
    printTestName(Thread.currentThread().getStackTrace()[1]
                  .getMethodName());
    var model = new InstanceExploitModel();

    var attacker = new Attacker();
    model.addAttacker(attacker,model.instance1.connect);
    model.addAttacker(attacker,model.instance1.authenticate);
        attacker.attack();

        //Access to instance
    model.instance1.authenticatedAccess.
        assertCompromisedInstantaneously();
    model.instance1.fullAccess.assertCompromisedInstantaneously();
    //Attempt DOS on application via instance
    model.instance1.deny.assertCompromisedInstantaneously();
    model.application1.deny.assertCompromisedInstantaneously();
    model.data2.deny.assertCompromisedInstantaneously();
}

```

D.2 test2

```

public class InstanceBreakOutTest extends KVMLangTest{

    private static class InstanceBreakOutModel {

```

```

        /**Instances & Hypervisor */
public final Instance instance1 = new Instance("instance1");
public final Instance instance2 = new Instance("instance2");

public final QemuKVM hypervisor = new QemuKVM("hypervisor", false);

        /**DATA */
public final Data encdata1 = new Data("encData1", false);
public final Data encdata2 = new Data("encData2", false);
public final Data data3 = new Data("data3", false);

        /**Applications*/
public final Application application1 =
        new Application("application1");
public final Application application2 =
        new Application("application2");

        /**System*/
public final System system = new System("system");
public final NovaService novaCLI = new NovaService("novaCLI");

public final HardwareMemoryEncryption datacreds1 = new
HardwareMemoryEncryption("datacreds1");

public final HardwareMemoryEncryption datacreds2 = new
HardwareMemoryEncryption("datacreds2");

public InstanceBreakOutModel() {

    /**---First model with activated memory encryption---
    --*/

        /**SYSTEM */
system.addHypervisor(hypervisor);

        /**Instances & Hypervisor */
hypervisor.addSysExecutedInstances(instance1);
hypervisor.addSysExecutedInstances(instance2);

```

```

        /**DATA */
        encdata1.addSecureVirtualization(datacreds1);
        encdata2.addSecureVirtualization(datacreds2);
        instance1.addContainedData(encdata1);
        instance2.addContainedData(encdata2);
        //To show that the data of the instance could
            reside on the host.
        system.addSysData(encdata2);
        /**Applications tied to instances*/
        instance1.addGuestSysExecutedApps(application1);
        instance2.addGuestSysExecutedApps(application2);
        // Instance mgmt
        hypervisor.addInstanceMGMT(novaCLI);
    }
    public void addAttacker(Attacker attacker,
        AttackStep attackpoint) {
        attacker.addAttackPoint(attackpoint);
    }
}
@Test
    public void testInstance1BreakOut_TC5() {
        printTestName(Thread.currentThread().getStackTrace()[1]
            .getMethodName());
        var model = new InstanceBreakOutModel();

        var attacker = new Attacker();
        model.addAttacker(attacker,model.instance1.connect);
        model.addAttacker(attacker,model.instance1.authenticate);
        model.addAttacker(attacker,model.datacreds1.use);
        attacker.attack();

        //Instance traverse
        model.instance1.authenticatedAccess.
            assertCompromisedInstantaneously();

        model.instance1.fullAccess.
            assertCompromisedInstantaneously

```

```

model.encdata1.read.
assertCompromisedInstantaneously();

model.instance1.deviceEmulationExploit.
assertCompromisedInstantaneously();

model.instance1.improperMemoryBounds.
assertCompromisedInstantaneously();

model.instance1.venomFDC.
assertCompromisedInstantaneously();
    //Hypervisor traverse
model.hypervisor.attemptVenomFDC.
assertCompromisedInstantaneously();

model.hypervisor.venomExploit
.assertCompromisedInstantaneously();

    //SystemTraverse
model.system.fullAccess.
assertCompromisedInstantaneously();

model.system._machineAccess.
assertCompromisedInstantaneously();
    /**Breakout Complete */
//Data from first instance is compromised(Access to instance,
however the data to the second instance is uncompromised).
model.encdata2.read.assertUncompromised();
    }
}

```

D.3 test3

```

public class SystemTakeOverTest extends KVMLangTest{

private static class SystemTakeOverModel {
    /**---First model without activated defenses---*/

```

```

/**Instances & Hypervisor */
public final Instance instance1 = new Instance("instance1");
public final Instance instance2 = new Instance("instance2");

public final QemuKVM hypervisor1 = new QemuKVM
    ("hypervisor1", false);

    /**DATA */
public final Data data1 = new Data("data1", false);
public final Data data2 = new Data("data2", false);

    /**Applications*/
public final Application application1 =
    new Application("application1");

public final Application application2 =
    new Application("application2");

    /**System*/
public final System system1 = new System("system1");
public final NovaService novaCLI1 =
    new NovaService("novaCLI1");

public SystemTakeOverModel() {
    /**---First model without defenses---*/

    /**SYSTEM */
system1.addHypervisor(hypervisor1);

    /**Instances & Hypervisor */
hypervisor1.addSysExecutedInstances(instance1);
hypervisor1.addSysExecutedInstances(instance2);

    /**DATA */
instance1.addContainedData(data1);
instance2.addContainedData(data2);

```

```

        //To show that the data of the instance
            could reside on the host.
system1.addSysData(data1);
system1.addSysData(data2);
    /**Applications tied to instances*/
instance1.addGuestSysExecutedApps(application1);
instance2.addGuestSysExecutedApps(application2);
hypervisor1.addInstanceMGMT(novaCLI1);

    }
    public void addAttacker(Attacker attacker,
                            AttackStep attackpoint) {
attacker.addAttackPoint(attackpoint);
    }

    }
    @Test
    public void testMODEL1HostTakeOverReadData_TC1() {
printTestName(Thread.currentThread().
                getStackTrace()[1].getMethodName());

    var model = new SystemTakeOverModel();

    var attacker = new Attacker();
    model.addAttacker(attacker,model.system1.connect);
    model.addAttacker(attacker,model.system1.authenticate);
    model.addAttacker(attacker,model.system1.fullAccess);
    model.addAttacker(attacker,model.system1._machineAccess);

    attacker.attack();

    model.system1.fullAccess.
        assertCompromisedInstantaneously();

    model.system1._machineAccess.
        assertCompromisedInstantaneously();

```

```
model.system1.denialOfService.  
    assertCompromisedInstantaneously();  
  
model.data1.read.assertCompromisedInstantaneously();  
model.data2.read.assertCompromisedInstantaneously();  
    }  
}
```


TRITA-EECS-EX-2021:124