AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

PRACA DYPLOMOWA MAGISTERSKA

Blockchain-based vulnerabilities registry for cloud native applications

Rejestr podatności oparty o łańcuch bloków dla aplikacji cloud native

Autor: ***inż. Piotr Zmilczak***
Kierunek studiów: ***Informatyka***
Opiekun: ***dr inż. Sławomir Zieliński***
Konsultant: ***mgr inż. Marek Konieczny***

Kraków, 2020

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „ Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

……..

Abstract

Security is a key element in data processing. This also applies to more and more popular cloud native applications, the development process of which is fast and assumes automation, so the emphasis on security may be insufficient. This master thesis presents an analysis on the use of a blockchain technology to create a distributed ledger where information about vulnerabilities about cloud native applications will be stored. During the development of the system and evaluation, analyses were conducted about the performance of the proposed solution and the most popular container images were scanned, which are the basis for creating this type of application. The presented results show that the problem exists, especially if the lifetime of the image is long.

Index terms— cloud native, blockchain, containers, registry

Acknowledgements

I would like to thank my thesis supervisor, EngD Sławomir Zieliński, and M.Sc. Marek Konieczny, for their help during research, and trust that my work will be successful. I also thank to my parents, for their love, hard work and commitment. I would like to also thank to my partner, Karolina Pawlikowska, and all the others who supported and motivated me, both in my private and professional life.

CONTENTS

List of Tables

List of Figures

List of Listings

CHAPTER 1

INTRODUCTION

Nowadays one of the most popular solution to deploy applications is to package them into containers and it is natural to hear the term *container*, mentioned in any discussion about cloud native. Cornelia Davis, in her book called Cloud Native Patterns[12], observes that, "Containers are a great enabler of cloud-native software." This approach is favoured by developers and system administrators because it isolates installed features and makes deployment much more faster[42]. There is no doubt that the most common solution for creation of containers is Docker, an open-source tool set with vast community and probably one of the largest images' repository (nearly three millions of images in January 2020)[25].

From time to time we can observe news about exploits and vulnerabilities of operating systems or installed packages appearing in the Internet[22]. This type of information is taken really seriously, that is why developers fix issues of this type immediately, as well as they publish information that the patch fixing vulnerability is ready. Afterwards system administrators install updates in a host operating system and everything works like it used to. Nevertheless when it comes to containers the things look quite different.

Docker images are often created by the software developers, who not pay enough attention to the security matter, because they think this issue should be

took care by administrators[44], and the other side thinks opposite. That is why images available at Docker Hub repository contain a lot of vulnerabilities what have been already analysed and described in many papers [45][53].

1.1 Motivation and Thesis Statement

In this master thesis I would like to analyse the opportunity of using blockchain network to improve Docker container's security by providing a distributed registry of vulnerabilities for cloud native applications. The idea of this thesis is to explore and develop approaches, that will let to register, by trusted units, Docker images with information about detected exposures, but also persist and distribute already found ones between peers of the network. In my opinion, blockchain is one of the solutions for the problem of distributing the knowledge, because it provides a way to have the same state across the globe between network nodes. At the same time, it provides trust at a different levels, thanks to which blockchain-based solutions can introduce various types of trust policies for stored data.

In this thesis a system to distribute the knowledge about vulnerabilities in Docker images will be presented and it has been created with use of the blockchain. The name of the system is *bbdivr*, and it comes from first letters of Blockchain Based Docker Images Vulnerabilities Registry. It could be used by developers or system administrators to scan their images and publish this information in the distributed ledger. This approach will create a global state with the knowledge about Docker images and detected vulnerabilities in them and in the future it can be used also as the notification system to spread information between interested units about new exposures.

Before starting the conceptual work, in order to give the scope of the conducted research, the list of requirements for the system was drawn up and it is presented

as follows:

(R1) System is based on blockchain network.

(R2) System is able to persist at least two types of objects (vulnerability, image layer).

(R3) System is able to interact with at least two exposures providers, and it should be open to new ones.

(R4) System is able to convert and transfer collected data to blockchain.

(R5) System is able to interact with blockchain to get already detected vulnerabilities in Docker images.

(R6) System can be used only by identifiable user and every user has to belong to one or more groups.

(R7) System provides different types of trust levels for persisted data.

(R8) System is able to update information in the blockchain due to detected new exposures in already scanned images.

The requirements listed above are presented in order of priority, starting with the most important one.

1.2 Scope of Research and Research Challenges

The main goal was to evaluate available blockchain technologies and select those meeting introduced requirements. Then, the selected technology was used in the system that delivered the distributed registry.

An important aspect during the research for this thesis was the possibility of using blockchain technology and the analysis of the efficiency and scalability of the solution, so that the system has a real use case. Trust policies regarding access to the network as well as confirming and expressing approval for particular data stored in the system along with a mechanism limiting data modifications only to

defined operations by smart contracts made it possible to create a solution that does not have to be controlled by one organisations.

The analysis of the system performance and its usage in increasing security in cloud native applications through the integration of multiple data sources and their concatenations also allowed to determine the number of vulnerabilities in popular Docker images.

The scope of research during the work on this master thesis can be presented as the list of questions, that outlines the main aspects of the work and will be discussed and answered later in this this thesis. These questions are:

(Q1) How useful is blockchain in creation of distributed registry with different levels of trust?

(Q2) Is data about vulnerabilities from different sources connectable?

(Q3) How does the use of combined data sources about vulnerabilities affect the number of them?

(Q4) How the amount and size of data affects a blockchain-based registry?

(Q5) How many vulnerabilities is in popular cloud native applications and and what is the severity level of them?

To answer these questions, I start by comparing a blockchain solutions. Next, the general concept of the model and the *bbdivr* system are presented. This brings us to the evaluation section where the results of this work are presented.

The most challenging part of this thesis was to develop a proof of concept of a system that will be evaluated, and based on that, conclusions will have been drawn. The created system needed a test network infrastructure which will let to develop system locally and perform evaluation. Running this network on a local machine was burden to the host operating system, what could have had influence on the time spent during the development.

1.3 Main Achievements

The scope of research for this thesis concerns a wide range of technologies, therefore one of the primary achievements is a comparison of available blockchain frameworks. It can be used as a preliminary materials for the future work on a system that will also use a distributed ledger in a similar way. Another important achievement of this thesis is the proof of concept of the system that is enable to persist information about Docker images' vulnerabilities and exposures and provide different level of trust for data persisted in the network. In summary, my contributions are:

(1) I have researched and discussed the current state of art of using blockchain networks to create distributed registry.

(2) I have created the *bbdvir* system, that is designed to store and distribute information about detected exposures in cloud native applications between the blockchain network users. It is capable of interacting with miscellaneous scanning tools, that provide communication via REST Api and persisted information can be trusted with different levels.

(3) I have evaluated the system and presented the outcome about vulnerabilities in the most popular Docker images, and how the outdatedness impacts the number of exposures.

Research and work on this thesis and the developed system might be valuable in future work, therefore source code, configuration scripts and notes that are important will be distributed as an open source project on the MIT License at GitHub repository hosting. The link to the repositories is in an Appendix A section. Everyone interested will have an opportunity to study the code and use the parts of the system that will be suitable to a someone's need.

1.4 Structure of the thesis

This thesis is divided into seven chapters. Chapter 1 introduces the problem and presents main achievements and structure of the master thesis. In the next chapter (2), technological background is discussed, it describes available technologies and their main features, which had influence on the final decision about chosen blockchain network technology. Chapter 3 presents related work in three main aspects of this thesis: cloud native applications vulnerabilities, the concept of registries in computer science and blockchain-based registries. The next chapter (4) shows the general concepts of the *bbdivr* system and proposes a model to persist data in. Chapter 5 presents the most important implementation aspects and the interaction between components. Chapter 6 is dedicated to present the evaluation of the created system and researched questions. The last chapter (7) is a thesis summary and conclusions about all the work and effort on this thesis, but it also shows the available future extensions and improvements.

CHAPTER 2

TECHNOLOGICAL BACKGROUND

This chapter is dedicated to discuss the choices that have been made regarding technological background(Section 2.1) and present the main features of chosen technologies in the area of: creating blockchain network (Section 2.2), cloud native applications (Section 2.3), and detecting vulnerabilities(Section 2.4), as to be able to provide data for the created system.

2.1 Blockchain Network Selection

One of the main aspects of this thesis refers to the blockchain network, therefore the most important choice concerns technology to develop a distributed ledger. A standard blockchain system is a list of records which are stored in the form of blocks and every user of the network keeps this state by himself. Thanks to the use of cryptography and consensus algorithms, these blocks are connected, and the world state is finally the same at every node.

The most important category according to which blockchain technologies are divided is access to the network. Public blockchain networks, also called global or permissionless, allows anyone to join and read or write to the world state. This type of network is decentralised and does not have a authority which controls the

network. Another important aspect of these networks is that, they are vulnerable to attacks in which attackers take control of the network by having sufficient participation in it [18][40].

The most popular blockchain network is Bitcoin[1], which uses virtual coins that are sent between wallets. This network uses Proof of Work[2] as the consensus protocols[31], which assumes that a block with transactions can be added to the chain after complex computational mathematical problem is solved. Nodes responsible for solving this problem are called miners, who receive coins as a reward. The performance of public networks is very low, e.g. one transaction every seven seconds for Bitcoin[47].

On the other hand, we have private blockchain networks, which are also called permissioned. These type of network is managed by one authority or consortium, but it does not mean that these entities control the ledger state. Also a very important aspect is that, every user of the network is identifiable and the access to the network has to be granted to him. In such types of networks, it is possible to create closed groups to which only selected users may belong. It also allows you to enter different types of trust levels for the data. Performance is also an important element, which is much higher in private networks[35], because instead of using PoW, other consensus protocol can be used. This element is really important, because with bigger effort comes bigger energy consumption, therefore using non-PoW protocols lowers the electricity usage [41].

[1]Reynard, Cherry (May 25, 2018). "What are the top 10 cryptocurrencies?". The Telegraph. Retrieved October 15, 2018. https://www.telegraph.co.uk/technology/digital-money/top-10-popular-cryptocurrencies-2018/

[2]Proof of work (PoW) is a form of cryptographic zero-knowledge proof in which one party (the prover) proves to others (the verifiers) that a certain amount of computational effort has been expended for some purpose. Verifiers can subsequently confirm this expenditure with minimal effort on their part. (source: https://en.wikipedia.org/wiki/Proof_of_work)

	Quorum	Hyperledger Fabric	Corda
Permissioned	X	X	X
Smart Contract functionality	X	X	X
No-industry focus	X	X	
Configurable consensus protocol	X	X	X
Pluggable Consensus	X	X	X
Asset instead of cash as currency		X	
Multi-tenancy		X	X
Java for Smart Contracts		X	X

Table 2.1: Comparison of permissioned blockchain systems.

One of the requirement (R6), which was assumed at the beginning, was to be able to create private ledger in which units will be identifiable. This requirement excluded Ethereum Network[17] and other solutions based on a public access. Assuming the above requirement, the choice was still large. The selection was limited to the most popular permissioned blockchains, thereby with the biggest community[3]: Quorum[37], Hyperledger Fabric[2] and Corda[37].

All of this solutions provide possibility to create a private ledger and units verification (it is needed to provide information about who guarantees that detected exposures of images are correct). The next important feature is the ability to create

[3]Recently, one of the best indicators of popularity has been ”stars” on a remote code repository, GitHub. As of March 30, 2020, the open source project, Hyperledger Fabric has 10,1 thousands of stars, Corda project has 3,3 thousands and Quorum project has 3,6 thousands. This makes the Hyperledger Fabric the most popular taking into consideration this indicator.

smart contracts, however, this is a common feature and all the above-mentioned technologies meet it.

Nowadays, all manufactured solutions have their purpose. In this case, the creators of Corda were driven by the needs of the financial industry, and Hyperledger Fabric and Quorum was conceived as a general purpose tool.

Due to the fact that the consensus protocol has an impact on the network performance and the transcription acceptance policy, I found it important to be able to configure it as needed. All mentioned solutions allow for configuration and Table 2.2 presents available algorithms.

Blockchain Systems	Consensus protocols
Quorum	Raft-based, Istanbul BFT, Clique PoA
Hyperledger Fabric	Raft, Kafka-CFT, BFT, Solo
Corda	Raft, BFT

Table 2.2: Consensus protocols in blockchain technologies.

Another aspect by which the technologies were compared was consensus pluggability. This means that transactions is delegated to a modular component for consensus that is logically decoupled from the peers that execute transactions and maintain the ledger. In Hyperledger Fabric this task is delegated and executed on Docker container, also Corda and Quorum have pluggable consensus.

Quorum bases its data model on accounts, Corda bases it on transactions, and Hyperledger Fabric uses a key-value approach. Therefore, in order to be able to fulfill the requirement (R2), the data model should be based on key-value. Thanks to this, more complex and more than one type of objects will be able to be defined in the network.

The next important aspect was multi-tenancy. In Quorum it is not available,

Hyperledger Fabric supports it by channel mechanism and Corda is isolated and multi-tenant by design, because transactions are private and only seen by the authorised participants. A Hyperledger Fabric channel is a private "subnet" of communication between two or more specific network members, for the purpose of conducting private and confidential transactions.

The last and least condition, that was included during this selection process, was the ease of programming and creating smart contracts. Table 2.3 presents programming languages for smart contracts in selected distributed ledger technologies.

Blockchain Systems	Programming Languages for Smart Contracts
Quorum	Solidity
Hyperledger Fabric	JavaScript, Go, Java
Corda	Kotlin, Java

Table 2.3: Programming languages for smart contracts.

Certainly developing smart contracts in Java is more simple than using Solidity, because Java is well known by most of back-end developers. Results of Stack Overflow Developer Survey from 2019 [49] show that:

- JavaScript is known by 67.8% of respondents
- Java is known by 41.1% of respondents
- Go is known by 8.2% of respondents
- Kotlin is known by 6.4% of respondents

This shows that in order to reach the largest possible number of developers, It is possible that this is not an important aspect, as it is the programming language that should suit the problem rather than opposite. However, I found it could be an interesting element due to the fact that the operations defined in the smart contracts must be deterministic and their result calculated during the *execute* phase

must be the same across all peers, and Java is a non-deterministic language.

All in all, Hyperledger Fabric suits the best for the needs of this problem, because it meets criteria discussed in this section, that have been summarised and presented in the Table 2.1. In addition, it has a large developer community which makes it easy to work with as the documentation is extensive, thorough and includes many examples.

2.2 Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain network, which is a infrastructure that enables usage of ledger services and smart contracts(chaincodes) by application consumers and administrators. The main advantage of Hyperledger Fabric over other blockchain solutions that was under consideration in previous section, is that, it provides policy of accessibility and traceability of peers. In addition it provides wide range of configuration options. Hyperledger Fabric is based on removable and configurable modules, which makes it a universal tool, that can be used in various industries.

2.2.1 Consensus Protocol

The current version of Hyperledger Fabric is 2.2, this fact is important due to the changes that have occurred in this system. Versions below 1.0 used PBFT as a consensus protocol. Then version 1.0 was based on concept of ordering services with implementation based on Apache Kafka [4]. The next major version was 2.0, which deprecated Kafka-based ordering service and made Raft-based as a default con-

[4]Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. url: https://kafka.apache.org/

sensus protocol (Raft-based implementation of ordering service was introduced in version 1.4.1). There were several reasons why Kafka is no longer supported, some of which were the difficulty of managing Kafka and ZooKeeper[5], these tools are not designed to be run across large networks because they should be run in a tight group of hosts.

2.2.2 Transaction Lifecycle

The crucial difference between Hyperledger Fabric and other blockchain systems is the lifecycle of a transaction[2], the standard approach is order-execute, in which:

- order - transaction is added to the blockchain and ordered using a consensus protocol.
- execute - transaction is executed in the correct order, the same for each peer.

However the lifecycle of a transaction in Hyperledger Fabric is different and is composed of execute-order-validate phases (presented in Figure 2.1) :

- execute - clients sign and send a transaction proposal and it is executed (peers execute proposal of transaction by running smart contract logic in isolated environment using containers). The result of executing it gives a write-set (modified keys along with their new values) and a read-set of keys read during the executing proposal[6], along with their version numbers. The endorsing peer signs an endorsement message, that includes the read-set and write-set, and sends this back to the client. The client is waiting until he

[5]For the latest of Kafka version (2.6.0) ZooKeeper is still required for running Kafka, but in the near future ZooKeeper will be replaced with a Self-Managed Metadata Quorum.

[6]Hyperledger Fabric uses a key value approach to store its state. Key is pointing binary data which can be queried using it.

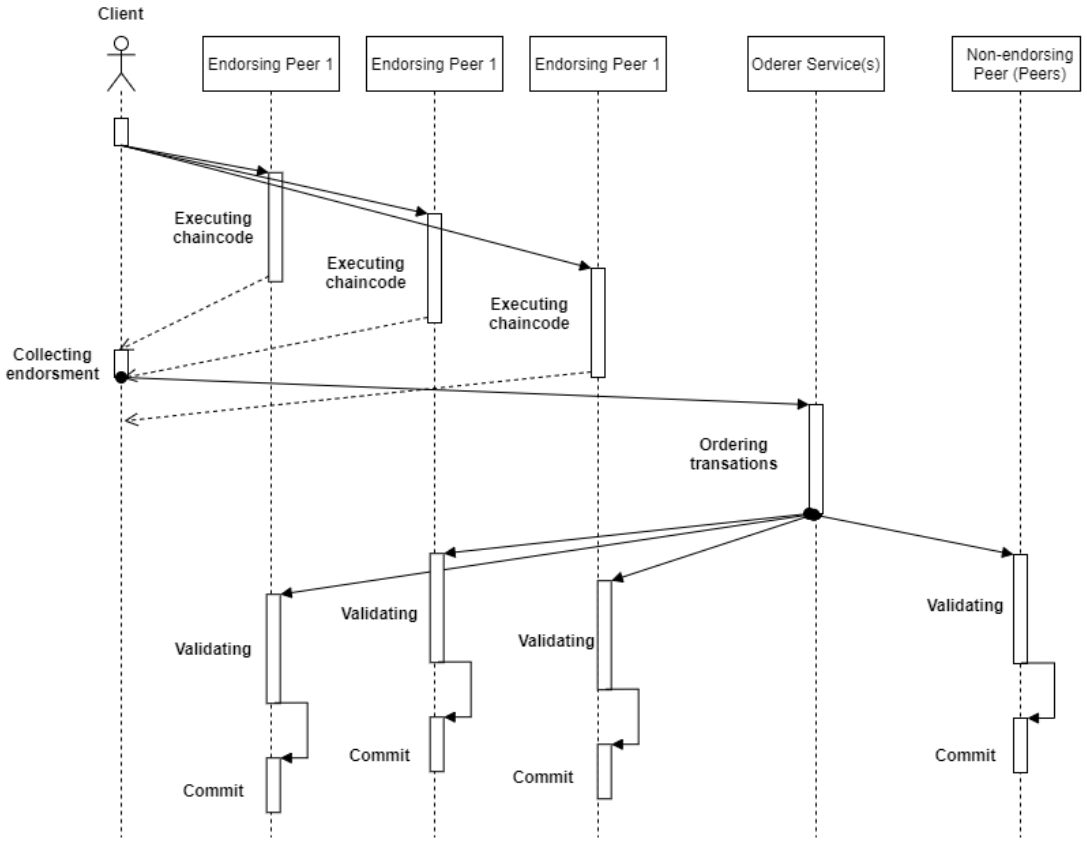

Figure 2.1: Hyperledger Fabric transaction lifecycle.

collects enough endorsements[7], what is defined in endorsement policy of the chaincode. All endorsing peers have to produce the same result of chaincode execution.

- order - when the client has collected enough endorsements, he creates a transaction and passes it to the ordering service. In this phase, submitted

[7]The endorsement system in Hyperledger Fabric was introduced with switch to execute-order-validate approach in version 1.0. It allows users to define policies around the execution of chaincode, in particular define which peers need to agree on the results of a transaction before it can be added to the ledger. This feature is very useful in defining business processes because it allows to create policies that require approval from two organisations rather than from all users of both organisations.

transactions are being ordered (endorsed transaction after being received by orderer are broadcast and consensus between other ordering nodes in the channel is established). One of the features of ordering service is batching transactions into blocks and as the results it creates a hash-chained sequence of blocks containing transactions. Hyperledger Fabric network assumes that despite the existence of many peers in the channel, only a few are responsible for determining the order. In addition the ordering service is not taking part in the persisting blockchain state, and does not validate or execute transactions. This makes protocol separated from execution and validation, therefor it can be easily replaced.

- validate - this phase begins when ordering service distributes blocks to all peers connected to it or via gossip protocol[8]. Every peer validates endorsement policy for all transactions in the block, if it is not satisfied or incorrect the transaction is marked as invalid[9]. The next step is validation of transactions to check if a later transaction has been voided by an earlier transaction (e.g. no funds because they were spent in an earlier transaction). The phase ends with committing transactions from the block to the current local ledger.

The transaction life cycle presented above is significantly different from that used in other blockchains. This makes Hyperledger Fabric very popular and used in a variety of applications.

[8]Gossip protocol in Hyperledger Fabric is responsible for the block dissemination via having peers distribute blocks among themselves instead of getting them directly from the orderer service.

[9]Hyperledger Fabric persisting even invalid transactions in a transaction log, it can be used for audit purposes, e.g. to determine what went wrong or who ordered the transaction.

2.2.3 Smart Contracts

Smart contract and chaincode in the Hyperledger Fabric community and documentation appears as interchangeably words. But to be precise smart contract is a set of instructions responsible for logic that controls the lifecycle of a business object. And the chaincode is the way smart contracts are packaged for deployment.

As mentioned in the Section 2.1, contracts in Hyperledger Fabric can be written in Java, Go or Node.js. But in one network, or even in one channel, contracts written in different languages can be combined. The contract implementation contains business logic that defines the objects as well as the actions that can be performed on them. Within one contract, other contracts can be executed even between channels.

Packaged contracts in chaincode must be added to the blockchain, and this process is described and defined by the chaincode lifecycle. This process is very important as it determines what conditions must be met in order to deploy chaincode. This cycle consists of four phases: package, install, approve and commit. This phases looks as follow:

- package - this first phase defines the creation of a TAR file with the contents of the compiled chaincode implementation and a metadata file. It can be performed by one or by each organisations, but the implantation must be exactly the same.
- install - this phase is for organisations that want to use chaincode. They need to install chaincode on every peer that should be allowed to participate in the transaction execution and endorsement. The successful installation returns label combined with a hash of the package, which is used to associate and compare the chaincode package installed on organisation's peers.
- approve - the next step is the approval of a chaincode definition in the channel by the organisations that belong there. Before the chaincode can be

started on the channel it has to be approved by a sufficient number of organisations to meet the channel's lifecycle endorsement policy[10]. Thanks to this, Hyperledger Fabric allows to set rules between organisations and that they cannot be imposed by one organisation, as is the case with standard client-server solutions.

- commit - the last step is to commit chaincode to the channel. This can only be done when the definition of chaincode has received a sufficient amount of approval. Commit is done by one of the organisations. The process is quite similar to adding a new transaction to a blockchain. First the commit proposal is sent to the peers of channel members, who check if the chaincode definition is approved by their organisations and endorse the definition if it is. Next, the transaction containing the chaincode commit proposal is sent to the ordering service, which then commits the chaincode definition to the channel. After the new definition has been committed, all peers in the channel start the new chaincode container, to be able to execute future transactions based on this new chaincode.

After successfully launching the new chaincode in the channel, it can be used. Additionally, Hyperledger Fabric introduces the possibility of updating the chaincode implementation in order to modify logic, which may change over time.

2.3 Cloud Native Applications

Cloud Native is a term used to describe technologies, that allow to create and use modern applications in a cloud (public, private or hybrid)[9]. This approach allows to create systems, that have a high degree of responsiveness. Thanks to the separation of the system into many loosely coupled elements, the replacement of

[10]Hyperledger Fabric as default use majority as lifecycle endorsement policy for chaincodes.

individual parts does not require switching off the entire system, and moreover, the development of a smaller piece makes it easier and gives the higher level of test coverage.

Containers usage allows to meet this practises by providing ideal, small application deployment unit[20][11,12]. Thanks to this, it was possible to create an abstraction that made the servers independent of the language in which the application was written, as well as the environment in which the application is run.

Containerisation technology, also called OS-level virtualisation, is an operating system paradigm in which the kernel can contain multiple isolated user space instances, which it consists of: process identifier(PID) , network, devices and user namespace. This instance is called container (by Docker users), zone, jail or virtual environments (each system names it differently). Applications running in two separate containers can see only their own container's assigned resources and devices. This approach allows system administrators to use one host to run more than one application in custom prepared, separate and isolated environments.

This technology is known since the early beginnings of UNIX systems[24]. The first attempts were made by IBM. In the 1970s this company has virtualised an operating system and called it VM/370[13][10]. This had not much use outside of mainframe computing but then it got transformed into a first commercial platform for servers called z/VM[51].

In 1982 the *chroot*[15] was added to BSD by Bill Joy and this feature was quickly implemented by most of UNIX-like operating systems. The tool called *chroot* is a command that changes apparent root directory for a current running process and its sub-processes. Chroot helps to create a separate directory tree for

[11]CNCF The Cloud Native Trail Map (`https://github.com/cncf/trailmap`) identifies containerization as the first step in the cloud native software development process.

[12]CNCF Cloud Native Interactive Landscape https://landscape.cncf.io/

[13]IBM VM History and Heritage https://www.vm.ibm.com/history/

this program and no others files outside this directory tree cannot be accessed by this process. However root user is able to escape from *chroot*, therefore it should have never suppose to be used as a security mechanism.

Before creation of Docker there was a lot of solutions that provided OS-level virtualisation like LXC[6] or Solaris Containers[28]. Docker, previously called dotCloud, was launched as a side project, and then it was introduced in Santa Clara at PyCon in 2013 and finally it was released as an open source project. During the first year Docker used as a OS-level virtualisation technology LXC[6] but quickly dropped it in favour of its own solution called *libcontainer* written in Go programming language.

Containers and virtual machines provide resource isolation and allocation benefits but containers are an abstraction at the app layer that packages code and dependencies together, and the virtual machines are an abstraction of physical hardware, turning one physical machine into many servers. Each virtual machine needs its own operating system and containers running on a single host (Figure 2.2).

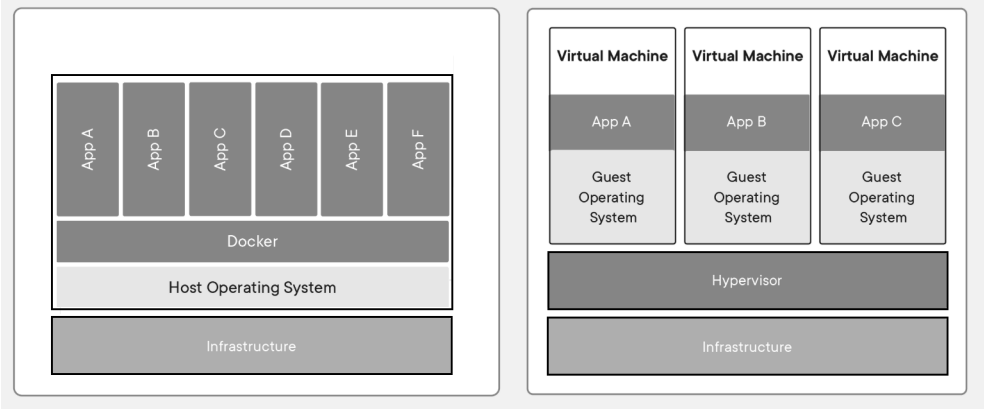

Figure 2.2: Comparison between containers and virtual machines architecture

This difference makes storing or shipping virtual machines more difficult, be-

cause it needs much more space, often taking up tens of gigabytes of storage[14]. Providing a few virtual machines that differ only in the variant of installed application often ends up by providing one virtual machine and then after turning it on the application is being installed by the administrator. On the other hand, containers are lighter[43] because there is no need to copy the whole operating system, necessary binaries or libraries. The size of a container is measured in megabytes[54] rather than in gigabytes.

One of the reasons why containers are called lightweight is that their performance compared to VM is much better. This is because the containers do much less query translations, because there are less layers between application and the actual resources.

Also Docker images[15] have a thoughtful layer system, and creating images that differ only in installed applications, requires less space, only to store the last layer, that includes application files. The rest of images are the same, because Docker images share parent layers, which saves the space and the network usage during downloading of images to the host. Shipping or developing smaller units is more convenient and that is the reason why Docker became so popular and nowadays it is everywhere.

[14]Data dive: VM sizes in the real world https://tintri.com/blog/data-dive-vm-sizes-real-world/

[15]Docker Image Manifest Version 2, Schema 2 format is the main format used by Docker to define images but to provide a dependable open specification that can be shared between different tools and provide compatibility the OCI format was created. It is a specification for container images based on the Docker Image Manifest Version 2, Schema 2 format. Container Registries support pushing and pulling OCI images, therefore the OCI format is simply synonymous with Docker image. https://github.com/opencontainers/image-spec

	Clair	Anchore	Trivy
Vendor	Red Hat	Anchore	Aquasec
REST Api	Yes	Yes	No
Open-source	Yes	Yes	Yes
Scan OS Packages	Yes	Yes	Yes
Scan application dependencies	No	Yes 4 languages	Yes 5 languages
Easy to use	Hard	Medium	Easy

Table 2.4: Comparison of tools for scanning container images.

2.4 Scanning Docker Images

The community of Docker users is growing really fast. But only some of them are noticing the problem of lack of tools to scan images in order to detect the vulnerabilities or exposures. The first articles that can be found in the World Wide Web concerning matter comes from 2017[23][46][45]. From 2017 till 2020 the community was able to develop some open source solutions, moreover companies saw the opportunity to fill this gap in the market. Even Docker noticed that users insist to provide some solution to increase security level. In mid-2020, during work on this thesis, they announced a cooperation with Snyk company and promissed to develop solution that will solve this problem[4].

Open source projects that can be useful to scan Docker images are: *clair*[8], *trivy*[50] or *anchore*[1]. All of these projects are on the same popularity level, but only *clair* and *anchore* provide the REST API. This is really important requirement, because a concept of a system and its future evaluation requires a way to provide data on vulnerabilities to the blockchain, and REST API is the most suitable way to communicate with a tool. There were attempts to provide REST API to *trivy*

but these are just first steps and the repository looks abandon. Both *clair* and *anchore* are still in development what makes them appropriate to use in the system, because the data sources are still updated. Table 2.4 presents the most important information about the above-mentioned scanners[16], which influenced the choice of tools to be used in further stages of work.

2.5 Summary

This chapter provided the most relevant information on the process of selecting the appropriate research technologies to answer the questions posed in Chapter 1. Hyperledger Fabric was chosen as the main system for creating and using the blockchain network. The key technological and conceptual aspects of this tool are presented in section 2.2. Further elements of this master's thesis are based on the presented information in this section.

The next chapter is intended to present related works, that either present past solutions to similar problems or provide information that can be used as assumptions for the conducted research.

[16]The comparison was made on the basis of information contained in the references[8][50][1] and articles. https://boxboat.com/2020/04/24/image-scanning-tech-compared/ https://www.a10o.net/devsecops/docker-image-security-static-analysis-tool-comparison-anchore-engine-vs-clair-vs-trivy/

CHAPTER 3

RELATED WORK

This chapter is dedicated to dissertation about related works in the area of cloud native applications vulnerabilities and detection of them (Section 3.1). Section 3.2 presents general concepts of registry used in computer science. The last section(3.3) shows, how blockchain was used for purpose of creating distributed registry and what were the conclusions and results of past research.

3.1 Cloud Native Applications Vulnerabilities

The first area of related work includes recent studies on the method of how to define and measure the outdatness of cloud native applications based on containerisation technology. This research is also intended to represent the current level of security and vulnerabilities that can affect the work of application packaged into containers.

Zerouali et al. [53] conducted a research on the technical lag[21] in public Docker containers that are build from the Linux-based Debian distribution. They studied 7,380 images with respect to security vulnerabilities and how much outdated the image was. Their research shows that all the containers have high severity vulnerabilities. They have also formed a statement that security management

tools should improve current state by providing data about security vulnerabilities, bugs and a level of technical lag.

Shu et al. [46] conducted a research on the state of security vulnerabilities in both: community and official Docker Hub repositories. They created a scalable framework, called the Docker Image Vulnerability Analysis (DIVA), which was able to discover automatically, download, and analyse Docker images for security vulnerabilities. They studied 356,218 images and observed that both: community and official repositories, contain, on average, more than 180 vulnerabilities. The results of their research show that the security level of cloud native applications based on containerisation may not be sufficient, and this area deserves further research.

Zerouali et al. [52] proposed a tool called ConPan which is able to analyse packages in software containers by reporting how outdated and vulnerable they are. Their work's motivation was to support DevOps and other people wanting to find the vulnerabilities or bugs in their containers. These people were forced to perform tedious task of writing error-prone ad-hoc scripts. ConPan, like two previous solutions mention before, is a tool that scans Docker images but at the same time creating no impact on other users of ConPan. Even though someone is currently scanning container, the other users of ConPan that scanned the same image in the past would not know that his container is not secure anymore.

Berkovich et al. [5] created a benchmark for Container Image Scanning. They have tested all three tools mentioned in the Section 2.4 and the results show that there is no one tool good for all the purposes, therefore system should be adjusted to use different sources of susceptibility.

3.2 The Concept of Registries in Computer Science

The second area of related work conducts the general concept of registries in computer science. The concept of a centralise storage for keeping informations is quite old. The beginnings of service registration dates back to the ARPANET's era. The basic idea was to keep more memorable name instead of a host's numerical address. The Stanford Research Institute was using a text file named HOSTS.TXT to map host names to the numerical addresses of computers on the ARPANET[27].

Later this idea became the basis for creating DNS. This decentralised naming system for computers, services, or other resources in the Internet is so crucial in today's world that turning it off even for a few minutes would led to a global crisis. There has been proposed a large number of architectures for this approach and the most known one is Service-oriented architecture (SOA)[16]. This design assumes the existence of three components: service provider, service broker (service registry or service repository) and service consumer. In the context of this work, the most important element is service broker, also called service registry. It is responsible for keeping the information and passing it on to requesters.

Michlmayr et al. [29] presented that the use of SOA was often reduced at that time to only two elements out of three (omission of the service registry). They have stated that using only service provider and consumer limits the envisaged potential of SOA implementations considerably. They have also discussed implementation parts of their infrastructure which could be of useful during creating large-scale SOAs.

The registry concept is also used in Peer-to-Peer(P2P) technologies. In the early 2000s, P2P was synonymous with services such as Napster or Gnutella[34]. In this type of networks every node is a client and the server at once. In order for a peer to join the network, it is enough for it to know the address of one of its users, and

then, in the process of information exchange, it gets to know others. Therefore, peer of this network was not only a client or server, but also acted as a dynamic service registry with information about other peers.

Nevertheless, the one of the most important factors that differs architecture of service registries is the degree of centralisation of the registry, where the information about services is stored[38]. Rambold et al. [38] presented a survey and comparison of service discovery approaches. Their research showed that industry focuses mainly on semantic service description and reasoning techniques as well as on scalability and robustness.

3.3 Blockchain-based Registries

The third area of related work includes research on using blockchain-based network to create distributed registries as a service broker.

P. de Lange et al. [26] conducted a research and presented a service registry and discovery solution for a decentralised community information system with blockchain technology use. As one of the arguments to study this area they have ascertained that decentralised infrastructures today do not offer a simple way for users to discover available services in the network, nor the ability to securely verify their origin and history.

Authors of this paper defined an underlying purpose of creating blockchain-based service registry as a need of creating solution for microservice infrastructure [13] created by them for distributed network. This network needed solution to enable users to find and verify microservices published by others. One of the problems that they had to face was the fact that there were no authorities, who could coordinate or serve as a universally trusted entity in the system.

This master thesis will consider two different approaches, one with authorities

responsible for publishing information about vulnerabilities of cloud native applications, and the other where each user will be able to publish it. The blockchain will be used to provide different trust levels, which will allow to react differently to the vulnerabilities reported by different peers.

As a conclusion of [26] what authors stated: that advantages of decentralisation are empowering communities to take control over their infrastructure. They are also confident that blockchain-based decentralised service registries can provide a valuable addition to nowadays solutions.

3.4 Summary

This chapter presented the current state of knowledge on vulnerability detection in cloud native applications, which has a significant impact on how the solution concept should be created. Then, the issue of registers and its impact on the appearance of current software development patterns was presented. It is quite general, but crucial concept that allows to create distributed solutions. The last section, on the other hand, contains information on the work related to the use of blockchain networks to create distributed registry with the ability to save and read data on available services.

The next chapter is intended to present general concept and the model of a *bbdvir* system, which task is to meet the requirements set out in Section 1.1 to be able to provide results on the research conducted in the areas presented in this chapter.

CHAPTER 4

BBDIVR CONCEPT AND MODEL

This chapter describes main aspects, general concepts and model that were designed to meet all the requirements of a system presented in Section 1.1, to be able to store and operate on data about cloud native applications' vulnerabilities. Section 4.1 is dedicated to discussion about major blockchain network and system assumptions, which had a significant impact on the solution. The next section(4.2) shows the structure of the data and the available business logic concluded in smart contracts. The main purpose of section 4.3 is to present system interaction with the chaincode. The last section(4.4) of this chapter is to present how *bbdivr* is combined with external services and the idea of detecting vulnerabilities in cloud native applications.

4.1 *BBDIVR* Concept

The concept of the system draws from the objectives of Common Vulnerabilities and Exposures(CVE)[11] or National Vulnerability Database(NVB)[33] registries. The system will provide availability to exists one or more organisations that peers will be considered as trusted units. The same concept is used in CVE and it is called CVE Numbering Authorities(CNAs). For the purpose of this thesis they will

be called ‘authority users‘ short AU . Other type of peer would be a unit that will be able to read information about vulnerabilities. For the purpose of this thesis they will be called ‘standard users‘, short SU. This type of division will deliver better policy management and certainty that vulnerabilities described in the ledger are correct and reliable.

The assumptions during the conducted research were made in such way, to enable the reconfiguration of the network or its modifications in the future.

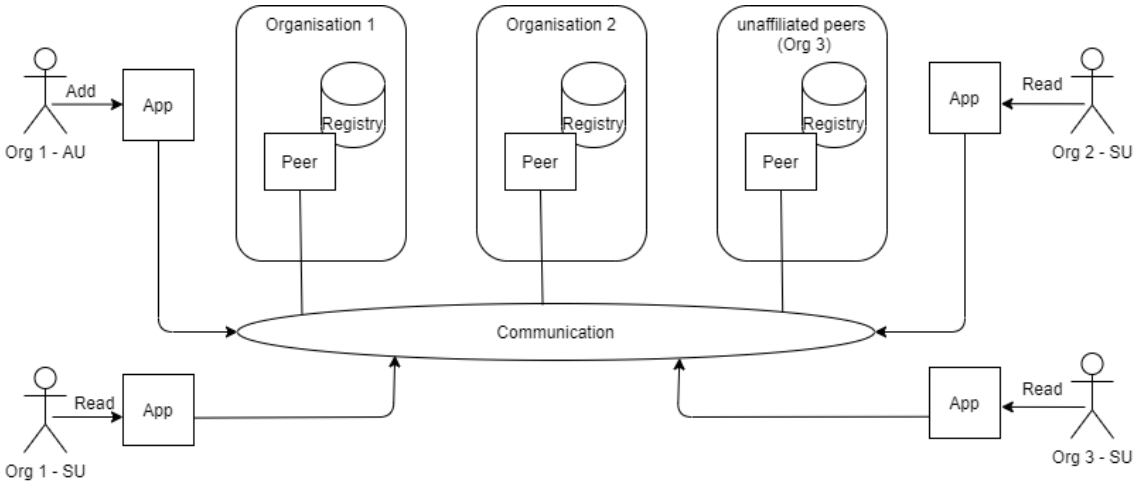

Figure 4.1: General concept presenting the appearance of the created blockchain network.

Figure 4.1 shows a general view of the system architecture to bring closer look at the concept. The presented system assumes that there can be any number of organisations in it. However, in order to enable the use of non-affiliated users, an organisation associating them will be created. For the purposes of the considerations, it can be assumed that a non-associated user will be able to obtain his certificate by registering in some Web application. In each organisation, the user will be able to belong to the SU or AU. Thanks to this, the system will have $2 * N$ (where N is the number of organisations) different levels of trust for the reported

vulnerabilities[1].

Information about vulnerabilities and associated Docker image layers found in cloud native applications, thanks to the use of blockchain networks, will be distributed among the peers of the organisation. This approach ensures that the use of various vulnerability search tools will create a distributed registry, that will contain all the information. A situation can be imagined where two different users scan the same image with two different tools. Information on vulnerabilities from both ends up in the blockchain. Now, querying the network using an image identifier returns the sum of these two scans. A more ambitious scenario can be assumed, in which after scanning the image, the system will locally remember its identifier and if someone in the future will add new vulnerabilities to blockchain connected with this image, the user will be informed about it.

Also AU users will be allowed to add vulnerabilities manually. This feature is important, because trusting only a scanning tool is not enough and some threats can be difficult to detect by tools.

4.2 Smart Contracts

This section and Section 4.3 can be considered as the most important in this chapter, because it describes the main aspect of this master thesis - creating a registry logic to persist information that can be discovered by other participants of the network. The registry in this case is the distributed ledger, also called the world state, which will contain two types of business entities.

The first type of object is a vulnerability with an ID as a key and a description, link, severity etc. as a state. The second type of object is a Docker image layer

[1]In Hyperledger Fabric, each transaction has an assigned sender's certificate, and on its basis, its organisation and type can be determined.

with layer's ID[2] as a key. Since information about parent layer's ID, creation date, modification date etc, are stored as state in the ledger, the most important is a list of vulnerabilities' IDs and the list of non-inherited vulnerabilities from the parent layers.

The first idea was to store only information about vulnerabilities in current layer and the sum of all vulnerabilities will be calculated by creating aggregation of vulnerabilities from parents until reaching layer without a parent. This concept was near to the ideal solution, because it was simple, it allowed to get all vulnerabilities and did not introduce data redundancy, but it did not considered removing vulnerabilities in higher layers. Simple example presenting this problem: layer A contains vulnerability A, then layer B based on layer A contains installed improvement that resolves problem with vulnerability A.

The second idea was to create a list of current vulnerabilities and the list of inherited vulnerabilities from the parents layers. This solution was correct but, it created a problem. When some low layers contain a lot of vulnerabilities, and then there is a stack of many layers without any vulnerabilities, then each higher layer will contain a list of the identifiers of vulnerabilities. This approach created a problem with memory and write time, because higher layers had a lot of vulnerabilities to persist - data redundancy. However, this solution had the advantage that reading the vulnerability about a given layer did not require aggregation of the vulnerabilities from the lower layers, and thus is was faster.

To solve the problem with redundancy the final concept was proposed. It consider storing information about non-inherited vulnerabilities from parents. It implies a slowdown in read operations, because smart contract has to gather all the vulnerabilities from parents' layers and exclude not inherited ones. A graphic

[2]Docker image layer identifier is calculated by applying the algorithm of SHA256 to a layer's JSON file content, it provides equality of image layer's IDs built on two different computers

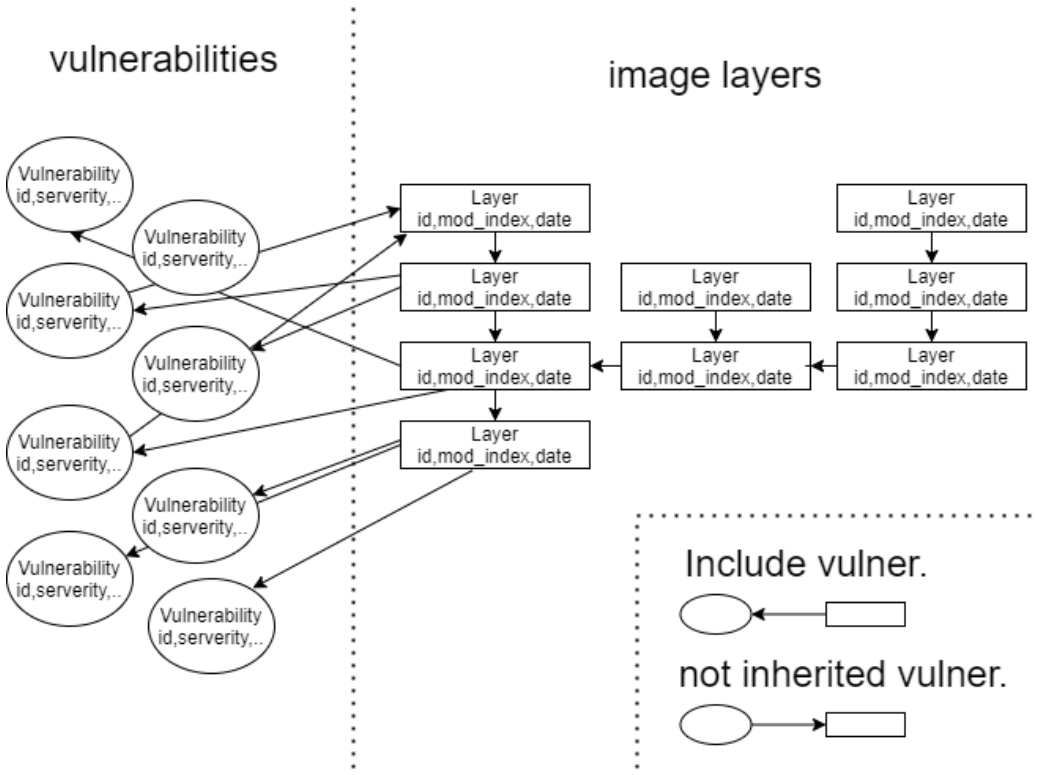

Figure 4.2: Graphical representation of chaincode model.

representation of this approach is presented in Figure 4.2. However the write operations are faster and the gain outperforms the reading loss. The ledger is distributed on all peer nodes in the network, so read operations are executed on the local world state, and the write operations have to be executed on all peers that are chosen by chaincode policy. Then after transaction is validated and approved it has to be transmitted to all peers in the network, therefore too large objects would be hard to handle.

Both smart contracts: vulnerability and layer contract, will be packaged into single chainecode. It will provide a consistency of business logic, because it should not be possible for existence of an image layer in the system that contains a vulnerability which does not exist in the world state.

4.3 Interaction with Chaincode

The proposed chaincode (in Section 4.2) will provide business logic to the created system. To be able to fully evaluate the correctness of the solution there is a need to develop some sort of a application that will interact with the chaincode to test it. The second reason of implementation of this application is a research on the capabilities of Hyperledger Fabric to be used as a system, which will provide not only data distribution, but also the possibility of grouping data according to the level of trust.

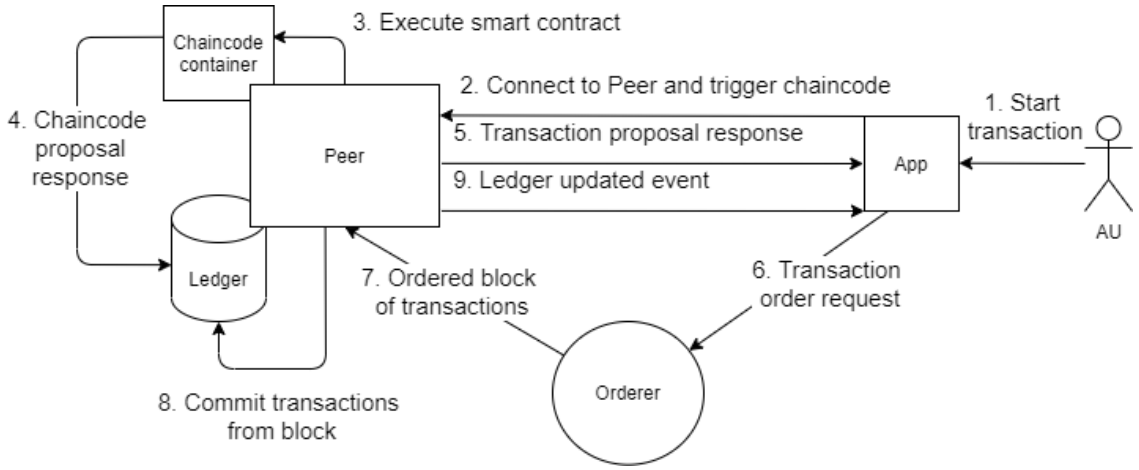

Figure 4.3: Underlying processes during interacting with *bbdivr* blockchain network.

The concept of application assumes interaction with the ledger. The first thing that has to be done is to confirm identity and register a user in the ledger. To persist information about it, a file system wallet shall be used. There is no need to use database wallet, because it complicates the setup and from the documentation is the rarest form of wallet storage. The third option is to use in-memory wallet, but all the identities will get lost after the crash or regular shutdown of application.

After user's enrolment, the application is ready to interact with the chaincode.

Figure 4.3 introduce the different steps in the process of interacting with chaincode after successful enrolment. The life cycle of a transaction containing data is the same as shown in Section 2.2.2, however this figure shows the successive nodes where the transaction is processed. This is important for the application's communication with chaincode. The application has to send two queries and then wait for the asynchronously sent ledger update event.

Taking advantage of the fact presented in the previous section that each transaction is signed by the person submitting it, the system can easily provide a mechanism for defining priorities and trust. It will be based on getting the signer certificate, and then check its correctness and extract from it information about the organisation to which it belongs.

4.4 Integration with External Services

The last important matter that will be included in this chapter refers to the detection of vulnerabilities in cloud native application and *bbdivr* integration with external services. In Section 2.4 *clair* and *anchore* were described as the most suitable, open source tools, that are able to scan Docker images. Both of them provide REST API, so the proposed solution is to create universal interface, that will be able to use this tools and allow in the future integration with other scanning services, that will provide more information about vulnerabilities or exposures. Thanks to this approach, bbdivr will have a diverse set of data and the ease and speed of adding new vulnerabilities to the system will allow for its better evaluation in terms of the size of the transactions being processed.

Figure 4.4 shows the system and used external services. Clair and anchore will be responsible for scanning images, while the user will also be able to add a

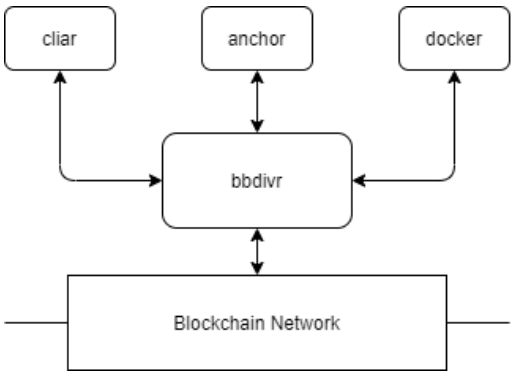

Figure 4.4: External systems that are interacting with *bbdivr*.

vulnerability that he finds or believes may exist via the command line interface. An important element is also communication with the Docker engine in order to retrieve information about the image, for example a list of layers, date of creation or identifiers.

There should also be an environment to run external tools regardless of the operating system. To achieve this goal the Docker container, in which these tools will be installed, will be created. It could be the only way to used them on Windows operating system, because *clair* and *anchore* are created with a view to use them on Linux operating systems.

4.5 Summary

This chapter presents the most important aspects of the implemented system to meet the presented requirements. This makes it possible to evaluate this solution and conduct research on its usage in the considered problem.

The next chapter presents the more detailed architecture of the proposed problem solution and the important implementation aspects that had a big influence on subsequent evaluations.

CHAPTER 5

BBDIVR DESIGN AND IMPLEMENTATION

This chapter is dedicated to discuss and present *bbdivr* design and important implementations parts. Section 5.1 presents system architecture, design and the main points of a cut between the system components. The next section (5.2) concerns the core component of the system - the *bbdivr* engine. Section (5.3) shows the implementation of the chaincode. Section 5.4 is dedicated to the implementation of clients, that would interact with APIs, which will provide informations about vulnerabilities and exposures. The last section (5.5) presents command line interface that was created to interact with the system to provide better users experience and encourage others to use and test this solution.

5.1 Project Overview and Architecture

Several approaches were considered when researching the issue of using blockchain networks to create a registry to detect vulnerabilities in cloud native applications. The final implementation and system architecture have been designed to meet all good practices and reflect the realities of enterprise solutions as much as possible.

5.1.1 Project Overview

The main programming language used for implementation is Java (version 11), both system implementation and smart contract have been written in it. This decision was made because of the popularity[1] and enterprise nature of this language. Additionally, few articles mention Java as a related language in blockchain technology[36][2], so one of the results of this master's thesis will be an evaluation of Java's suitability for this problem.

The concept of the application assumes usage of enterprise-level Java framework and the most suitable option is to choose Spring[48] framework. It provides a vast range of ready components and libraries that are used during professional development. Using this framework to interact with the chaincode will provide information for this thesis about Hyperledger Fabric suitability for teams of JVM based languages' developers.

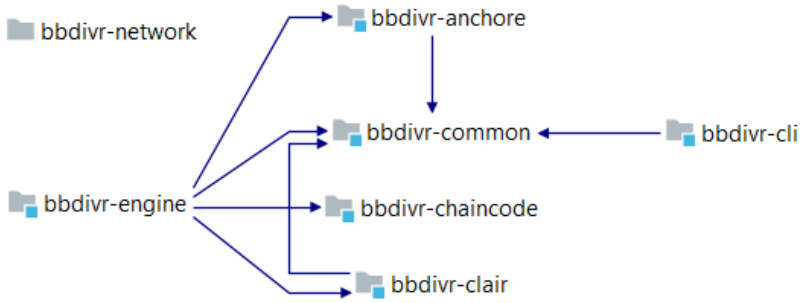

Figure 5.1: Dependencies between *bbdivr* system modules.

The management and comprehension tool, which was chosen for this project,

[1]According to the most citable indicator of the popularity of programming language - TIOBE Index, Java is in top 2 used languages in the World (July 2020). https://www.tiobe.com/tiobe-index/

[2]Porru et al. [36] analysed and stated that number of Java repositories concerning Blockchain-oriented Software is about 3.5% of the total.

is Apache Maven[3]. The solution also contains Bash scripts to bring up the infrastructure and Jupiter Notebook files to evaluate system and visualise results. Figure 5.1 presents project modules, dependencies between them and it is made up of the following parts:

- **bbdivr-network** - this module contains infrastructure that is useful during development and the evaluation process. It was created to start all the important processes with just one command.
- **bbdivr-engine** - this module is responsible for providing REST API that exposes system's functionality. The second also crucial responsibility is the interaction with Hyperledger Fabric and the chaincode (further details in Section 5.2).
- **bbdivr-chaincode** - this module includes the source code of two smart contracts which are the parts of this chaincode (further details in Section 5.3)
- **bbdivr-anchore** - this module is an implementation of the client for Anchore Engine API (further details in Section 5.4.2).
- **bbdivr-clair** - this module is an implementation of the client for Clair API (further details in Section 5.4.1)
- **bbdivr-cli** - this module is a implementation of command line tool that provides easy way to use the system (further details in Section 5.5).
- **bbdivr-common** - this module contains the common model classes that are shared between other modules. This approach creates a kind of contract between the interfaces provided by the rest of the modules. Thanks to this,

[3]Apache Maven https://maven.apache.org/

it was not necessary to use the Consumer Driven Contracts[39] approach like Spring Cloud Contract[4].

5.1.2 System Architecture

The previous section presented the project modules that are responsible for providing components of the system. The system architecture consists of six internal components and uses four external interfaces.

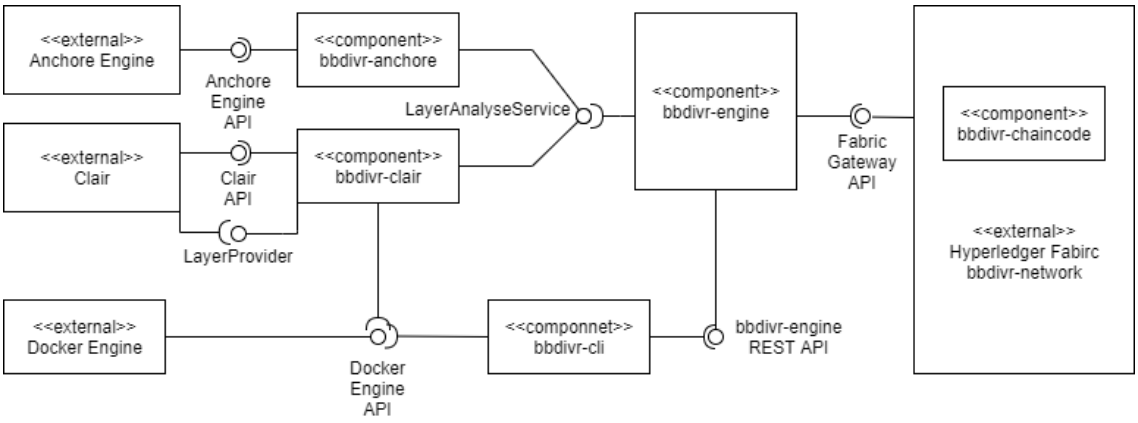

Figure 5.2: *BBDIVR* architecture detailing individual modules and external systems.

Figure 5.2 shows how the individual components interact with each other and where are the main points of interaction with other parts and external services. The *bbdivr-engine* component is the heart of the whole system. It provides a REST API interface that allows to use all the functions. This API is used by the *bbdivr-cli*. The *bbdivr-engine* is using Hyperledger Farbic Gateway API to interact with the created network.

On the left side of this figure, the engine requires providing implementation of *LayerAnalyseService* interface. This implementation is delivered by two modules

[4]Spring Cloud Contract https://spring.io/projects/spring-cloud-contract

and more can be added in the future. The *bbdivr-anchore* component is one of them, and it is using Anchore Engine API provided by external system running on Docker container.

The second of mentioned modules, implementing *LayerAnalyseService*, is the *bbdivr-clair* component. This module on the contrary to previous one needs an access to Docker Engine API because it has to provide access to image's layers through REST API which is marked in the diagram as *LayerProvider*. This interface is consumed by external system - Clair, which is also running on Docker container.

The module *bbdivr-chaincode* is packaged and deployed in the Hyperledger Fabric network. The following sections will provide better understanding and look on presented modules.

5.2 *BBDIVR* Engine

The *bbdivr-engine* is a server side application to interact with chaincode deployed in the Hyperledger Farbic blockchain. This module is based on Spring Boot[5] that allows to create stand-alone, production-grade Spring[6] based Applications. This framework is responsible for creating REST API and dependency injection.

The packages in this module were created to isolate the controller, service and communication layer from the blockchain-concerned classes. Figure 5.3 presents a simplified class diagram, which presents the most important classes that are responsible for adding a Docker image layer to distributed ledger. The diagram does not include auxiliary classes, elements responsible for blockchain querying

[5]Spring Boot provides many modules with their default configurations and embedded application servers. https://spring.io/projects/spring-boot

[6]Spring Framework provides a comprehensive programming and configuration model for modern Java application, it is included and configured in Spring Boot https://spring.io/projects/spring-framework

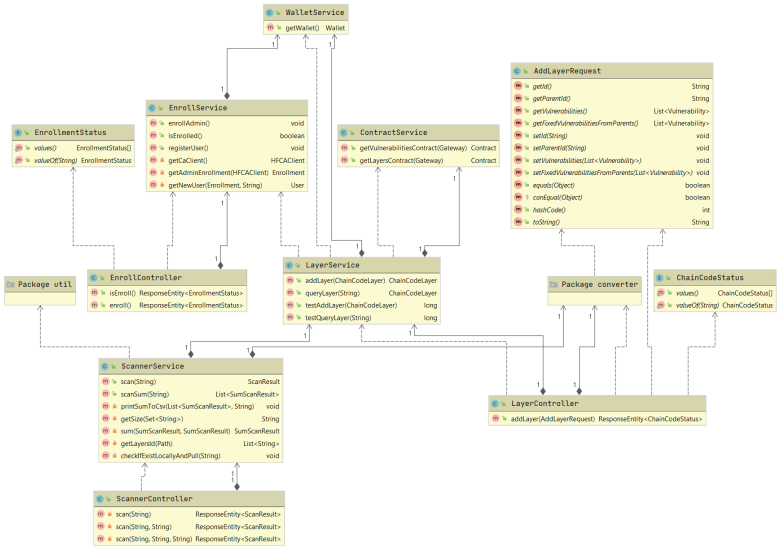

Figure 5.3: The simplified class diagram (not all included) of the *bbdivr-engine* module, which is responsible for the business logic of adding a new layer with related dependencies do blockchain network.

and for adding individual vulnerabilities, because they are similar to those in this diagram.

The most important element of this module is responsible for interaction with the chaincode. To achieve that Hyperledger Fabric Gateway SDK for Java was included. The chaincode-interaction package was extracted from the others to encapsulate the blockchain logic. Most of the configuration is stored in the static constants fields and files due to the conceptual character of the project. The time and work devoted to this paper were mainly spent to research and the creation of prototypes.

```
1  public ChainCodeLayer addLayer(ChainCodeLayer layer)
2          throws IOException, InterruptedException, TimeoutException,
           ↪  ContractException {
3
4      builder.identity(walletService.getWallet(),
       ↪  APP_USER).networkConfig(networkConfigPath).discovery(true);
5      try (Gateway gateway = builder.connect()) {
6
7          byte[] result = contractService.getLayersContract(gateway)
8                  .createTransaction(ChainCodeOperations.ADD_LAYER
9                  .getOperationName())
10                 .submit(layer.getId(),
11                         layer.getParentId(),
12                         layer.toJson());
13
14         return ChainCodeLayer.fromJson(new String(result,
           ↪  StandardCharsets.UTF_8));
15     } catch (Exception e) {
16         log.error("Error evaluating the contract", e);
17         throw e;
18     }
19 }
```

Listing 1: Part of the code from *bbdivr-engine* presenting the implementation of *addLayer* function from *bbdivr-layer-contract*.

Listing 1 presents a part of Java code responsible for calling function from *bbdivr-layer-contract*. First line of function's body sets the identity which is used to connect to the network. The builder is a class field of type *Gateway.Builder*. All operations under this gateway connection will be performed using this identity.

Then the connection is open using Java a try-with-resources statement[7]. Then we have the most important part, first function from this method call chain is providing the right contract(*getLayersContract*), then on the contract, transaction is crated with name of the function from contract. Then we have *submit* method. From the documentation, this method submits a transaction to the ledger. The transaction function represented by this object will be evaluated on the endorsing peers and then submitted to the ordering service for committing to the ledger. The arguments are the ones defined in the contract.

The *addLayer* method is a type of function, that has to be committed, because it changes the state of the ledger. For functions that only queries the world state we can use *evaluate* method instead of *submit*. The contract function is evaluated on the endorsing peers but the responses will not be sent to the ordering service. The return statement, in discussed code snippet, is converting byte array, which is result of *submit* method, to string and then to *ChainCodeLayer* object. The creators of Hyperledger Fabric decided that result of all functions calls (submit or evaluate) will return byte array because this gives the freedom of choice for developers of what type will be returned.

[7] Java introduced *try-with-resources* statement in version 7 to close resources automatically and regardless of exceptions after try code-block ends

5.3 *BBDIVR* Chaincode

The *bbdivr-chaincode* module contains implementation of two contracts: *bbdivr-vulnerability-contract* and *bbdivr-layer-contract*. This two contracts share some model classes, and a detailed diagram showing the types of objects and the dependencies between them is in Figure 5.4.

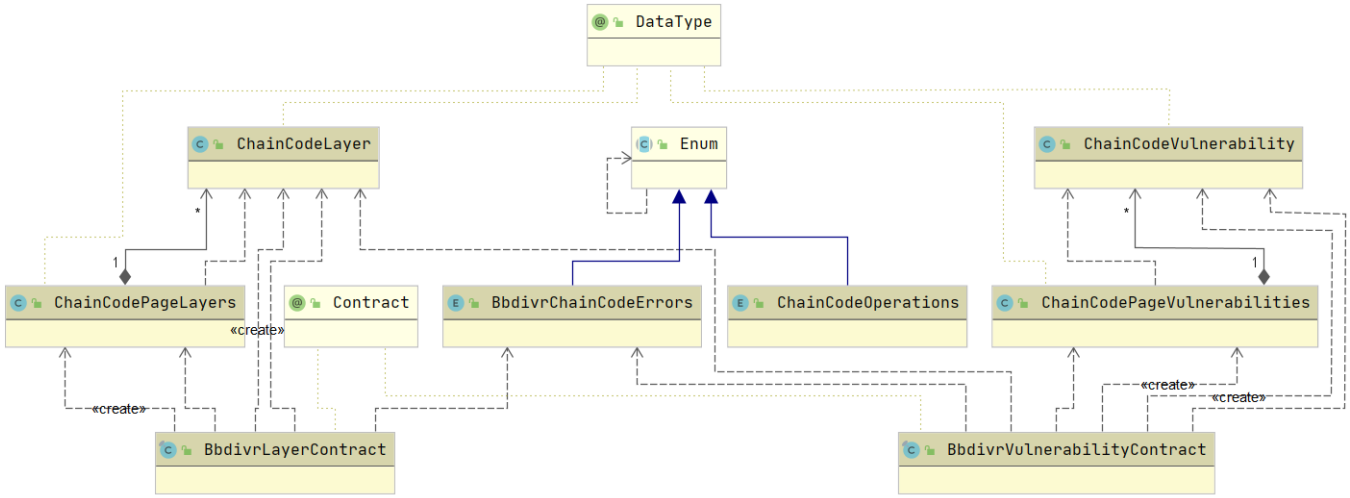

Figure 5.4: The dependencies between different types of objects in the *bbdivr-chaincode* module.

The first smart contract - *bbdivr-vulnerability-contract* - is responsible for the model of the items stored in the world state and business logic available for the peer's applications. The conducted research has shown that most of information about vulnerabilities has a similar set of characteristics. The proposed model is the combination of these features. The table 5.1 presents explanation to the model fields.

Model Property	**Property Description**
id	The unique identifier of the vulnerabilities in the ledger. To provide easy understanding, better portability and prevent duplication of entities describing the same vulnerability but under different ids, the best choice were the code-name of vulnerabilities (e.g. CVE-2020-24584)
featureName	The name of the feature affected by the vulnerability.
featureVersion	The version of the feature affected by the vulnerability.
featureVersionFormat	The version format is the format used by installer package manager to store and package versions. (e.g. dpkg[8])
namespaceName	The name of the namespace (e.g. debian:8)
description	This is complete description of the vulnerability including information like: how to use the exposure or more detailed information on how to fix it.
link	The link to the first information where the vulnerability was described.
severity	The level of vulnerability (available levels: defcon, critical, high, medium, low, negligible, unknown)
fixedBy	The solution to fix the exposure (if exists)

Table 5.1: Description of vulnerability contract model.

[8]dpkg - it is a package management system in the Debian operating system and its numerous derivatives. https://en.wikipedia.org/wiki/Dpkg

The *bbdivr-vulnerability-contract* provides functionality for adding vulnerability, getting vulnerability by the identifier and querying all vulnerabilities in pages. Using paging to collect them is necessary, because the number of all vulnerabilities is to large[9] to be accessed at once.

```
1 @Transaction()
2 public ChainCodeLayer addVulnerability(final Context ctx, final String
  ↪  vulnerabilityId, final String vulnerabilityAsJson) {
3     ChaincodeStub stub = ctx.getStub();
4
5     getVulnerability(ctx, vulnerabilityId, false);//check if vulnerability
      ↪  already exists
6     stub.putStringState(vulnerabilityId, vulnerabilityAsJson);
7
8     return jsonConverter.fromJson(vulnerabilityAsJson, ChainCodeLayer.class);
9 }
```

Listing 2: Code snippet of *addVulnerability* function from *bbdivr-vulnerability-contract* responsible for adding new vulnerability to world state.

The listing 2 presents the most interesting and the main Java code of *bbdivr-vulnerability-contract* responsible for adding new vulnerability to world state. All functions implemented in the contract must be annotated with *@Transaction*, what allows Hyperledger detect, which functions should be available to execute in the blockchain and which not. The first argument of all annotated methods is *Context* and all the remaining arguments are of *String* type. The *Context* is responsible for access to the APIs for the world state. Presented function in the code snippet use context to get *ChaincodeStub*, that gives access to modify the distributed ledger or

[9]This function is responsible for retrieving all vulnerability information from the ledger regardless of the layer in order to create statistics or studies them. To get all layer-related vulnerabilities, use the *bbdivr-layer-contract*, which returns a layer with all related vulnerabilities.

```
@Transaction()
public ChainCodePageVulnerabilities queryPagedVulnerabilities(final Context ctx,
    final String pageId, final String pageSize) {
    ChaincodeStub stub = ctx.getStub();
    final int pageSizeInt = Integer.parseInt(pageSize) + 1; //increased to get
        next bookmark
    List<ChainCodeVulnerability> vulnerabilities = new
        LinkedList<ChainCodeVulnerability>();
    QueryResultsIteratorWithMetadata<KeyValue> results =
        stub.getStateByRangeWithPagination("", "", pageSizeInt, pageId);
    for (KeyValue r : results) {
        vulnerabilities.add(jsonConverter.fromJson(r.getStringValue(),
            ChainCodeVulnerability.class));
    }
    ...
}
```

Listing 3: Code snippet of *queryPagedVulnerabilities* function from *bbdivr-vulnerability-contract* responsible for querying paged data.

querying it. The *addVulnerability* method before adding vulnerability to the world state is using function *getVulnerability*, that checks if the identifier already exist and throws exception in that case. Then the *putStringState* is called, which writes the specified value and key into the ledger.

Listing 3 presents the next method from *bbdivr-vulnerability-contract* called *queryPagedVulnerabilities*. *It* uses *getStateByRangeWithPagination* to get objects from world state with pagination. To get the first page, the argument *pageId* should be an empty string. In the code can be noticed *pageSizeInt*, which is increased by one to get the identifier of the vulnerability at the next page, which is used as the bookmark in the future function calls. As the *startKey* and *endKey* the

empty string is passed to query all vulnerabilities in the ledger, from which page will be calculated.

The second contract included in the chaincode is *bbdivr-layer-contract*. It is responsible for providing the model, which persist information about Docker image layers. Using layers instead of whole image in this model allows to link layers into the chain in the order in which layer has been built. Also business logic, implemented in this contract, requires existence of parent layer in case when the upper layer is added to the ledger. Thanks to this fact, more space is gained, what is more, the addition of new images built on already existing layer requires less interaction with the network than it used to. The table 5.2 presents explanation to the model fields.

Model Property	**Property Description**
id	The identifier of the layer that is the SHA256 number which is calculated by applying this algorithm to the layer's content. It provides equality of image layer's ids build on two different computers.
parentId	This is the identifier of the parent layer.
createDate	The date when the layer was added to the world state.
modifyDate	The date when the layer was modified in the ledger by updating the vulnerabilities connected to the layer.
modifyIndex	The index, it is a integer number incremented after every modification of this layer in the ledger.

vulnerabilityIds	The list of vulnerabilities identifiers detected in the layer.
fixedVulnerabilityIds	The list of vulnerabilities identifiers fixed in the current layer and detected in parent layers.

Table 5.2: Description of layer contract model.

The *bbdivr-layer-contract* provides functionality for adding new layers, getting layers with vulnerabilities by the identifier, adding new vulnerabilities to existing layer and querying all layers in pages. Using paging to retire them is necessary because the number of all layers[10] and the size of each are too large to get them at once.

The contract code is similar to the one shown above, hence it is not presented in this thesis but can be easily accessed in the GitHub repository.

In the contract the functionality of querying chain of layers from the top layer to the lowest layer was also implemented. It provides an overview of all exposure in the Docker image. The last function gives the ability to add new vulnerabilities to already existing layer in the ledger. It is really important, because vulnerabilities can be found by the bug hunters or researches after the image was scanned and added.

Modification of the layer increases the index of the modification and changes its date of the modification. It is crucial, because scanning of images should be

[10]This function is responsible for retrieving all layers from the ledger regardless of the image in order to create statistics or studies them. To get all layers for specific image, *queryLayerWithParents* method from the contract should be used, which queries the ledger with successive image layer IDs.

done frequently, not only once when i the image is build. For companies that use private Docker image registry the suggestion might be to store information about the images deployed in production environment and perform regular scans. Also subscription to the specific layer identifiers could be implemented and defined. This would provide immediate information after the new vulnerabilities are added to the layer stored in the ledger.

5.4 Vulnerabilities Providers

Detecting exposures in the system or in Docker images is not a simple task. The approach presented in Chapter 4 is the most suitable one to provide information about vulnerabilities. In order to use the best practices, the system has been designed to be able to use various sources of information about vulnerabilities. The first used external service was Clair. However, to demonstrate that the system was created in a correct way the second client for Anchore Engine REST API was implemented. The next two sections present the most important aspects of *bbdivr-clair* (Section 5.4.1) and *bbdivr-anchore* (Section 5.4.2).

5.4.1 Clair API Client

The usage of Clair API can be described as difficult to start work with. It requires from the developer to implement the Clair REST API client that will provide information to the Clair about the layer URL (Uniform Resource Locator). However, before this happens, layer needs to be available on the client host. To achieve this goal, interaction with Docker Engine API is needed. In this purpose the Java API client for Docker was added to dependencies.

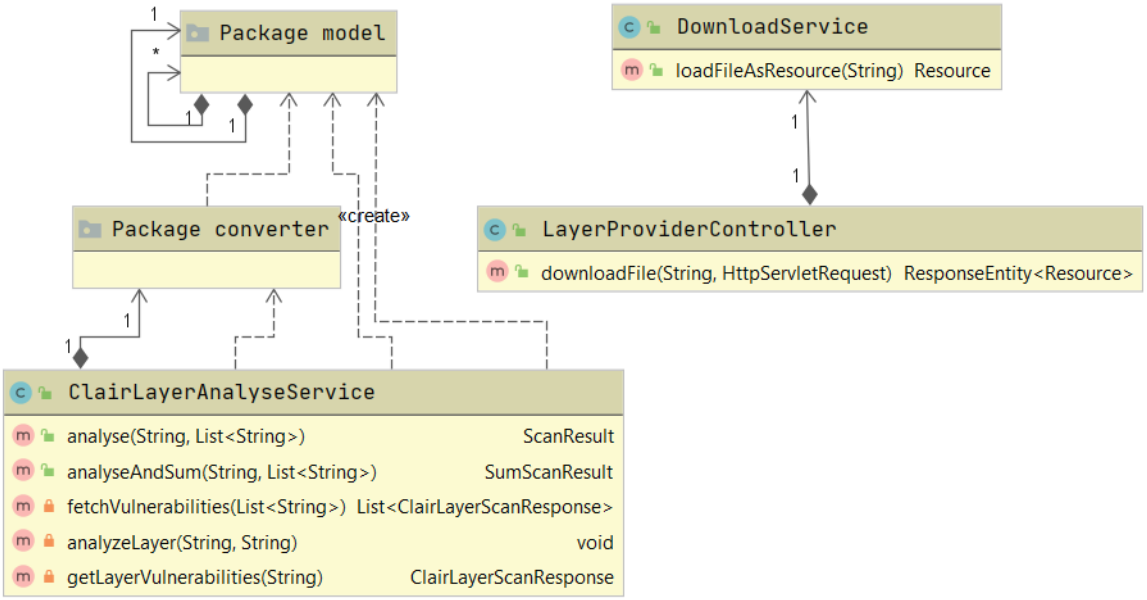

Figure 5.5: The simplified class diagram (not all included) of the *bbdivr-clair* module.

The class diagram (Figure 5.5) was presented in a simplified way without taking into consideration classes responsible for the API model. It contains the abovementioned functionalities, i.e. the Cliar API client and the controller that provides the layers for this tool.

Listing 4 shows the most important implantation elements that allow to establish communication between the application and the Docker Engine API and then call all necessary commands. The first method is responsible for the client configuration for this API.

Next *bbdivr-clair*, needs at first to check if the image exists in the local repository by calling the inspect command. If the image does not exist locally it has to be pulled from the remote repository, in this case Docker Hub. Then the image needs to be saved in the local files' system, the output of this command is an input stream of compressed data to the TAR format. After decompression and

```
1  public DockerClient getClient() {
2      DefaultDockerClientConfig config =
       ↪  DefaultDockerClientConfig.createDefaultConfigBuilder()
3              .withDockerCertPath(System.getProperty("user.home") +
               ↪  "/.docker/certs")
4              .withDockerConfig(System.getProperty("user.home") + "/.docker/")
5              .withDockerTlsVerify(true)
6              .withDockerHost("tcp://localhost:2376").build();
7      DockerCmdExecFactory dockerCmdExecFactory = new OkHttpDockerCmdExecFactory();
8      return DockerClientBuilder.getInstance(config)
9              .withDockerCmdExecFactory(dockerCmdExecFactory).build();
10 }
11 private void checkIfExistLocallyAndPull(String imageName) {
12     try {
13         getClient().inspectImageCmd(imageName).exec();
14     } catch (NotFoundException e) {
15         ...
16         try {
17             getClient().pullImageCmd(imageName).start().awaitCompletion();
18         } catch (InterruptedException interruptedException) {
19             ...
20     }
21 }
22 ...
23 checkIfExistLocallyAndPull(imageName);
24 InputStream inputStream = getClient().saveImageCmd(imageName).exec();
25 ...
```

Listing 4: Elements of *bbdivr-clair* implementation that are responsible for communication with Docker Engine API.

saving input stream as files and directories in the local files' system there is access to directory containing manifest file and list of TAR files for every layer. Parsing manifest file provides information on the unique layer identifiers in the order in which the image has been built.

The next step is to call Clair REST API with information about the name, the URL path to the layer content, name of the parent and the format. As the name and parent's name the layer's identifier were used.

Clair after receiving the URL path is calling it to download the file. To provide it, the download service needs to be implemented and it serves saved layer's files. Clair is supporting two containers formats - Docker and App Container (appc)[3], but this format was deprecated in favour of Open Container Initiative (OCI). The App Container (appc) Image format (ACI) maps more or less directly to the OCI Image Format Specification with some small exceptions. When Clair finishes downloading, it starts scanning the layer and after that the response to the first call is returned. Now the call to Clair API is needed to fetch data about installed features and related vulnerabilities. Figure 5.6 presents UML sequence diagram that will allow to better understand the whole presented process.

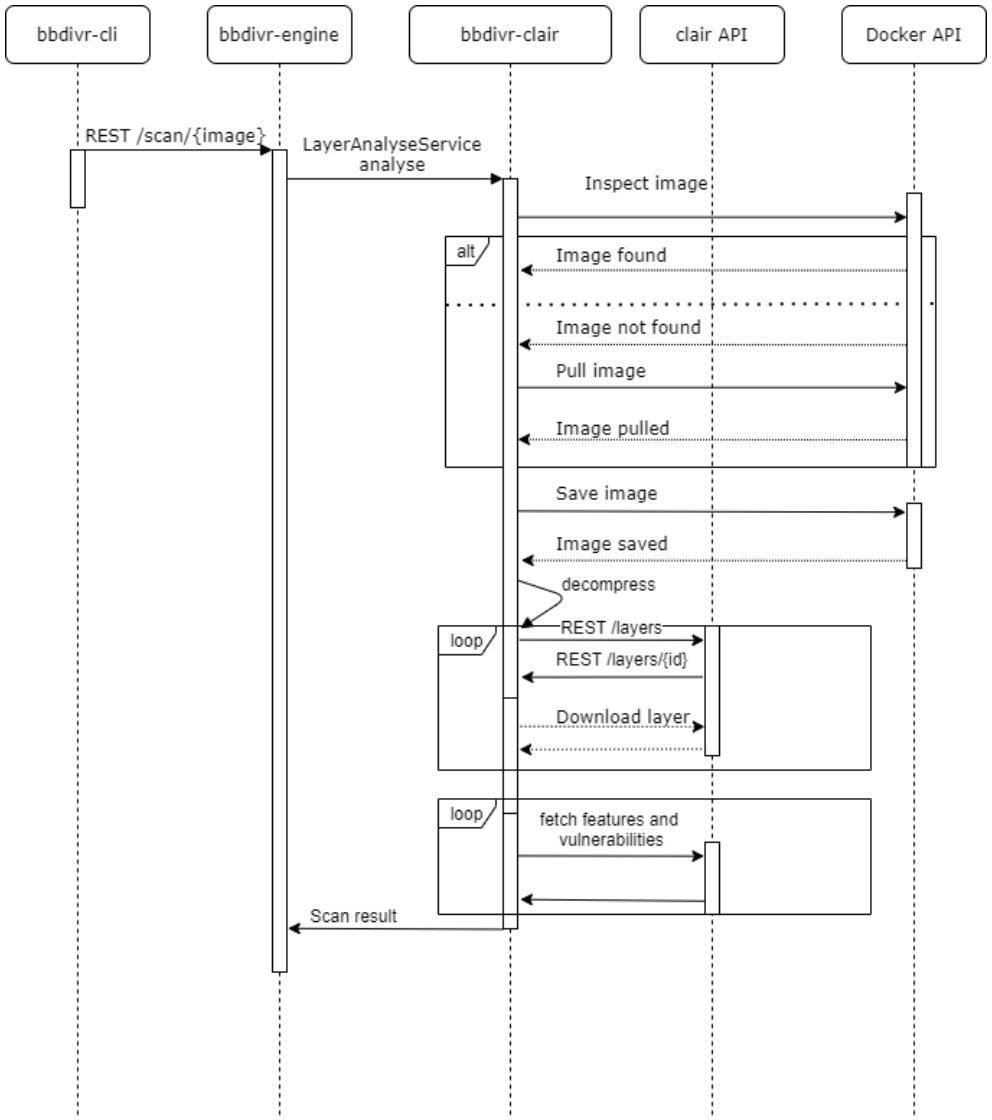

Figure 5.6: Sequence diagram of the *bbdivr-clair* component during image scan.

5.4.2 Anchore Engine API Client

The next implemented client was responsible for communication with Anchore Engine REST API. The name of this module is *bbdivr-anchore*. This tool differs significantly from the one described in Section 5.4.1. The most important difference is that Anchore Engine has implemented communication with Docker Engine, and REST client does not have to implement it by its own. Anchore Engine also has much wider API and the model for communication contains more data. It is also worth mentioning that this API requires authorisation, and in this case it was basic authorisation model. The whole process starts from sending image name to Anchore Engine. Afterwords the image is scanned and during this process *bbdivr-anchore* performs a request every ten seconds to check if the image has been analysed. The next step is to get a manifest file from Anchore Engine REST API. Unique layer identifiers are extracted from this manifest file and then these IDs are used to fetch information about vulnerabilities in every layer. Figure 5.7 shows UML sequence diagram presenting the process of getting information about vulnerabilities from Anchore Engine.

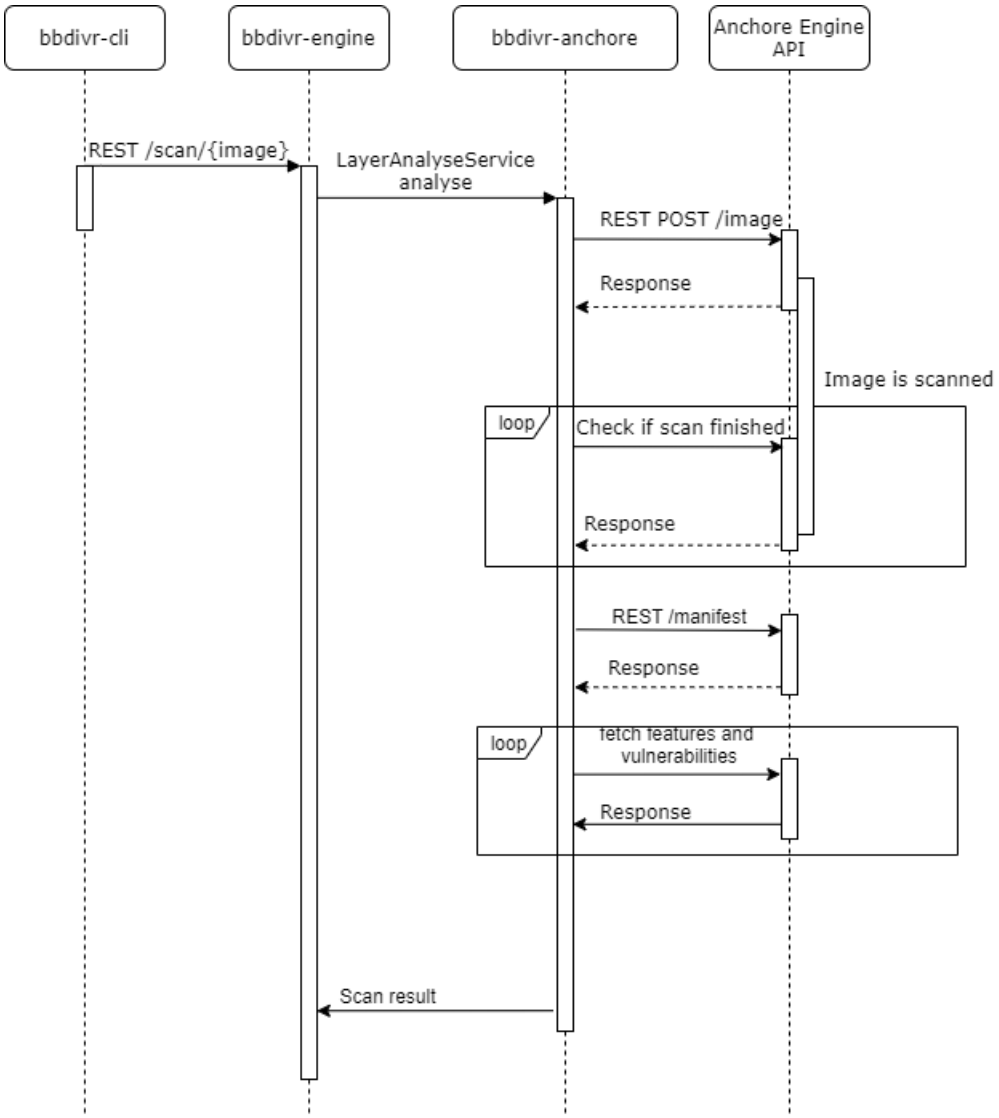

Figure 5.7: Sequence diagram of the bbdivr-anchore component

5.5 Command Line Interface

The last element of the system, that will be presented is *bbdivr-cli*. This component was created to make it easier to use the *bbdivr-engine* REST API. The functionality of this element is the same as functionality provided by engine, and additionally provides some simple operations on Docker Engine API like listing images. To implement command line interface using Spring components the Spring Shell[11] library was used. The main assumptions of this dependency is to use the annotations to create interface commands. This approach is very good as it is analogous to created other APIs with Spring framework, such as a RESTful API and it also does not require from developers to create a lot of repetitive code.

5.6 Summary

This chapter introduced the most important implementations details that have had a key impact on the architecture developed and how the system works. Main elements such as engine, chaincode and integration with external services were covered. Each of these elements was designed to meet the requirements introduced at the beginning of the research.

The next chapter is devoted to the evaluation of the system in order to check its suitability and performance. In order to answer the questions posed, an analysis of the vulnerability of the cloud native application will also be performed. In addition, next chapter includes other evaluations that were performed to compare the external services and used frameworks.

[11] Spring Shell - it is a library, which allows to create a full featured command line application https://spring.io/projects/spring-shell

CHAPTER 6

SYSTEM EVALUATION

This chapter is dedicated to present the usage and evaluation of the system for detecting vulnerabilities in cloud native applications (Section 6.2). The next section (6.3) is dedicated to present the results of interaction with the the blockchain network and to show how the size of the message affects the time of execution, usage of other host resources is also included. The last section (6.4) provides the overview to the Docker Hub images scan results, what serves to present how outdated the images are, and that this is a threat to the safety of the application installed inside of Docker containers.

6.1 Evaluation Environment

The experiments were run on machine with Intel® Core™ i5-7300HQ with 2,50 GHz, 4 cores, 16 GB RAM, 512 GB hard drive. Resources for Docker were limited to 4 cores and 7 GB RAM. Version of Docker Engine was v19.03.12.

6.2 *BBDIVR* Evaluation - Cloud Native Application

In order to evaluate the created solution, a cloud native application was needed, that simulates some types of business processes. Therefore, the best option was to search for an application that has been created with a view to use it for benchmarks and meets the standards of creating cloud native applications.

The application, that was used for evaluation had been developed as part of the DeathStarBench[1] [19] project. Currently, it is possible to use one of three different systems: social network, media service or hotel reservation. A system simulating the social network service was selected for further research.

Social Network benchmark consists of about 30 services, but most of the services are based on the same Docker images. There are 8 different images that, contrary to good practice, use the *latest* version[2]. Table 6.1 presents a list of Docker images, their versions and the time of their creation. In order to perform research, images older than 3 months have been rebuilt[3].

The images presented above were scanned and the vulnerabilities were saved in the blockchain. During the conducted research, the following versions of the images were scanned: latest, 3-month, 6-month and yearly. The problem with the interpretation of the results appears because the images with ”yg397” prefix do not have the correct versioning.

[1] DeathStarBench - https://github.com/delimitrou/DeathStarBench

[2] It is not good practice to use the latest version, because running such an application in different environments, at different time, may depend on different versions of the same image. However, as this project is for testing purposes, the authors probably wanted to rely on the latest version of the available images.

[3] Names of rebuilt images on the charts start from the ”marwin1991” prefix instead of ”yg397”

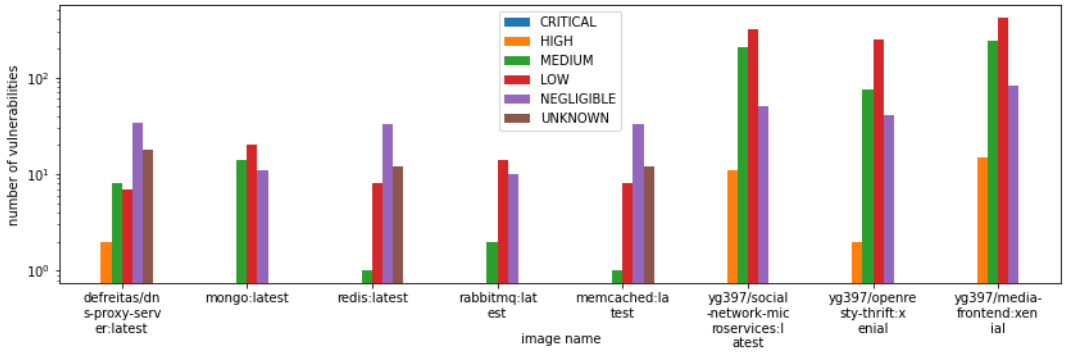

Figure 6.1: Vulnerabilities detected in a Social Network application in latest (original) version with division into Docker images.

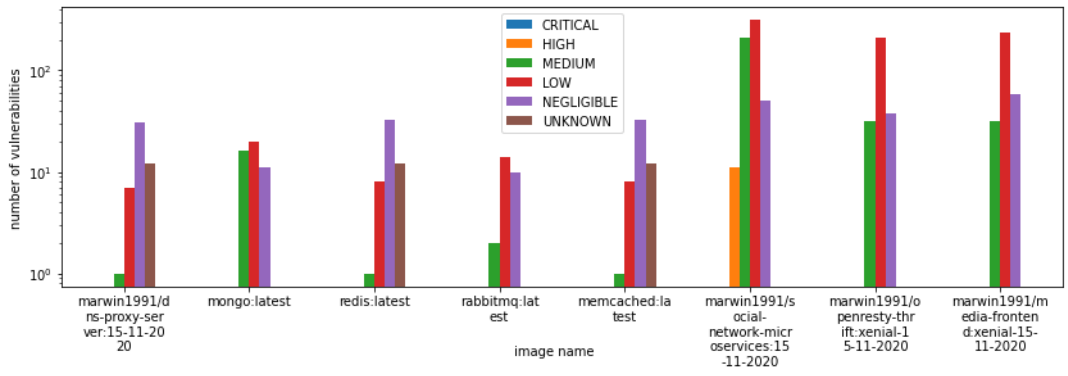

Figure 6.2: Vulnerabilities detected in a Social Network application in latest(rebuilt) version with division into Docker images.

Image	Version	Creation Date	Rebuilt Date
defreitas/dns-proxy-server	latest	2020-01-15	2020-11-15
mongo	latest	2020-09-26	-
redis	latest	2020-10-27	-
rabbitmq	latest	2020-11-06	-
memcached	latest	2020-10-30	-
yg397/social-network-microservices	latest	2019-08-30	2020-11-15
yg397/openresty-thrift	xenial	2020-04-14	2020-11-15
yg397/media-frontend	xenial	2019-05-01	2020-11-15

Table 6.1: Docker images used in Social Network benchmark system.

The results shown in Figure 6.1 presents, that the number of found vulnerabilities in originally built images is enormous, especially for the images with the ”yg397” prefix. In contrast, Figure 6.2 shows the vulnerabilities found in the rebuilt images. The difference of found exposures is large, because some of original images are 7 months old. It is worth paying attention to the fact that applications running on production can often be of similar age or be older.

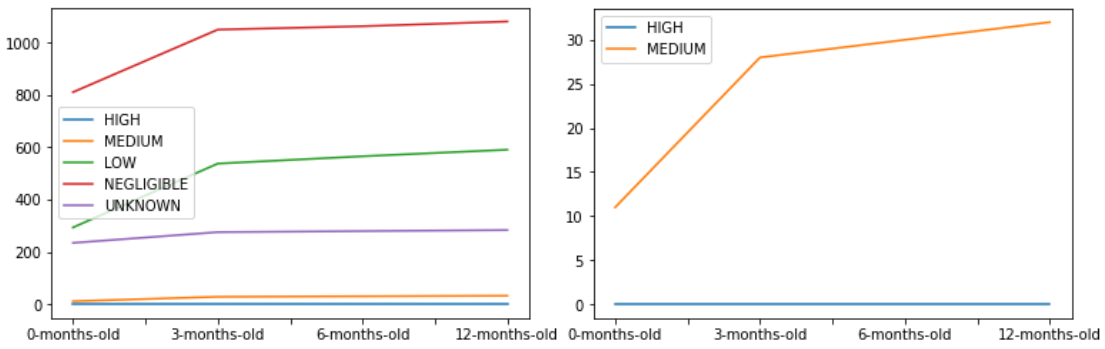

Figure 6.3: Vulnerabilities detected in a Social Network application depending on the age of the Docker image.

Charts showing the correlation between the age of the Docker image and the detected vulnerabilities can be found in Figure 6.3. However, the number of vulnerabilities is so large, that despite the greater number of vulnerabilities in older versions, do not significantly affect the left chart. The chart on the right presents only vulnerabilities with severity high or medium.

The results presented above confirm the proper implementation of the *bbdivr* system and that its use may increase security by spreading information about the identified vulnerabilities. All scan results of the individual images that make up the Social Network application are on the remote GithHub repository[4].

6.3 System Interaction With Chaincode

One of the aspects, that had to be evaluated was the execution time of the writing and the reading operations. The most common case of using blockchain is to store information about some balance, for example, an account balance and transactions between accounts. This type of objects represented in JSON format are still not too large. In the case presented in this thesis the size of data can be quite large. The first idea was to create only the layer contract so that every layer would keep data about vulnerabilities. This approach was incorrect for two reasons. First, there would be large redundancy, because information about vulnerabilities would be duplicated for every layer. And the second is that performing write operation on truly large objects would take really long time.

To ensure that this problem exist, a measurements have been made. This process involved adding longer and longer entities to the blockchain network and measuring the average time. Two lowest and two largest results were excluded

[4]Social Network application scan result `https://github.com/marwin1991/bbdivr/tree/master/bbdivr-evaluation/death_star_bench/social_network`

from the average time results in order to obtain the most reliable ones. The formula below presents how the average values have been calculated ($max_1(N)$ - represents maximum from the set of N numbers and $max_2(N)$ represents the second highest number in this set).

$$avg(N) = \frac{\sum_{n=1}^{|N|} N - max_1(N) - max_2(N) - min_1(N) - min_2(N)}{|N| - 4}$$

The results are presented in a table 6.2 and a graphic representations of them is included in the figure 6.4.

The most important conclusion from this research was that the length under 70 kB had no impact on the performance during the read operations unlike it had on the write operations. The size of 70 kB have reached the limit of 120 seconds of chaincode time execution. This is also the reason why the read time measurements could not be performed, because first the object had to be added to the ledger. Figure 6.4 shows that the relation between the length of the message committed to the ledger and write chaincode execution time is $O(n^2)$ and write chaincode execution time is $O(1)$.

bytes	**write time[s]**	**read time[s]**	**bytes**	**write time[s]**	**read time[s]**
211	13.579631	13.566787	37728	43.235527	12.657676
5569	13.194232	12.749377	43090	52.628066	12.883468
10930	15.097876	12.576883	48450	63.144673	12.564993
16289	18.255191	12.549164	53810	74.665279	12.530155
21650	22.652967	12.576210	59170	87.486027	12.627814
27009	28.290555	12.561198	64530	104.439015	12.576920
32370	35.146715	12.499852	69890	124.222137	12.576285

Table 6.2: The read/write time to Hyperledger Fabric Network

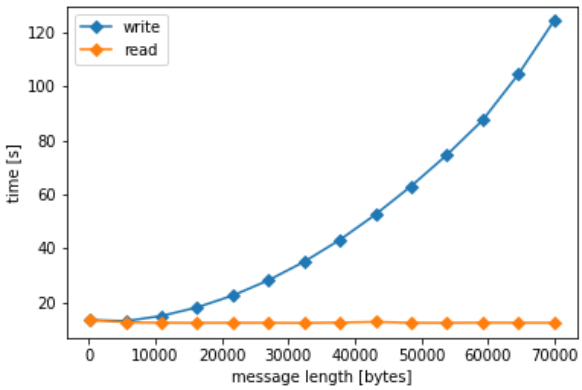

Figure 6.4: The relation between read/write time and message length.

In order to test the performance of the created solution, I examined the levels of usage for CPU, memory and network traffic. Figure 6.5 shows the relationship of CPU consumption over a time interval. The events that affect the course of the chart are: 19:22:00 - enrolment, 19:23:00 - begin scan, 19:25:15 - start adding layers to blockchain, 19:26:38 - finished adding layers to blockchain. The graph shows an orange line that beeps 5 times as many as the scanned image has layers.

Figure 6.6 and 6.7 presents the incoming and outgoing network usage, what confirm that the data flow between nodes is correct, which proves the correctness of the created solution.

6.4 Vulnerabilities Of Docker Hub Images

The next requirement of this thesis was to perform evaluation and measure how many vulnerabilities there are in the most popular images available on the Docker Hub images' registry, so as to be able to generate reports from the scan with division to scan tool type and the severity level. To achieve this goal the new functionality that generated CSV results file was implemented. The generated file was used to prepare charts presenting the outdatness of scanned images. During the

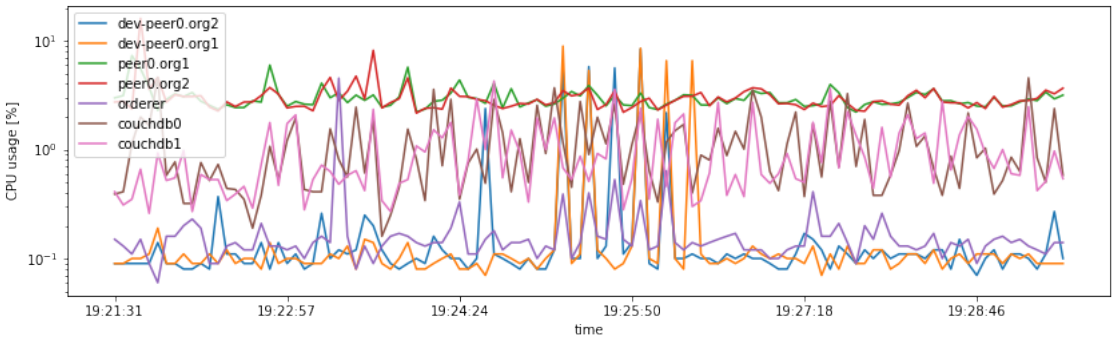

Figure 6.5: CPU consumption by individual blockchain services.

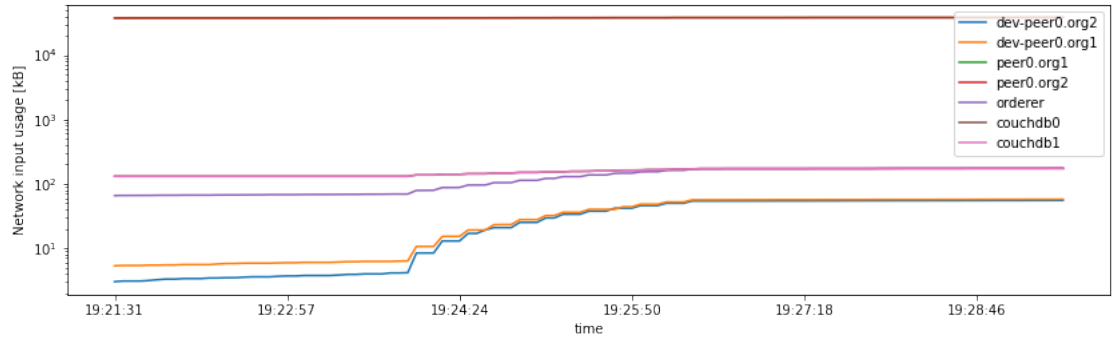

Figure 6.6: Network incoming traffic by individual blockchain services.

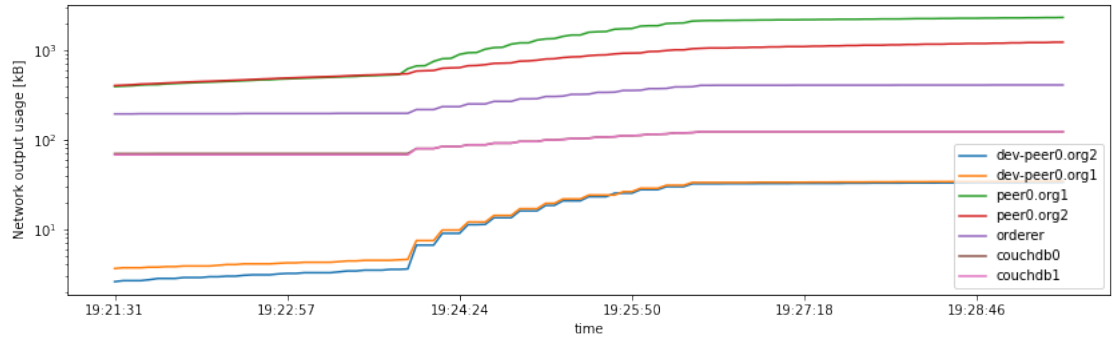

Figure 6.7: Network outgoing traffic by individual blockchain services.

work on this thesis the top 50 most popular images were analysed.

All the results are available under the link included in Appendix A while the most interesting ones are presented in this section.

One of the most popular and vulnerable images is *httpd*. It is an image providing the *httpd server*[5]. It is the most popular solutions for creation of HTTP servers and it is used to provide access to the static applications content or proxy requests.

It is worth mentioning that, this is often a first line of defence in server architecture, because users from the Internet are enabled to access endpoints served by this tool. The second important thing explaining why this image was chosen for evaluation is that, this container after being started is running really long time without updates, because it is working properly.

The decision was to analyse three versions of this image: 2.4.39, 2.4.33, 2.4.23 and results are presented in Table 6.3 and visualised in Figure 6.8. These versions were published with a time difference of about a year and it is certain that they are still used in production. The number of detected vulnerabilities is worrying, because in each a significant number of high-level exposures were detected. The two older ones also have a vulnerabilities with critical-level of severity, which means that this image cannot be used in production environment.

The next Docker image, that results are interesting and valuable to this thesis, is one of the most popular images that are used as base images. It is a Debian operating system and it was scanned in four versions: debian-stable-20200414, debian-stable-20190910, debian-stable-20190122 and debian-stable-20180213. The results are shown in the Figure 6.9 and it is clear, that the number of vulnerabilities with high severity increasing rapidly and it should not be forgotten, that debian image is used by many services. In all of mentioned images the high-level vulnerabilities were detected, what proves that even the use of new images may expose

[5]Apatche HTTPD Server Project `https://httpd.apache.org/`

	httpd:2.4.39			httpd:2.4.33			httpd:2.4.23		
Severity	**Cl.**	**An.**	**Sum**	**Cl.**	**An.**	**Sum**	**Cl.**	**An.**	**Sum**
CRITICAL	0	0	0	14	0	14	25	0	25
HIGH	0	60	60	105	75	105	224	181	224
MEDIUM	0	132	132	200	165	200	336	301	336
LOW	81	66	81	120	71	120	162	104	162
NEGLIGIBLE	151	79	151	122	102	122	132	102	132
UNKNOWN	230	21	230	12	9	12	13	10	13

Table 6.3: The number of vulnerabilities detected in images httpd:2.4.39 - httpd:2.4.33 - httpd:2.4.23

the system to attacks.

Figure 6.10 presents the results of scanning the image responsible for one of the popular databases - PostgreSql[6]. The research were conducted on 10 successive versions, from 9.6.20 to 9.6.10. The presented chart also includes the dates of building individual images in order to better present the correlation between the image age and number of vulnerabilities found.

These results are valuable in one more aspect, because they provide information on how correctly Clair and Anchore Engine detect vulnerabilities. It refers directly to the paper discussed in the chapter 3 written by Berkovich et al. [5] and presents benchmark to test scan tools. The results obtained are consistent with those presented in that paper.

[6]PostgreSQL also known as Postgres, is a free and open-source relational database management system (RDBMS)

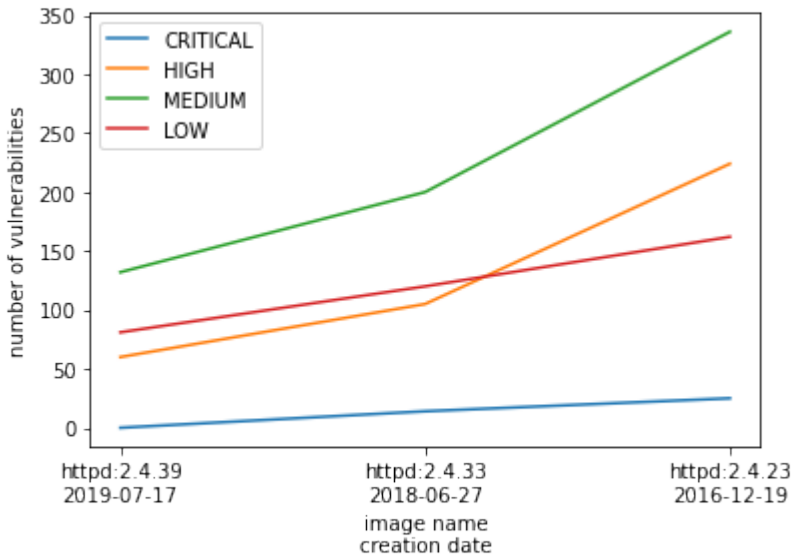

Figure 6.8: Vulnerabilities found in three versions of the httpd Docker image.

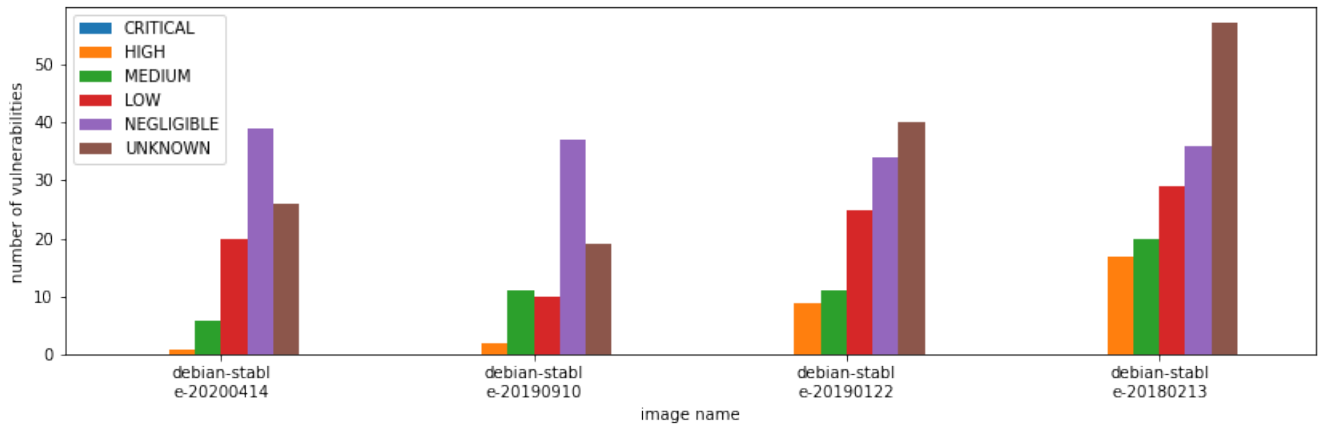

Figure 6.9: Vulnerabilities found in three versions of the Docker image providing Debian operating system.

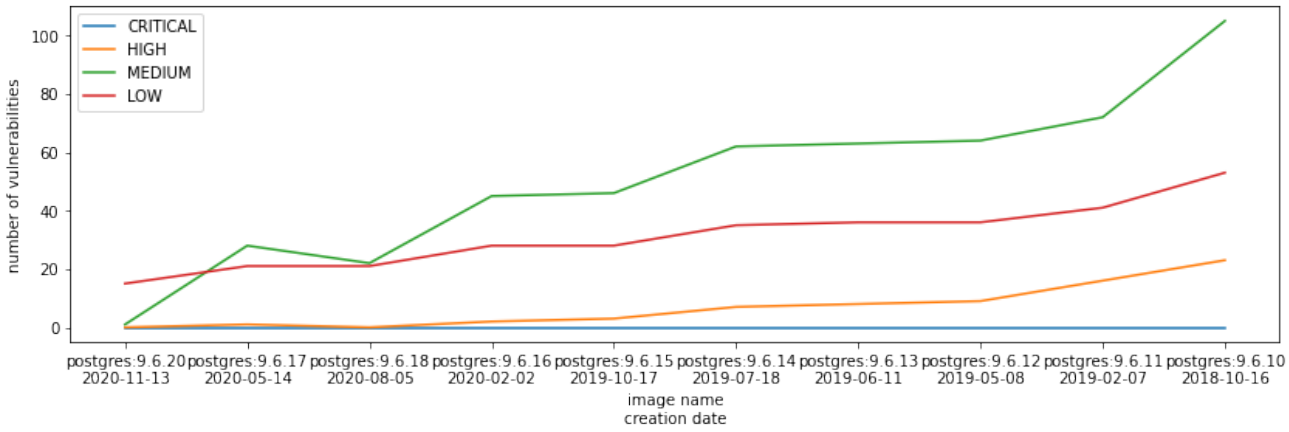

Figure 6.10: Vulnerabilities found in ten different version of postgres Docker image.

6.5 Summary

This chapter presents an evaluation of the *bbdivr* system, which shows that it is operational and its performance is comparable to other systems built with usage of Hyperledger Fabric[32]. In addition, the chosen approach allows to increase the number of detected vulnerabilities by aggregating information from various places. In the presented solution, 3 sources are used: Anchore, Clair and adding manual vulnerabilities. Another confirmed aspect is the use of the system to scan the cloud native application and search for vulnerabilities in all images, which was presented in the example of the Social Network application. A scan of popular images from the Docker Hub repository was also performed. About 200 CSV and PNG files with the results are included in the project, which prove that most Docker images have dangerous vulnerabilities.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter is divided into two sections. Section 7.1 is dedicated to present thesis summary and conclusion. Section(7.2) presents the future work and how the system and some parts of it can be used during researches on the blockchain networks or detecting vulnerabilities in cloud native applications.

7.1 Thesis Conclusions

This master's thesis presents analyse on the use of blockchain networks as the registry for persisting information about vulnerabilities found in could native applications with the possibility of defining different levels of trust. The scope of research was large and concerned blockchain-based systems, smart contracts, detecting vulnerabilities as well as measuring the outdatedness of Docker images.

The created *bbdivr* system was written by means of Spring framework, and Hyperledger Fabric was used for the blockchain creation. The system was integrated with Clair and Anchore Engine to provide information and analyse images.

The results of the system evaluation present current number of exposures in popular images available in Docker Hub and confirm related work in this field of study.

Using blockchain for persistent information, that should be distributed between nodes, in my opinion is really promising, because data are connected with users, that submitted them and this allows to define complicated trust policies. The biggest advantage of this approach is that it allows corrupted nodes to exist as opposed to databases. The second important advantage is the idea of smart contracts, which define an available logic for organisations and users that are in the network. However, blockchain has a downside compared to databases, which is efficiency and time of operation [14][7]. Nevertheless, the current trend and research show that in the future, distributed ledger solutions may improve performance.

The created system - *bbdivr* - meets all requirements presented in this thesis and is a proof of using Hyperledger as a framework to develop a good quality blockchain network. This system also presents an approach to the implementation of a distributed registry, that can be used with benefits in various areas of the IT industry.

The general outcome of this thesis is positive, because all the research challenges were fulfilled. Created system and all the results available as open source project will be valuable for the future work during developing or prototyping systems similar to this one.

The last important matter, which is worth mentioning, is that this thesis is providing proofs that security is really important in all aspects of developing production grade software including containerisation.

7.2 Future Work

The future development potential of the system is quite large. One of the new main functionalities could be the subscription of Docker images identifiers, and after modification of the ledger state of this particular ID or the one of parents'

IDs, the notification would be send to the subscriber. This functionality can be defined as a dynamic service discovery used, for example, in IoT networks [30]. This feature should be implemented with the functionality responsible for creating some sort of policies that will describe, for example, the level of severity or specific vulnerabilities that should be ignored.

The proposed architecture also provides opportunity to implement clients to a new provider of vulnerabilities' data. The system is ready to be used, in this case for benchmarks of scanning tools. And future research question can be following: how to integrate system based on blockchain networks with continuous integration and delivery tools used nowadays for automate deployment.

APPENDIX A

GitHub Repository

The work on this thesis, created solution, all meaningful results and other useful things are distributed as an open source project on the MIT License at GitHub repository hosting. The link to repository is:

`https://github.com/marwin1991/bbdivr`

Evaluation Results

The evaluation results were uploaded to GitHub repository, so that it could be used during different researches in the future.

`https://github.com/marwin1991/bbdivr/tree/master/bbdivr-evaluation`

REFERENCES

[1] Anchore. https://docs.anchore.com/current/. Accessed July 2020.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, and et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[3] App container (appc). https://coreos.com/rkt/docs/latest/app-container.html. Accessed May 2020.

[4] Jim Armstrong. Snyk and docker partner to secure containerized applications. https://snyk.io/blog/snyk-docker-secure-containerized-applications/. Accessed July 2020.

[5] Shay Berkovich, Jeffrey Kam, and Glenn Wurster. UBCIS: Ultimate benchmark for container image scanning. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association, August 2020.

[6] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[7] Si Chen, Jinyu Zhang, Rui Shi, Jiaqi Yan, and Qing Ke. A comparative testing on performance of blockchain and relational database: Foundation for applying smart technology into current business systems. In *International Conference on Distributed, Ambient, and Pervasive Interactions*, pages 21–34. Springer, 2018.

[8] Clair. https://github.com/quay/clair. Accessed July 2020.

[9] Cloud native computing foundation technical oversight committee. https://github.com/cncf/toc. Accessed September 2020.

[10] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[11] Common vulnerabilities and exposures. https://cve.mitre.org/, 2020. Accessed May 2020.

[12] C. Davis. *Cloud Native Patterns: Designing change-tolerant software*. Manning Publications, 2019.

[13] Peter de Lange, Bernhard Göschlberger, Tracie Farrell, and Ralf Klamma. A microservice infrastructure for distributed communities of practice. In *European Conference on Technology Enhanced Learning*, pages 172–186. Springer, 2018.

[14] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[15] TO DOCKER. 4. from chroot over containers. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 2, 2016.

[16] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, And Design*. Pearson Education India, 2006.

[17] Ethereum introduction. `https://github.com/ethereum/wiki/wiki/Ethereum-introduction`. Accessed March 2020.

[18] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[20] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[21] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. Technical lag in software compilations: Measuring how outdated a software deployment is. In Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli, editors, *Open Source Systems: Towards Robust Practices*, pages 182–192, Cham, 2017. Springer International Publishing.

[22] Munawar Hafiz and Ming Fang. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering*, 21(5):1920–1959, 2016.

[23] Oscar Henriksson and Michael Falk. Static vulnerability analysis of docker images, 2017.

[24] Scott Hogg. Software containers: Used more frequently than most realize. *URL: http://www. networkworld. com/article/2226996/cisco-subnet/software-containers–used-more-frequently-than-most-realize. html*, 2014.

[25] Docker Inc. Docker hub. https://hub.docker.com/. Accessed March 2020.

[26] Markus Klems, Jacob Eberhardt, Stefan Tai, Steffen Härtlein, Simon Buchholz, and Ahmed Tidjani. Trustless intermediation in blockchain-based decentralized service marketplaces. In *International Conference on Service-Oriented Computing*, pages 731–739. Springer, 2017.

[27] John Klensin. Role of the domain name system (dns). *Internet Request for Comments: RFC*, 3467, 2003.

[28] Menno Lageman and Sun Client Solutions. Solaris containers—what they are and how to use them. *Sun BluePrints OnLine*, pages 819–2679, 2005.

[29] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken soa triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 22–28, 2007.

[30] Hessam Moeini, I Yen, Farokh Bastani, et al. Summarization in semantic based service discovery in dynamic iot-edge networks. *arXiv preprint arXiv:2009.02858*, 2020.

[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[32] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, and Ali Bou Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018, 2018.

[33] U.s. national vulnerability database. https://nvd.nist.gov/, 2020. Accessed May 2020.

[34] Manoj Parameswaran, Anjana Susarla, and Andrew B Whinston. P2p networking: an information sharing alternative. *Computer*, 34(7):31–38, 2001.

[35] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2017.

[36] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. Blockchain-oriented software engineering: challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171. IEEE, 2017.

[37] Quorum whitepaper. `https://github.com/jpmorganchase/quorum/blob/master/docs/Quorum%20Whitepaper%20v0.2.pdf`. Accessed March 2020.

[38] Michael Rambold, Holger Kasinger, Florian Lautenbacher, and Bernhard Bauer. Towards autonomic service discovery a survey and comparison. In *2009 IEEE International Conference on Services Computing*, pages 192–201. IEEE, 2009.

[39] Ian Robinson. Consumer-driven contracts: A service evolution pattern. *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, pages 101–120, 2006.

[40] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and Aziz Mohaisen. Exploring the attack surface of blockchain: A systematic overview. *arXiv preprint arXiv:1904.03487*, 2019.

[41] Johannes Sedlmeir, Hans Ulrich Buhl, Gilbert Fridgen, and Robert Keller. The energy consumption of blockchain technology: beyond myth. *Business & Information Systems Engineering*, pages 1–10, 2020.

[42] J. Shah and D. Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019.

[43] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[44] Alyssa Miller Sharone Zitzman. The state of open source security 2020. https://snyk.io/open-source-security/. Accessed July 2020.

[45] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[46] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

[47] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptol. ePrint Arch.*, 2016:1159, 2016.

[48] Spring framework. https://spring.io/. Accessed May 2020.

[49] Stack overflow developer survey results 2019. `https://insights.stackoverflow.com/survey/2019#technology`. Accessed: 2020-03-31.

[50] Trivy. https://github.com/aquasecurity/trivy. Accessed July 2020.

[51] D Turk and Jonathan Bausch. Virtual linux servers under z/vm: security, performance, and administration issues. *IBM systems journal*, 44(2):341–351, 2005.

[52] Ahmed Zerouali, Valerio Cosentino, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Tom Mens. ConPan: A Tool to Analyze Packages in Software Containers, May 2019.

[53] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities and bugs, 2018.

[54] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.