Informatics

Injecting Shared Libraries with LD_PRELOAD for Cyber Deception

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Patrick Kern, BSc
Matrikelnummer 11807870

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Mitwirkung: Dipl.-Ing. Mario Kahlhofer

Wien, 5. Dezember 2023

Patrick Kern | Jürgen Cito

TU WIEN Informatics

Injecting Shared Libraries with LD_PRELOAD for Cyber Deception

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Patrick Kern, BSc
Registration Number 11807870

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Assistance: Dipl.-Ing. Mario Kahlhofer

Vienna, 5th December, 2023

Patrick Kern

Jürgen Cito

Erklärung zur Verfassung der Arbeit

Patrick Kern, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Dezember 2023

Patrick Kern

Acknowledgements

First, I would like to thank my supervisor, Jürgen Cito, for enabling me to work on this topic, which contains low-level topics like shared library implementation in C and simultaneously creating and managing cloud clusters.

Secondly, I want to thank Mario Kahlhofer for introducing me to cyber deception, his support with the prototype and all the problems that occurred during development. I also want to thank Sören Henning for his help in designing the benchmarks.

There are, of course, a lot of non-technical factors behind the scenes. Therefore, I want to thank Carina for the good and the not-so-good days and nights. And, of course, I also want to thank my family without this thesis and the whole degree would not have been possible.

Lastly, I also want to thank Scue and Fabs for all the fun we had during our studies (and hopefully will have in the future).

Kurzfassung

Seit mehr als drei Jahrzehnte haben Cyber-Täuschungen gezeigt, wie Angreifer entdeckt, analysiert und getäuscht werden können. Eines der Probleme, das die Forscher in dieser Zeit zu lösen versuchten, war die Automatisierung und Orchestrierung der Täuschungssysteme, um die Komplexität für den Benutzer zu minimieren und die Akzeptanzrate von Cyber-Täuschungen zu erhöhen. Darüber hinaus muss die Forschung die Echtheit der Täuschung bewahren, damit die Angreifer nicht misstrauisch werden und ihre Interaktion mit dem System abbrechen. Diese Eigenschaften sind besonders schwer in komplexen Cloud-Umgebungen zu erreichen, die sich im Laufe des Tages schnell verändern. Eine Lösung für diese Eigenschaften ist die Verwendung von Function-Hooks. Obwohl Function-Hooks im Bereich der Malware-Erkennung und -Täuschung erforscht werden, bestehen im Bereich des Cloud Computing noch Forschungslücken.

In dieser Arbeit analysieren wir Function-Hooks in Bezug auf Shared Libraries für HTTP-Täuschung in Cloud-Umgebungen. Im Detail haben wir die Umgebungsvariable LD_PRELOAD und ihre Möglichkeit, eine benutzerdefinierte Binärdatei bei Prozessstart zu laden, um Shared-Library Methoden zu überschreiben, analysiert. Auf der Grundlage der vorhandenen Literatur haben wir einen Prototyp entwickelt, der zwei Täuschungsmethoden einführt, um HTTP-Antworten zu täuschen. Nach der Bereitstellung in einem Cluster kann sich unser Prototyp in neu erstellte Ressourcen innerhalb eines Produktionssystems einschleusen. Die Täuschungselemente können während der Laufzeit des Produktionssystems durch Änderung einer globalen Konfigurationsdatei modifiziert und deaktiviert werden. Schlussendlich haben wir eine Leistungsbewertung durchgeführt, die zeigte, dass unser Täuschungsprototyp zu einer durchschnittlich 6,72% langsameren Antwortzeit führte, wenn er aktiviert war, und zu 1,58%, wenn er deaktiviert war.

Abstract

For over three decades, cyber deception has shown how adversaries can get detected, analysed, and deceived. Throughout this time, one of the problems researchers tried to address was the automation and orchestration of the deception systems to minimize the user's complexity and increase the adoption rate of cyber deception. In addition, research must still preserve the deception's genuineness so that attackers do not get suspicious and stop their interaction with the system. Such attributes are especially hard to achieve within complex cloud environments, which have rapid changes throughout the day. One solution to address those attributes is by using function hooks. Even though hooks are explored in the area of malware detection and deception, research gaps still exist in the area of cloud computing.

In this work, we analyse function hooking of shared libraries for HTTP deception inside cloud environments. In more detail, we analysed the LD_PRELOAD environment variable and its possibility to preload a custom binary file to hook shared library methods. Based on existing literature, we created a prototype to inject two deception tactics to make HTTP responses deceptive. Once deployed to a cluster, our prototype can inject itself into newly created resources inside a production system. The deceptive elements can be modified and deactivated during the runtime of the production system by modifying a global configuration file. Finally, we conducted a performance evaluation showing that our deception prototype resulted in an average of 6.72% slower response time with activated deception elements and 1.58% with deactivated deception elements.

Contents

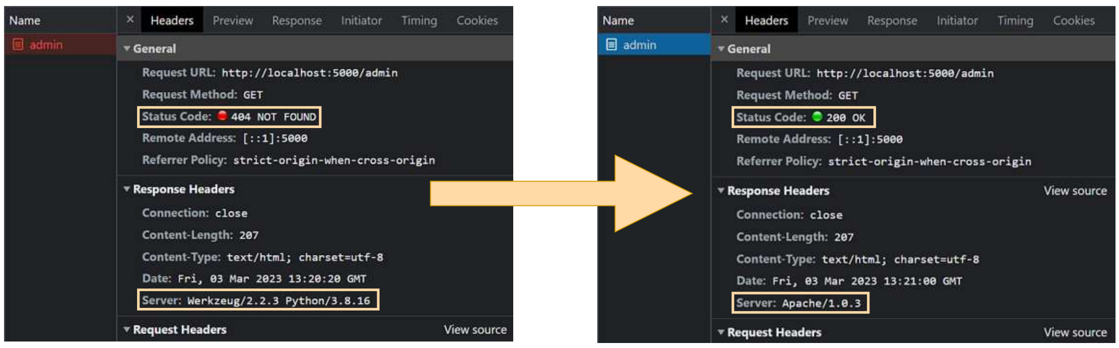
CHAPTER 1

Introduction

For over three decades, cyber deception techniques have been utilised to detect, analyse, and deceive attackers. Even if the attackers know that deceptive elements exist in a system, those deceptive elements still impact the attacker's behaviour [1]. One of the most commonly known deception techniques is the honeypot, which tries to mimic a whole system or protocol and uses comprehensive monitoring to analyse all interactions. Contrary to that, honeytokens are decoy data, like credentials, that will be detected when used. Deception techniques usually do not interact with the actual application itself. Therefore, they are hidden from the average user and can generate solid indicators for malicious behaviour.

Despite the variety of existing developed techniques, the adoption in real-world applications still needs to be improved. As stated by Lance Spitzner, cyber deception was not held back by the concept but by the technology [2]. To tackle this problem, recent solutions try to automate and orchestrate the deployment as well the lifecycle of the deception itself, reducing setup and maintenance costs [3]. Specifically for HTTP-based deception inside a cloud environment, common techniques are based on honeypots [4] or proxies [5, 6]. A downside of such approaches is that they are outside the container, making it hard to put deceptive elements inside a production system.

An established method to address the adaptation of a system inside the OS is function hooking [7]. Even though different operating systems offer different hooking capabilities, most have shared libraries. Hooking shared library is already a used concept for malware identification and malware deception [8], but to our best knowledge, no research evaluated hooking the shared library specifically for deception inside a cloud environment.



```
Name    × Headers  Preview  Response  Initiator  Timing  Cookies
admin   ▾ General
            Request URL: http://localhost:5000/admin
            Request Method: GET
            Status Code: ● 404 NOT FOUND
            Remote Address: [::1]:5000
            Referrer Policy: strict-origin-when-cross-origin
          ▾ Response Headers
            Connection: close
            Content-Length: 207
            Content-Type: text/html; charset=utf-8
            Date: Fri, 03 Mar 2023 13:20:20 GMT
            Server: Werkzeug/2.2.3 Python/3.8.16
          ▾ Request Headers                              View source
```

```
Name    × Headers  Preview  Response  Initiator  Timing  Cookies
admin   ▾ General
            Request URL: http://localhost:5000/admin
            Request Method: GET
            Status Code: ● 200 OK
            Remote Address: [::1]:5000
            Referrer Policy: strict-origin-when-cross-origin
          ▾ Response Headers                             View source
            Connection: close
            Content-Length: 207
            Content-Type: text/html; charset=utf-8
            Date: Fri, 03 Mar 2023 13:21:00 GMT
            Server: Apache/1.0.3
          ▾ Request Headers                              View source
```

Figure 1.1: Example how an HTTP response can be made deceptive.

1.1 Contribution

This thesis investigates the modification of shared libraries inside a Kubernetes cluster for a cyber deception use case. Figure 1.1 demonstrates the two deceptive methods we aimed to implement. The methods are called status code tampering and version trickery, as presented by Fraunholz et al. [6]. Additionally, we add the requirement that the administrator of the deception system can specify an endpoint on which the status code tampering will be active to showcase that the system can link a specific request to its response.

We expect that tracing through multiple methods is not trivial since applications may use methods of the shared library in different ways (for example, sending a payload in one vs multiple requests) or even use different methods (for example, send vs write method). Therefore, we first analyse how a Java and a Python application use the shared library. Those languages are selected since they are both well-used for backend applications, have existed for over 20 years, have a well-supported community, and related libraries, language interpreters, and compilers are open source.

Secondly, we use the knowledge gained from the first step to implement a simple LD_PRELOAD deception system. The system's goal is to deploy it once, and afterwards, the deception is automated for newly created services. The deceptive elements are demonstrated in Figure 1.1 as described at this section's beginning.

Finally, we will conduct a performance evaluation for our prototype to get insight into the amount of overhead that is introduced by our system. We expect to see a similar overhead as protocolled in the hooking review done by Lopez et al. [7].

This thesis is guided by the following research questions:

1. How can a socket connection in Linux be traced by injecting shared libraries?
2. What are architectural trade-offs when designing a deception system with LD_PRELOAD?
3. How does the deception system with LD_PRELOAD affect the performance of the original applications?

1.2 Structure of the Thesis

The rest of this thesis is structured as follows: First, Chapter 2 provides the background in cloud computing, function hooking, and cyber deception. Secondly, Chapter 3 acknowledges the related work of this thesis. Then, Chapter 4 introduces the methodology for the three research questions. Afterwards, each research question is addressed in its own chapter, starting with Chapter 5, which analyses how socket connection can be traced. The architectural trade-offs of a deception system with LD_PRELOAD are then discussed in Chapter 6 before concluding a performance evaluation in Chapter 7. Finally, Chapter 8 concludes this thesis.

CHAPTER 2

Background

In order to understand the chapters later on, some prior knowledge is needed. Therefore, this chapter gives an overview of the most important concepts for this thesis. Starting with Section 2.1, the concepts of cloud computing and Kubernetes are explained. Section 2.2 explains the layers of a computer system, and the function hooking is a possibility with a special focus on LD_PRELOAD. The last section, Section 2.3, states what cyber deception is as well as its taxonomy.

2.1 Cloud Computing

One of the most used definitions for cloud computing is provided by NIST [9], which defines a cloud with three service models, four deployment methods, and five essential characteristics. Subsection 2.1.1 explains the service models in more detail since they are essential to understanding what layers we have to manage inside the cloud. Afterwards, we will elaborate on the essential characteristics in Subsection 2.1.2 to understand which characteristic a prototype designed for cloud computing has to support. Conversely, deployment models impose no influential factors for deception in this thesis and are, therefore, not further explained. Lastly, we explain in Subsection 2.1.3 the major components of Kubernetes we used for developing our prototype.

2.1.1 Service Model

"As a service" refers to the various services offered by a cloud service provider (CSP). Examples of CSP are Amazon Web Services (AWS), Google Cloud, and Microsoft Azure. These services are provided over the internet instead of having the resources on a server on-site, also often called on-premises. Less costs, better scalability, and fewer personal resources needed for maintenance are often seen as advantages by the consumers of CSPs.

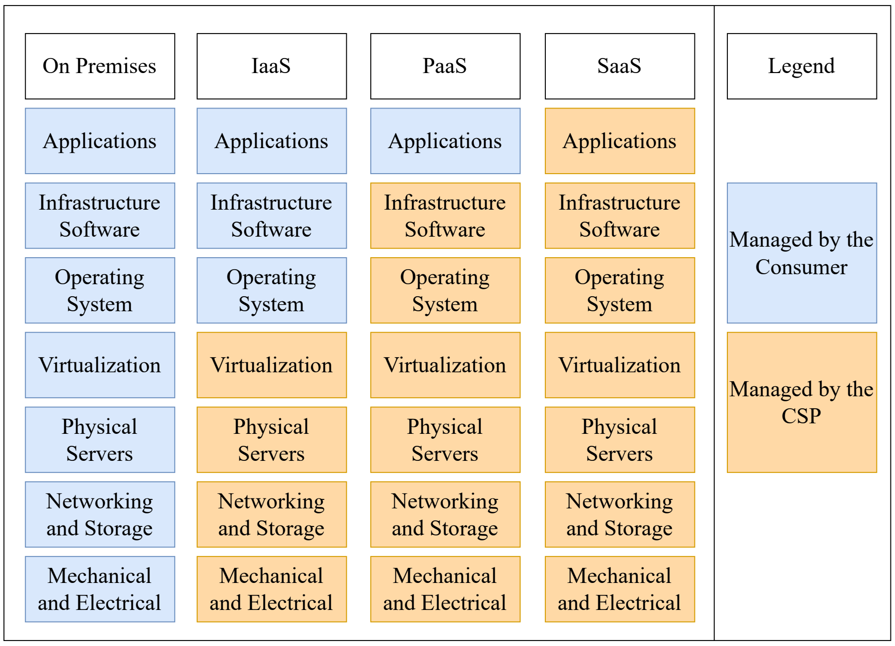


Figure 2.1: Most common as-a-service offerings compared to on-premises solutions [10].

NIST defines the three most known services available in cloud computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Guided by the description of Rountree and Castrillo [10], we created Figure 2.1 that summarizes the difference between those services. The rest of this subsection will go into the details of the services.

Infrastructure as a Service

IaaS provides the infrastructure for running an operating system in the cloud. On the one hand, IaaS excludes all physical engagement needed to provide software services compared to an on-premises solution. On the other hand, it still enables the consumer to choose the operating system running on the server and the infrastructure software and application. Compared to PaaS and SaaS, IaaS offers the most flexibility by only managing the physical servers with networking, storage, power supply, and physical access.

Example: Amazon Elastic Compute Cloud (EC2) and Google Compute Enginge (GCE)

Platform as a Service

On top of IaaS, PaaS additionally offers the operating system (OS) layer as well as the infrastructure software. It includes essential tools and services like development frameworks, databases, application hosting, and application management tools. This allows developers to focus on the coding and application logic part rather than managing servers and software components.

Example: AWS Elastic Beanstalk, Google App Engine and Azure's App Service

Software as a Service

Lastly, SaaS offers applications with all the underlying layers to scale the application depending on consumer needs. Instead of purchasing and installing software locally, users access it through a web browser, making it cost-effective and convenient. SaaS eliminates the need for software maintenance, updates, and hardware infrastructure management, as the SaaS provider handles these responsibilities. Example: Dropbox, Microsoft (Office) 365 and Gmail

2.1.2 Characteristics

NIST defined the five characteristics [9] to point out distinctive attributes that have been fulfilled by the service offered by a provider to call itself a CSP. Those essential characteristics are explained below with the functional requirement for an additional newly built feature, like a deception system, in mind.

On-demand self-service

As the name suggests, on-demand self-service describes the ability of the consumer to request a service without the CSP having to manually provide access to resources like a processor or network storage. This has the advantage of the consumer's ability to quickly acquire new services within seconds rather than minutes or hours. At the same time, the CSP has the advantage of automating user management, and the administrators only have to focus on the availability of the service rather than being in charge of thousands of service requests.

Considering a new feature within an existing cloud service, these characteristics imply that the feature must either be a fixed part of an existing service or be made available with a fully automated process to be on-demand self-service.

Resource pooling

Resource pooling refers to having multiple resources grouped to serve multiple consumers. Physical resources like a CPU or GPU are split virtually into multiple resources to achieve this. Hence, the resources can be "leased" by a consumer until they no longer need them. CSP primarily uses virtualization technology to pool a server, a whole data centre, or even parts of a country together and offer it to consumers. With that, the CSP has the advantage of freely organizing resources and balancing them across the available physical resources to balance their usage. Then, fewer physical resources are needed in total, and the price a consumer has to pay can be reduced.

For a newly implemented feature, this must be considered within its architectural design since, for example, a computing unit does not have to be at the same geographical location as the storage resources needed for the processing. This can have a significant impact on performance. Of course, CSP can offer guarantees that specific resources are geographically close together, but this typically affects the price of the service and hence has an economic impact on the feature itself.

Rapid elasticity

Instead of providing new services like the on-demand self-service, Rapid elasticity provides more resources to a service when needed and freeing up resources if they are not needed. Therefore, rapid elasticity can also be seen as "on-demand self-resources". This essential feature provided by CSP is becoming economically attractive due to the resource pooling characteristics. The process of elastically changing the resources is mostly automated. Because many services experience spikes in usage during a specific time of the day or the year, this characteristic reassures consumers about such cases since the maximum resources they can access often appear unlimited.

A new feature has to be designed in a way that it can scale upwards and downwards as needed. This can impose rudimentary changes within the architecture of a feature.

Measured service

Since services and resources are primarily used on-demand within a cloud environment, pricing is also mainly on a pay-per-use basis. To make costs transparent, CSP monitors the system and uses different measurement systems to evaluate consumer usage. This measurement can, for example, be done on an hourly usage basis of computing units or network connection per minute.

As a result, one must consider if introducing a new metric for a new feature is applicable or if it is better to charge with an existing measurement. Finding the right metrics can be difficult, and using existing resources rather than creating new cost plans is better manageable by the CSP and the consumer. For example, when introducing a new algorithm to process specific data within a more extensive system, it can be better to charge based on processor utilization of the whole system rather than creating a new metric for the usage of the specific algorithm.

Broad network access

Broad network access within cloud computing implies that consumer should be able to access their resources no matter the capability of a device (e.g., smartphones, laptops, and workstations) or the device's operating system (e.g., Android and IOS). To achieve this, a heterogeneous client platform is needed. Additionally, a thin client platform is often preferred over a fat client since consumers might need a higher network connection or computing resources on their devices to use them.

Similar to previous characteristics, this impacts a feature at an architectural level the most.

2.1.3 Kubernetes Overview

Kubernetes[1] (also known as K8s) is one of the most popular systems for deploying and managing cloud applications at a scale. Kubernetes is supported by most vendors with services like Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS). Those services often leverage the IaaS service of the CSP to run a Kubernetes cluster. Kubernetes clusters establish a platform with components that allow the management of applications. For example, we used EKS to create a Kubernetes cluster and an EC2 instance to run the computational load of the cluster.

Kubernetes leverages containers to deploy and manage applications. Kubernetes leverages containers to deploy and manage applications. With the system layers of Rountree and Castrillo [10] in mind, a VM would include the OS layer and everything upwards, making it a good use case for IaaS. The difference between container and virtual machines (VM) is that VMs virtualize a whole hardware layers machine, whereas container virtualizes on top of the OS. Being lightweight makes containers better suited for running microservice applications on a larger scale.

The rest of this subsection describes the main components of Kubernetes that we used for this thesis. Figure 2.2 represents an overview of the components. The components are explained as follows:

1. Starting with the smallest deployable compute unit in Kubernetes, namely **Pods**. Each Pod has a unique IP address inside a cluster and can contain one or more **containers**. Containers within a Pod share the same network and, therefore, can reach each other with localhost. A sidecar container can be used to enhance the functionality of the main container. Use cases for it is monitoring, caching or SSL termination
   Resources within Kubernetes are described and deployed with a manifest file, typically written in YAML. Pods can be deployed with their own manifest or created and managed by a **Deployment**. Deployments have the advantage of leveraging mechanisms like ReplicaSet to ensure that there are always a certain number of Pods of an application up and running. If a Pod shuts down for an error or any other reason, this would get automatically detected. The ReplicaSet would then start a new Pod to meet the minimum number of Pods requirement defined within the Deployment.

2. Deployments run inside a **namespace**. A namespace enables to isolate resources inside a cluster. Within a namespace, the resources must be unique. For example, when a ReplicaSet starts three Pod named test-application, it will add a different hash to each Pod name to make it unique. Different namespaces get automatically created during cluster initialization. The most important one is the *kube-system* namespace, which contains objects created by the Kubernetes system. Examples of

[1] `https://kubernetes.io/`

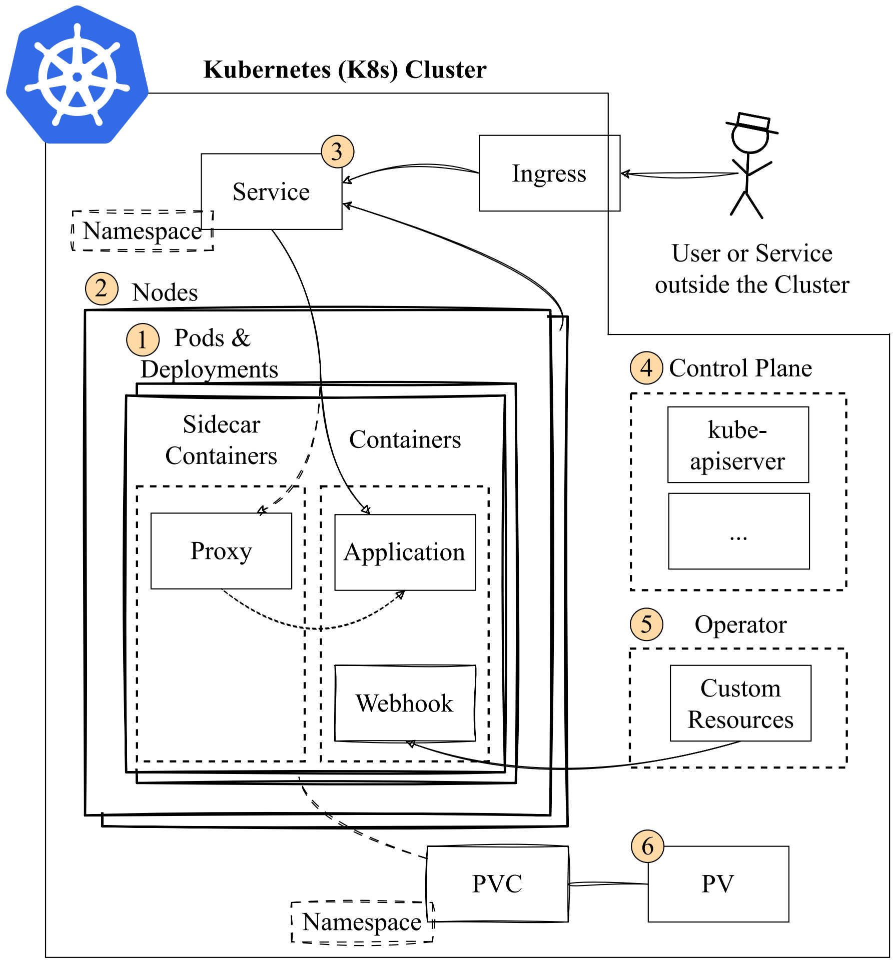


Figure 2.2: An overview of Kubernetes components that are relevant to this thesis. The enumerated components are explained within Subsection 2.1.3.

objects inside this namespace are the *kube-dns* and *kube-proxy* Pod for managing network names and access across multiple nodes.
The **node** is a physical machine or VM that runs the Pod. A typical cluster contains multiple nodes. The assignment of the Pod to a node and the general management is controlled by the control plane (see Enumeration 4). A namespace can span over multiple nodes, and a node typically has multiple namespaces. It is important that namespaces are not bound to nodes but span across the whole cluster. Some resources, like a Service, do not run or are bound to a specific node but exist within a namespace.

3. A **Service** is an option to enable communication with the Pods within a cluster. Pods are typically labelled, which the Service can then use to identify which Pods are related to it. The Service knows which of the Pods are healthy and available. Each node has a *kube-dns* or *kube-proxy* Pod for name resolving. To access a specific

Service, the URL looks as follows: *<service-name>.<namespace>.svc.cluster.local*
A **Ingress** can be defined to forward a request from outside the cluster to a service, which then forwards the request to a specific pod.

4. The **control plane** represents the core of Kubernetes. It manages the cluster and resources like the nodes. The control plane can reside on the same machine as a node but mostly run on a separate VM.
The **kube-apiserver** is a Pod and the core of the control plane. It manages the container lifecycle and acts as an entry point to the Kubernetes system. For example, when a developer deploys resources like a Deployment or a Service, the kube-apiserver is the Pod that creates those new resources.

5. **Operators** are software extensions of Kubernetes that apply the operator pattern. Operators are using custom resources to automate tasks that a human normally would have to perform. The **custom resource** used within this thesis is the **mutating webhook configuration** (MVC), and it is used to modify a configuration before it gets deployed. The MVC can define certain conditions and an address to a webhook. An example of a condition: The manifest has to specify that a Pod is created within a namespace called "deception". Before the kube-apiserver deploys a manifest file, it will send the manifests to the webhook if the MVC condition applies.
The **webhook**, also called mutating admission webhook, is a simple service that processes incoming requests with manifests and responds with a list of changes or with the info that no changes have to be applied. The webhook itself runs inside a container.

6. Kubernetes clusters aims to manage compute resources for a cloud environment, but not storage resource. A **PersistentVolume (PV)** can be used for the cluster to access persistent storage. A PV is a cluster resource just like a node. PVs can be manually created by the cluster administrator, also known as **static PVs**, or the administrator provides storage classes so that the PVs can get dynamically generated as needed, also known as **dynamic PVs**.
Kubernetes does not manage the underlying storage of a PV. Therefore, a different service has to be used for persistent storage. There are different services available for Amazon: Elastic Block Store (EBS), Simple Storage Service (S3) and Elastic File System (EFS). When considering those services, one must consider a PV's access modes. The access modes relate to the node usage and are ReadWriteOnce, ReadOnlyMany or ReadWriteMany. During the development of our prototype, we chose EFS since it supports ReadWriteMany and enables file access with the Network File System (NFS) protocol, which makes it easier to administrate our shared object and configuration files.
Pods can access the storage of a PV via a **PersistentVolumeClaim (PVC)**. PVCs are created within a namespace, and multiple pods can use a PVC as long as they are in the same namespace. Only one PVC can be bound to a single PV and vice versa.

2.2 Function Hooking

The term function hooking describes the process of adjusting the behaviour of a function inside the application, the OS or any other software component. In Subsection 2.2.1, we give an overview of the different layers an application has to pass with its network connection. Additionally, we discuss each layer's capability to trace or modify a payload. Subsection 2.2.2 then focuses on using the LD_PRELOAD for hooking into the traffic of applications.

2.2.1 System layers and Hooking Capabilities

For this thesis, we want to create a cyber deception system that modifies HTTP responses. When looking at a response inside a cluster, the response has to travel through several layers on the device until it reaches its destination, as depicted in Figure 2.3. We choose those layers based on what parts are manageable by us as a consumer of an IaaS offering. Similar to the layers presented by Rountree and Castrillo [10], we have the application layer, the infrastructure software renamed as a runtime layer and the Operating System (OS) represented split into the kernel and the shared libraries. As presented in previous Sections, the traffic of a container can be tunnelled through a sidecar container that acts as a proxy, adding an optional layer until the response reaches its destination. The destination can be another service inside the cluster or an external user. The rest of this subsection will shortly introduce each layer and its capabilities to add new functionality with minimal interactions needed to deploy it, as we are trying to do.

The first layer is the **application** itself. Functionality can be added to this layer within the source code or the binary. The latter is hard to achieve since different languages store the binary data in different formats. On the other side, adding a framework within the source code might be more straightforward, but this would need experts in the source code to implement the framework, which would still take longer to implement compared to the other layers. Hence, it is hard to use the application for deception as a service.

The **runtime**, also known as runtime environment, manages the application lifecycle. Examples of the runtime environment are the Java Virtual Machine (JVM) for Java or the Common Language Runtime (CLR) for Microsoft's .NET Framework. Hooks at the runtime layer can be generated with tools like Java Reflection as presented by Li et al. [11]. This method's downside is that it still needs to be compiled with the source code.

The **shared library** is the first layer that does not need the recompilation of the whole application. Shared libraries are used for commonly used cases like socket connection to be supplied as a compiled library once and serve multiple applications without including the shared library in every application. Examples of shared libraries are shared objects (.so files) on Linux or dynamic-linked libraries (.dll files) on Windows. Lopez et al. [7] explores different hooking approaches for Windows, Linux, macOS, iOS, and Android. The most relevant method for this thesis is using the LD_PRELOAD environment variable as described in more detail in Subsection 2.2.2. Important to add here is that the

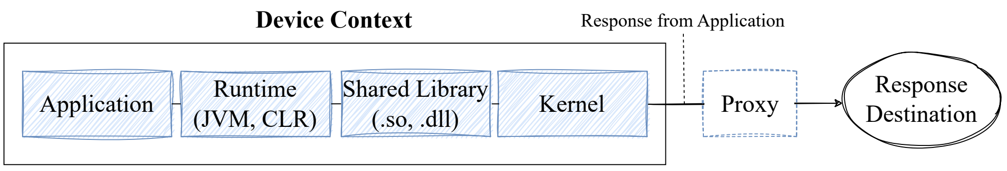


Figure 2.3: Layers of which an application's response must pass until it reaches its destination. The device context describes the layers that can access local resources like files (with the help of the kernel).

application has to link the shared libraries dynamically rather than statically. Otherwise, hooking with methods like the LD_PRELOAD variable is not possible.

The **kernel** layer, as a core part of the OS, manages access to the machines in general, including access to hardware resources. A prominent example specifically for cloud environments is Extended Berkeley Packet Filter (eBPF) as presented by Soldani et al. [12]. The big difference between the previously mentioned methods and the kernel layer is that containerisation occurs on top of the kernel. This means that each application has to be hooked, except for the kernel, since each EC2 instance has one kernel shared by all containers.

Lastly, we consider the **proxy** as a possibility to change network traffic. A typical use case for sidecar proxies in cloud environments is within a service mesh [13]. Service mash enables developers to separate communications in a dedicated infrastructure layer. Contrary to the kernel, a sidecar proxy has to be deployed for each container where the network traffic should be modified. An additional downside of proxy is that there is, in general, less information about the application and its system available. For example, if the application encrypts its traffic, the proxy would have no reasonable possibility of adjusting or even identifying the payload.

2.2.2 Hooking with LD_PRELOAD

LD_PRELAOD is an environment variable used by the dynamic linker of Linux to preload a shared library before all other shared libraries. The value of LD_PRELOAD has to be the shared library name of the library that should be preloaded. The shared library will be loaded at the starting phase of a process, meaning that modification of the shared library file (.so file) or the variable itself has no effect when the process has already started. The rest of this subsection showcases LD_PRELOAD with an example.

We created a small demo to overwrite the response of the Python service. Listing 2.1 represents the code of our shared library, and we named the compiled library "overwrite-send.so". The code itself defines a method *send* in line 7, which has the same name and arguments as the *send* method defined in libc[2]. The first line of the method (line 9) will retrieve the original libc *send* method pointer by calling the dynamic linking loader (*dlsym* method). We do this to call the original *send* method directly instead of

[2]https://linux.die.net/man/7/libc

implementing the send functionality ourselves. We then check in line 11 if the buffer contains the string "Hello World". If so, we call and return the original *send* method with our custom "Overwritten!" string. Otherwise, we will forward the original parameters.

To test this, we created Hello-World Python service (Listing 2.2) that responds with the string "Hello World!" if a request was sent to *http://localhost/*. We then set LD_PRELOAD with the command:

```
export LD_PRELOAD="./overwrite-send.so"
```

Since we used a relative path, we have to start the Python service in the same directory as our .so file. After we start the Python service and send a request, the Python service will call our hooked method, which results in "Overwritten!" being sent in the response.

Listing 2.1: overwrite-send.c: Overwrites all *send* calls that contain "Hello World".

```
1  #include <string.h>
2  #include <sys/types.h>
3  #include <dlfcn.h>
4  
5  ssize_t (*libc_send)(int, const void *, size_t, int);
6  
7  ssize_t send(int sockfd, const void *buf, size_t len,
8               int flags){
9      libc_send = dlsym(RTLD_NEXT, "send");
10 
11     if (strstr(buf, "Hello World") != NULL) {
12         return libc_send(sockfd, "Overwritten!\r\n", 15, flags);
13     }
14 
15     return libc_send(sockfd, buf, len, flags);
16 }
```

Listing 2.2: test-app.py: simple flask demo service that response with "Hello World!".

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def index():
  return "Hello World!\n"
app.run(port=80)
```

2.3 Cyber Deception

Cyber deception refers to techniques to mislead attackers into interacting with simulated data, services or devices. Numerous studies showed that such deception techniques can detect, deceive and delay attackers [1, 14, 15]. For this thesis, we consider three of the deception techniques as relevant: the honeypot, honeytokens and their combination called tripwires.

One of the most known techniques for cyber deception is the **honeypot**. To describe it with Lance Spitzners [16] words: "A honeypot is a security resource whose value lies in being probed, attacked, or compromised". A honeypot often mimics a service or system, and many implementations are publicly available [17]. For example, a honeypot can implement the Secure Shell Protocol (SSH). SSH typically enables remote access to a system and is, therefore, a coveted resource for hackers. If an attacker connects to the SSH honeypot, he or she will only find a fake system with fake resources. Additionally, when one interacts with the honeypot, those interactions will be logged, and the administrator can use those logs to learn more about the attackers and their behaviour.

Contrary to the honeypot, **honeytokens** are fake digital entities without any compute resources [18]. Examples of honeytokens can be real-looking credentials, documents or database entries. Like honeypots, the production system does not use honeytoken, and their purpose is only to attract attackers' attention. Since they have no computing part, honeytokens need an additional system for log interactions with it or mechanisms that detect when honeytokens, like fake credentials, are used.

Tripwires are a combination of a honeypot and a honeytoken which injects lures (i.e., honeytokens) in a system that can be used to interact with decoys (i.e., honeypots) [19]. For this thesis we set the focus on injecting lures automatically and we leave the decoy implementation for future work.

CHAPTER 3

Related Work

This chapter introduces related literature that has similar approaches or influenced us in creating this thesis. Section 3.1 describes different literature on malware detection and deception inside Windows systems, which we consider similar to Linux systems like containers in cloud infrastructure. Section 3.2 addresses additional relevant deception literature focusing on cloud computing.

3.1 Deception as a Service for Malware

Apart from microservices, deception is also used for malware detection and deception on Windows. Alsaleh et al. [20] developed gExtractor, a tool that extracts malware traces of Windows's system (kernel level) and library (shared library level) API calls. Additionally, it uses those traces to generate deception parameters in the form of environment variables, enabling the automation of a deception system. They validated their claims by using honeytokens in the form of file transfer protocol (FPT) passwords to mislead cryptocurrency and credential-stealing malware as well as different ransomware.

Sajid et al. present DodgeTron [21], which builds up on gExtractor and extends it with a machine-learning approach to support real-time deception orchestration. The authors also introduce deception playbooks as a term to impose certain deception schemes for certain malware types. Their deception schemes include fake resources like registry entries and files. With DodgeTron, the authors could automatically deceive 869 out of 953 malware they tested, with a few seconds needed to classify the malware categories. Their analysis to identify new malware types took 27.7 minutes on average.

Islam et al. [3] created CHIMERA that further adapts malware identification using a global adversary tactics and techniques knowledge base. Additionally, they extended the deception playbooks by using an open-source hooking library for Windows called EasyHook [22]. EasyHook is also a hooking method based on the shared library layer, but in contrast to preloading shared libraries with LD_PRELOAD, EasyHook allows the hooking during the runtime of a process. CHIMERA uses hooks to forward malicious processes to what they call a honey factory (HF). The HF has various deceptive tools, like honey tokens, fake web pages and honey-patches. They evaluated their system by running various malware on it, and they proposed that their system take a maximum of 15 milliseconds to decide a deception strategy for detected malware. Including the decision for the strategy, CHIMERA took a maximum of 47 seconds for the deception deployment and orchestration, which they argue is low compared to the time it takes the malware to exploit a vulnerability.

The latest system presented by Sajid et al. is called symbSODA [23]. SymbSODA contains a client application to trace and inject DLLs, similar to what our prototype is doing with the shared libraries. However, symbSODA also contains a server application which communicates with the clients. The server can create HF as needed, enabling the deception system to be offered as a service. Deception as a service with symbSODA contains a centralized server with a deception playbook and the HFs managed by the cloud service provider. Customers can create custom deception schemes for the deception playbook or use default schemes. They then install the symbSODA client on different Windows environments, and the client will connect to the server to successfully deceive malware.
To evaluate the performance overhead introduced by symbSODA, the author compared malware response time with symbSODA deployed on a test system against the test system without deception capabilities. Four different malware types were selected, which performed between 13 and 126 seconds without deception. With the deception system, the generated performance overhead was between 7% and 15%.

3.2 Microservice Deception

As presented by Soldani et al. [12], extended Berkeley Packet Filter (eBPF) is a technology used for observability and security inside Kubernetes. The eBPF system is based on the kernel layer and, therefore, only has to be installed once per VM to trace all pods running on it. Like LD_PRELOAD, eBPF uses function hooking to inject code into the system.
One application that is using eBPF is Pixie[1]. Similar to this thesis, Pixie can trace HTTP traffic by overwriting *send* and *recv* calls, but they achieve that on the Kernel layer. They also support HTTPS tracing, which is impossible in the kernel since encryption happens in the application layer. Most applications do not implement the TLS encryption on their own but rather use libraries like OpenSSL. Moreover, OpenSSL uses a shared library to enable applications to dynamically link it. As presented by Nano [24], Pixie uses this fact to its advantage by hooking the *SSL_write* and *SSL_read* method of OpenSSL in the shared library layer in the same way we did for the *send* and *read* method in this thesis.

[1]https://px.dev

We can achieve similar HTTPS support for our prototype in this way, even though the identification of the file descriptor is not trivial, as mentioned by Nano.

The tripwire framework proposed by Kahlhofer et al. [19] proposes an orchestration model with tripwires for cloud environments. The tripwires are composed of lures and decoys. The lures can be resources like fake credentials to register at a decoy system like a fake service. The authors propose using function hooking to inject such lures or even decoys.

Faunholz et al. [6] created a deception as a service system called Cloxy. The system can manipulate HTTP responses with nine different deceptive elements, out of which we included two for this thesis. Cloxy is a reverse proxy supporting HTTP at its current implementation. However, it can also use HTTPS if the system fully implements the underlying proxy technology. Their approach is about eight times slower, with most overhead introduced through the proxy. They claim that compared to the proxy without deception, the proxy with version trickery does not generate further delay, and only advanced method like obfuscating JavaScript code increases the overhead by a factor of 1.78.

Reti et al. [25] presented a prototype that modifies the HTML code of the HTTP response payload if the request contained a honeytoken. The author used the libnetfilter_queue library to achieve this. This library provides an interface for userspace programs to communicate and modify network packages inside the kernel. Like Pixie, execution inside the kernel limits the libnetfilter_queue technique to unencrypted traffic since additional userspace hooks are needed to intercept encrypted communications like HTTPS.

Araujo et al. [26] introduced the term honey-patches in their prototype called RedHerring. Honey-patches are patches that fix a vulnerability in a service by redirecting exploits to a honeypot with an unpatched version of the service. Patches are made directly in the source code, and therefore, they are at the application layer. The authors use a reverse proxy to orchestrate the deception and create new honeypot containers with the previous service state for malicious requests. They measured the round-trip times of malicious requests and concluded with an overhead of about 250 milliseconds.

Araujo and Taylor [27] used just-in-time patching to update a running production application. They compile their patches into a shared library, which they link with a special hooking agent during runtime. Their performance evaluation of the RTT concluded an overhead between -20% and +10% with the performance improvements arise due to optimizations that the JIT compiler can do in larger patches.

Apart from various systems that add or inject payloads into production systems, there is plenty of literature for honeypots inside cloud environments. For example, Priya and Chakkaravarthy [4] run eight containerized honeypots on multiple VMs at Microsoft Azure Cloud, and Alyas et al. [28] propose a model to integrate and manage honeypots in a multi-cloud platform environment.

CHAPTER 4

Methodology

This chapter establishes the fundamental concept for this thesis. We first define the three research questions (RQs). Afterwards, each RQ is addressed in its own section. Finally, we address threats to validity, potential limitations, and challenges.

4.1 Research Questions

This thesis investigates the usage of hooking shared libraries for cyber deception inside cloud environments by injecting deceptive code modules into specific shared libraries related to the HTTP response. Three research questions were defined to split the topic into the fundamental concept, a proof of concept, and a performance evaluation. The first RQ documents the technical interaction between an application and a shared library inside a Linux OS. The second RQ generates a proof of concept for embedding deceptive elements into a cloud environment. Finally, the third RQ focuses on the performance overhead introduced by the LD_PRELOAD approach. The RQs are defined as follows:

1. How can socket communication in Linux be traced by injecting code with shared libraries?
2. What are architectural trade-offs when designing a deception system with LD_PRELOAD?
3. How does the deception system with LD_PRELOAD affect the performance of the original applications?

4.2 Understanding Libc

RQ1: *How can socket communication in Linux be traced by injecting code with shared libraries?*

Despite the standardized definition of socket connections within the Linux man pages[1], applications still have different options for using it. For example, the man page states the following possible methods for communication when the socket type is SOCK_STREAM: *"Once connected, data may be transferred using* ***read(2) and write(2) calls or some variant of the send(2) and recv(2) calls****."*. Therefore, this research question should answer the usage of specific technologies.

We present the results of RQ1 in Chapter 5. Further, we differentiate Java and Python to understand which methods must be overwritten for the respective language. The differences between those languages will also outline the possible development overhead that must be considered when supporting multiple languages with the same code base.

We use the following steps to comprehend the interaction of the mentioned languages with libc:

1. We create and preload a simple shared library to print a log line when a specified libc method gets called and then call the original shared method. This is used to trace and verify which library methods are called.

2. On the application side, we implement a minimal "Hello World" REST service. This enables us to debug the application to find where the native method calls are.

3. Additionally, we do a static code analysis of the JDK and CPython, which should give support during the debug sessions and enable us to document the findings without the need to run an application.

Since stepping into the native code while debugging a managed application is not trivial, the above steps should reduce the risk of missing or misinterpreting code fragments. For the development of the simple shared library, Devcontainers[2] with the openjdk:17-bullseye image where used. We used the Ubuntu 22.04.3 LTS version on WSL for the dynamic code analysis. On Ubuntu, IntelliJ and PyCharm are used for debugging the technologies. For Java, we selected the latest LTS version 17.0.8+7 of the Eclipse Temurin build of OpenJDK[3] and for Python we used the default version of the apt-get manager, which was 3.10.6-1[4]. For the static code analysis, the equivalent versions we took the open source repositories from OpenJDK repository[5] as well as for Python repository[6].

[1] `https://linux.die.net/man/2/socket`
[2] `https://code.visualstudio.com/docs/devcontainers/containers`
[3] `https://adoptium.net/temurin/releases/?version=17`
[4] `https://packages.ubuntu.com/source/kinetic/python3-defaults`
[5] `https://github.com/openjdk/jdk/tree/jdk-17+8`
[6] `https://github.com/python/cpython/tree/v3.10.6`

4.3 Proof of Concept: Deception System

RQ2: *What are architectural trade-offs when designing a deception framework with LD_PRELOAD?*

After specifying the shared methods needed for tracing and modifying HTTP calls in Chapter 5, we created a proof of concept (PoC) for a deception system. There are different ways with different trade-offs of achieving that. RQ2 will highlight some of the obstacles we encountered using LD_PRELOAD and describe the path we took for this thesis.

We present the use cases and the expected functionality in the first subsection. Afterwards, we define to which extent we will document the findings in Chapter 6.

4.3.1 Requirements for the Deception System

The main goal is to manipulate the HTTP response header of an application to advertise potential weaknesses or vulnerabilities. The two changes (visualised in Figure 1.1) are known as the status code tampering and version trickery technique, defined by Fraunholz et al. [6]. This thesis uses the term wire to indicate the deceptive techniques. As the name wire (derived from tripwire) implies, those changes intend to trick an attacker into interacting with the system in a certain way, which an intrusion detection system can later detect.

We name the first change *response-code wire*, which can set an application's original HTTP response code to a new value, known as status code tampering. Since this would easily break applications if applied to all endpoints, applying the status code change only to responses for requests of specific paths must be possible. For example, an unimplemented "/admin" path could be returning a 200 status code with this wire, which can irritate attackers and their tools. This requirement includes a functional component that tracks incoming requests, determines the path, and preserves the state until the corresponding response message is sent.

The second change is named *HTTP-header wire* , and it changes the value of an HTTP attribute. If the "Server" attribute is chosen, it is equivalent to the version trickery technique, but it is also possible to overwrite different attributes. The specific attribute, as well as its new value, should be configurable. Changing, for example, the value of the "Server" attribute to "Apache/1.0.3" could lead an attacker to exploit vulnerabilities of this version like CVE-2016-4469[7]. The configuration of the wires should be centrally controllable for better manageability. Additionally, changes in the wire configuration should be processed without restarting the whole system. This requirement on runtime updates serves the purpose of enabling more flexibility. We argue that this flexibility allows

[7]https://nvd.nist.gov/vuln/detail/CVE-2016-4469

the system to be managed by an autonomous system like SODA [8]. This requirement also aligns with the tripwire framework proposed by Kahlhofer et al. [19].

Since we propose the deception for cloud environments, we considered the range in which the deception will be active. For example, the test bench should not have any additional system that could affect the test results. Therefore, we set the scope to a single predefined namespace. All resources that are deployed within this namespace will be injected with the LD_PRELOAD prototype, whereas deployments in other namespaces are unaffected by the PoC. Creating and setting a dedicated namespace for external applications also eases the deployment, compared to manually labelling all deployments that should be affected.

4.3.2 Documentation of the PoC

To answer RQ2, the architectural decision and their trade-offs will be pointed out. This should cover and explain the following requirements:

- What are the options to hook shared libraries in cloud environments?
- What are the options for externally updating the state of a preloaded shared library?
  - Explain used YAML configuration, also called honeYAML, to specify the wires.
- How to trace incoming requests throughout different shared library methods with as little performance overhead as possible?

Further, we discuss the design decisions for this PoC.

4.4 Performance Overhead Benchmarking

RQ3: *How does the deception framework with LD_PRELOAD affect the performance of the original applications?*

Performance not only matters from the user perspective, where long loading time might be a reason to leave the website, but also from the developer and commercial perspectives, where less request throughput might imply upscaling resources and, with that, adds additional complexity and costs to the equation. Moreover, considering the deceptive use case, an attacker might unmask deceptive traces due to the longer delay introduced by the deception system. Therefore, a deception system must have as few negative performance implications as possible.

This section defines the setup used for the evaluation presented in Chapter 7. An overview of the test bench (TB) and system under test (SUT) is presented within the first subsection before the different variables leading to the individual benchmark result are described in the other subsections.

4.4.1 Test Bench and System Under Test

The test bench (TB) is the system that executes a benchmark by sending and measuring the system's response time under test (SUT). The metric to test the performance is the round-trip time (RTT), the millisecond it takes from the TB to send the HTTP request until the TB receives the HTTP response message. The TB is implemented as a Python Locust[8] container. The benchmarks are based on the survey on function hooking from Lopez et al. [7]. One benchmark within the survey contained four rounds in which each round conducted file manipulations on 10,000 files. At first, this thesis oriented on the survey by having 40,000 requests for the evaluation with the alternation of adding 10,000 requests up front as a warm-up phase to account for possible web caching. The warm-up requests will be excluded in the evaluation phase. As described in more detail in the evaluation in Chapter 7, this resulted in erroneous data. To overcome this problem, we analysed the warm-up similar to Kahlhofer et al. [29]. The outcome was that the requests were quadrupled, resulting in 200,000 total requests, with 40,000 requests for warm-up for each specific SUT configuration. In addition, the TB container and the SUT containers affected by the benchmark are redeployed before each benchmark to minimize any risk of caching before the benchmark has started.

We evaluated three settings considering the different SUT configurations, resulting in 24 benchmarks. The following settings are explained in more detail in the next subsections:

- Three different states of the LD_PRELOAD prototype (Subsection 4.4.2)
- Two different SUT scenarios, each containing two different endpoints forming a total of four use cases (Subsection 4.4.3)
- Two different server environments (Subsection 4.4.4)

4.4.2 LD_PRELOAD state and Control Group

Considering the LD_PRELOAD system, we labelled three different separations for the benchmark's evaluation :

- w/o wires: This is the control group. We deploy the SUT as intended without any additional shared library.
- w/ wires=f: We deploy the application with a shared library injected, but the wires are set to **false**.
- w/ wires=t: We deploy the application with a shared library injected, and the wires are set to **true**.

[8] `https://locust.io/`

The main objective of the benchmark is to evaluate if the deception system introduces significant performance overhead. Therefore, we separated the SUT by creating an environment where the application is deployed as intended (= *w/o wires*) and one where the deception is active (= *w/ wires=t*). Active, in this case, means that we deploy the application with the shared library and with the wires set to true. We perform this by deploying the same application into two different namespaces, where one of the namespaces has the deception system running.

In addition to the two separations, we added a third option where the shared library is deployed, but the wires are set to false (= w/ wires=f). We have to inject our shared library at the beginning if we want the deceiving element later on since a shared library is preloaded when the process is started. This option is expected to only introduce a small overhead compared to the control group since this option only adds a simple check and then directly links to the original syscalls. Nevertheless, comparing a deactivated deception against the control group leads to the not-skippable overhead for a deception system based on LD_PRELOAD being assessed. Conversely, when comparing the SUT with wires deactivated versus activated, the performance overhead of the specifically implemented wires can be assessed. Those performance differences can significantly differ depending on the intended goal and the optimization of the wires.

4.4.3 Use-Cases Coverage

The use cases consist of an average- and a worst-case scenario with two different endpoints benchmark each. These are labelled within the evaluation as follows:

- /admin (average-case SUT)
- /tools.descartes.teastore.webui/ or /home/ (average-case SUT)
- /benchmark Java or just Java (worst-case SUT)
- /benchmark Python or just Python (worst-case SUT)

It is important to note that the slash at the end of /tools.descartes.teastore.webui/ is essential to address the main page at the TeaStore SUT. Otherwise, a 404 page is shown.

To better understand the performance overhead inside the cloud environment, the microservice reference application TeaStore[30, 31] was chosen. Additionally, we created a new custom benchmark to compare the overhead between a Python and Java application in an isolated environment. Therefore, logical operations similar to [7] were created.

A detailed specification of the two use cases with their endpoints is listed in the following subsections.

Average-Case: TeaStore

TeaStore[9] is a reference application for benchmarking microservices. Their out-of-the-box support for running Locust is limited to running the TB locally. Therefore, we used the official Locust Docker image[10] for the benchmarks to be able to deploy the TB within a cluster. Deploying the TB inside a container promotes better reproducibility and enables testing services inside a cluster without using an Ingress to excess the endpoints. In general, adding more complexity to the TCP connection distance also adds more noise to the results [29]. This would happen when testing over the World Wide Web and entering the cluster with an Ingress versus testing directly inside a cluster. The TB container is deployed within the deception namespace. In contrast, the SUT is deployed inside the benchmark-deception-sut-with and benchmark-deception-sut-without namespace, depending on whether the SUT is deployed *with* or *without* the LD_PRELOAD variable set. With that structure in mind, the TB can access an SUT via the cluster DNS.

For the TeaStore, we present the honeYAML configuration in Listing 4.1. The configuration is based on the deceptive HTTP response introduced by Figure 1.1. To test this deceptive use case, the TB will request the /admin path, and the deception system will overwrite its response with a 200 status code instead of the 404 status code. Additionally, the "Server" attribute will be set to an old and vulnerable version of Apache.

The second use case is the path /tools.descartes.teastore.webui/, which is the main page of the TeaStore. This request also interacts with the other containers of the TeaStore. Such behaviour resembles a typical request on a webpage and will display the impact of the deception when only overwriting the server header.

Worst-Case: Custom SUT

Even though TeaStore is a comprehensive micro-service reference application, its components are all implemented with Java. Hence, to compare the relative performance differences between Java and Python, two custom SUTs were used. The TB setup and testing modality stay the same as with the average-case SUT.

Linux treats file operations similarly to HTTP requests in the sense that both use the same read and write method from libc for communicating over a socket. Therefore, local file manipulation still negatively impacts the response time introduced through the deception system. We used file manipulation for the functionality of the worst-case SUT. Hence, we created a custom SUT similar to the benchmarks done by Lopez et al., [7]. Considering the concrete functionality, the tasks are inherited from Lopez et al., except that the execution sequence was changed. Instead of processing 10,000 files simultaneously, only one file gets manipulated for each HTTP request sent to an endpoint of the application. The endpoint is located at the path /benchmark and will accept POST requests with a "message" attribute as content. The TB will sequentially request

[9] `https://github.com/DescartesResearch/TeaStore/tree/master`
[10] `https://hub.docker.com/r/locustio/locust`

the endpoint 200,000 times and always send the message "Benchmark nr. i", where the index of the current call will be sent instead of "i". The following tasks will be executed after the endpoint is requested:

- Create a temporary file.
- Write the request message to the file and close the file.
- Open the file, log the first line of the file, and close the file.
- Delete the file.

Compared to the average-case scenario, the wires for the worst-case scenario are modified as highlighted in Listing 4.1. First, the *response-code* wire is set to "203" to overwrite the otherwise sent "200 OK" status code on the */benchmark* endpoint. Secondly, the *http-header* wire is changed to overwrite the "Date" header attribute since the Java implementation has no "Server" attribute per default, which could impact the performance.

Listing 4.1: Average-case honeYAML

```yaml
honeywire:
  kind: response-code
  enabled: true
  name: status-code-admin-path
  description: Returns @value instead of
      the original status code when @path
       is requested
  operations:
    - op: replace-status-code
      value: 200
      condition:
        - path: /admin
---
honeywire:
  kind: http-header
  enabled: true
  name: http-header-server-replace
  description: Changes the Server
      attribute of @key to @value if the
      key exists.
  operations:
    - op: replace-inplace
      key: Server
      value: "Apache/1.0.3"
```

Listing 4.2: Worst-case honeYAML

```yaml
honeywire:
  kind: response-code
  enabled: true
  name: status-code-admin-path
  description: Returns @value instead of
      the original status code when @path
       is requested
  operations:
    - op: replace-status-code
      value: 203
      condition:
        - path: /benchmark
---
honeywire:
  kind: http-header
  enabled: true
  name: http-header-server-replace
  description:Changes the Server attribute
       of @key to @value if the key
      exists.
  operations:
    - op: replace-inplace
      key: Date
      value: "Sun,␣01␣Jan␣2023␣11:55:00␣
          GMT"
```

4.4.4 Server Environments

For the server environment, two different hardware systems are chosen:

- Amazon Web Services (AWS)
  with a t3.medium EC2 instance.
- Virtual Machine (VM)
  running on-premises with four logical cores.

We proposed our PoC for the cloud environment and used Kubernetes for the deployment. Kubernetes was chosen as the technology since it is the most popular platform for cloud-native workloads. Amazon Elastic Kubernetes Service (EKS), the managed implementation of Kubernetes by AWS, was chosen because there were already existing resources and prior knowledge, resulting in faster development of the PoC. Considering the hardware resources, we chose two t3.medium instances of Amazon's Elastic Compute Cloud (EC2). Those instances have two virtual cores with 4 GB of RAM. Due to the way EKS handles networking, those instances are capable of running a total of 17 pods[11] per instance. This is important since the deployment needs at least 14 pods for the TeaStore benchmark. Seven pods are used for running TeaStore itself (i.e., auth, db, image, persistence, registry, webui, and recommender service), in addition to two used for the test bench and the deception operator and five general purposed pods necessary for running the node (aws-node, CoreDNS, kube-proxy, efs-csi-controller and efs-csi-node). We chose EFS to provide the PoC resources since it enables access from different EC2 instances to enable centralized management of the honeYAML file. Additionally, it is a file system capable of using locks on files, which the PoC leverages to prevent race conditions while updating the honeYAML file.

Since cloud environments have higher variance [32], the benchmarks are also run at an on-premises virtual machine (VM). The VM is managed by VSphere[12] and has the following characteristics: Intel Xeon E5-2680 v3 with 4 out of the 12 cores used, 64GB RAM, 100 with Ubuntu 22.04.3 LTS. Initially, we used Docker for running the benchmarks. However, during the evaluation phase, we replaced Docker with the local Kubernetes cluster Kind[13] version 0.20.0 to reduce HTTP traffic noise, which is described in more detail in Section 7.1. An SSH connection was established to connect and run the tests, and port 8089 was forwarded to access the Test Bench from the local PC.

4.5 Threats to Validity

This section presents the threats to validity guided by the validity categories of Wohlin et al. [33]. In more detail, the following subsections discuss construct, internal and external validities.

4.5.1 Construct Validity

Construct validity addresses whether the conducted tests answer the research questions appropriately. The PoC was designed to give insight into the performance of the LD_PRELOAD prototype for deceptive purposes. For this scenario, we chose two types of wires that overwrite HTTP headers to represent a deceptive task. This representation resembles a limited part of possible deceptive techniques. Different techniques might

[11]`https://github.com/awslabs/amazon-eks-ami/blob/master/files/eni-max-pods.txt`
[12]`https://www.vmware.com/products/vsphere.html`
[13]`https://kind.sigs.k8s.io/`

need more or less processing resources and memory, which can significantly impact the measured performance overhead.

Considering the benchmark, we ran our setup within a single node on a VM and AWS. In order to better generalise our statement, the benchmarks should be performed on multiple nodes, repeated more than once and performed on further cloud environments as stated by Papadopoulos et al. [32]. Especially, running benchmarks on one provider does not apply to other providers, as shown by Leitner and Cito [34].

4.5.2 Internal Validity

The internal validity addresses concerns about the conclusion being affected by factors other than the study. The length of each of our 24 benchmarks ranges from about five to about 37 minutes, and the AWS tests were conducted within a day, whereas the VM tests were conducted the following day. Therefore, we can not eliminate possible daily patterns as detected by Iosup et al. [35]. Even though newer studies could not replicate those patterns, they still showed that the performance of EC2 instances, as we use them, are highly variable [34]. Different performances depending on the EC2 instance could also impact the relative performance overhead of our prototype.

4.5.3 External Validity

The external validity addresses concerns to which point the findings of this thesis can be generalised. We considered two use cases with two different endpoints each to extend the generalizability of our result. Nevertheless, these endpoints had an overall simple functionality, so the result can not be applied to more complex endpoints. Additionally, we mainly tested on Java Spring applications, with only one endpoint being on a Python Flask endpoint. Therefore, our results might differentiate from other languages and libraries.

CHAPTER 5

Tracing with Libc

This chapter focuses on resolving the question, "How can socket communication in Linux be traced by injecting code with shared libraries?" (RQ1). For this, Section 5.1 introduces the first steps of building a shared library with the concept of tracing in mind. After the basics of the shared library construction are created, the focus will be to analyse the communication between a Java application and the shared libraries in Section 5.2. Similarly, Section 5.3 examines a Python application. Section 5.4 then draws a conclusion and outlines the support for tracing languages other than Java and Python.

5.1 Tracing with Shared Methods

We have to know which specific methods relate to an HTTP connection before we can hook those methods. Despite the socket man page presenting a good overview of a method's functionality, it does not specify which languages use it. Therefore, we discuss how the tool strace can be utilized to identify the shared library methods that correlate with HTTP requests. The second subsection describes how we created a small shared library based on the strace results to track the HTTP request throughout the methods.

5.1.1 Strace

The OS tool strace[1] can be used for monitoring system calls of specified processes. As an example, a Test.java file was created that implemented a simple "Hello World" REST endpoint with the Java net.httpserver[2] library. The command to start and trace the application looks as follows: `strace -f -o strace_log.txt Java Test.java`. Option -f is for following forks (typically occurs for every new incoming request), and option -o FILE to log the traces into a file instead of stdout. After starting the application,

[1] `https://linux.die.net/man/1/strace`
[2] `https://github.com/openjdk/jdk/blob/jdk-17%2B7/src/jdk.httpserver/share/classes/com/sun/net/httpserver/HttpServer.java`

we sent a request to the endpoint and stopped the application again. In this case, the resulting log file had about 37,000 lines. We summarized the most important traces in Listing 5.1. To find those traces, we started by searching for distinct keywords. The keyword "HTTP" is a good start, appearing only in four lines out of the 37,000. The first line was related to a meta information check on the net.httpserver library. The second line was naming the process or threat name, and the last two results were the actual read (line 35292) and write (line 36487) call resulting from the HTTP request. As seen in lines 35290-35292, the original call of *read* spans over two separated loglines with a *gettid* call in between. The *read* as well as the *gettid* are unfinished, meaning that the call itself takes some time in which another process or threat is called a method. The first *read* logline mentions in its first argument FD 7. This FD has to be created at some point with an accept call, and with that, one can track back the connection to its creation. By searching with the keyword "accept" and looking for the response of the *accept* methods to be 7, logline 35256 was found. As with the transition from the read and write calls to accept, one can track additional methods with the first accept parameter. This parameter is the original socket FD, and the methods involved are socket, bind, listen, and getsockname. With that in mind, one could gain additional information about a connection. For example, the used port 8080 of the application is mentioned within the bind attributes.

Listing 5.1: Log investigation with strace

```
// Line number out of about 37000 strace loglines was added manually.
       Then strace logged process ID, function name, function
       parameter and return value.
30787: 18333 socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 5
...
31175: 18333 bind(5, {sa_family=AF_INET, sin_port=htons(8080),
       sin_addr=inet_addr("0.0.0.0")}, 16 <unfinished ...>
...
31183: 18333 listen(5, 50 <unfinished ...>
31187: 18333 <... listen resumed>)              = 0
...
31199: 18333 getsockname(5, <unfinished ...>
31203: 18333 <... getsockname resumed>{sa_family=AF_INET,
             sin_port=htons(8081), sin_addr=inet_addr("0.0.0.0")},
             [28->16]) = 0
...
35256: 18422 accept(5, {sa_family=AF_INET, sin_port=htons(34084
       ), sin_addr=inet_addr("127.0.0.1")}, [28->16]) = 7
...
35290: 18422 read(7, <unfinished ...>
35291: 18346 gettid( <unfinished ...>
35292: 18422 <... read resumed>"GET / HTTP/1.1\r\nUser-Agent:
       Post"..., 8192) = 201
...
36487: 18422 write(7, "HTTP/1.1 200 OK\r\nDate: Mon, 21 A"...,
       76 <unfinished ...>
36494: 18422 <... write resumed>)               = 76
...
36502: 18422 write(7, "Hello world! <br> This is a test"...,
       38 <unfinished ...>
36510: 18422 <... write resumed>)               = 38
```

5.1.2 Shared Library

After the methods used by the technology were found with strace, a shared library can be created for tracing specific HTTP requests. If one considered a simple approach for logging as explained in the method hooking background, one would generate much noise in the form of unimportant loglines. This might still be sufficient for simple Java applications. However, when using frameworks like Spring Boot[3] for Java or Flask[4] for Python, the loglines can increase dramatically to a few thousand just for starting the application. As one can imagine, this can dramatically decrease performance. In the case of a spring boot application we tested, this resulted in a startup time increase from initially ∼6 seconds without LD_PRELOAD to ∼50 seconds with LD_PRELOAD, making debugging sessions unfeasible. The main reasons for the noise are the read and write method calls used for file interaction, like reading ELF files.

An uncomplicated solution to minimize noise is preserving a context generated by different shared methods inside global variables. This context can then be checked when different shared methods are called to decide if the method call is useful or just noise. The preserved context used within this thesis can be split into the following methods:

- **__libc_start_main()**: This method will always be called before a process starts. The main objective is to initialize any necessary things for the execution environment. An example of such a thing can be initialize threading, call main methods, or the exit method after the main method is finished. Therefore, this method is ideal to initialize the resources needed for a deception system. In addition, this method's parameters include the argv arguments of the CLI command. This means one can filter out for different tools, for example, comparing if the first argument (argv[0]) is "java" or "python".
- **bind()**: The bind method assigns an address to a socket referred to by an FD. This behaviour can be used to mark a specific FD when the application binds a particular address, such as a specific IP address, protocol or port. Based on that, the accept method then knows if a new connection is relevant based on whether the bind method marks the FD.
- **accept()**: Within the accept method, a new connection will initialized for an incoming request. This includes creating a new socket and a new file descriptor. The accept method can be used to mark specific TCP connections by saving newly created FD. With the parameters of the accept method, additional information related to the connection type can be tracked, for example, the address families like AF_INET (IPv4) and AF_INET6 (IPv6).
- **read()**: With the read method, the application can read the information a client sends, for example, the request header. Therefore, information about parameters

[3] https://spring.io/projects/spring-boot
[4] https://flask.palletsprojects.com/en/2.3.x/

like the requested endpoint or the request method (like GET or POST) can be acquired.

- **write()**: The write method is used on the application side to answer the client. The write method contains the response header and body from the application.
- **close()**: When an FD is closed, the close method will be called either from the client or application side. If information about the FD is saved inside the global variables, this method can be used to free those resources to mitigate false positives inside the log data.

5.2 Java Code Analysis

As we described in the previous section, our trace analysis resulted in a ldpreload.so a prototype that overwrites specified shared methods to log certain aspects throughout a TCP connection. We specified "java" as the second parameter apart from setting the absolute path of our generated ldpreload.so file as the first parameter of LD_PRELOAD. This enabled us only to load the shared library if we started a Java debugging session. Hence, we did not make use of the ___libc_start_main() to filter out processes that are unrelated to our analysis.

When starting the net.httpserver application and making a request with curl, the following logs are generated:

$$bind(4) : 0 \rightarrow accept(4) : 8 \rightarrow read(8) : 201 \rightarrow write(8) : 76 \rightarrow \\ write(8) : 38 \rightarrow read(8) : 0 \rightarrow close(8) : 0$$

The first word is the method name, followed by the traced FD within the brackets. The value after the colon is the return value of the original shared method. With the →, we depict that some time passes until the following log line is generated. For the *bind* and *close* methods, the return value is either zero or one, depending on whether the method succeeded. The *accept* returns the FD of a new connection. Lastly, the *read* and *write* methods return the bytes read or written.

As we expected, the accept method opens the FD 8 based on the previously bound FD 4. Afterwards, the request exchange happens with the read() and write() methods. The double-write operation is due to the nature of net.httpserver has separate commands for sending the header and response body. The return values of the read() and write() methods are the bytes received and sent, respectively. What stands out is the read operation with 0 bytes read before the FD got closed. If the request was sent from a browser, this read() call would not appear anymore.

Java uses the Java Native Interface (JNI) to enable the Java Virtual Machine (JVM) to communicate with native methods like socket operation. The net.httpserver library, for example, uses Net[5] class to call the native bind method (line 555) and the accept method (line 592) through JNI. We found that the UnixNativeDispatcher[6] class is used to call the native read (line 465) and write (line 470) method. We used these methods as a starting point for investigation.

Considering the zero bytes read by the read(8):0 trace, we debugged the read call further, checked for error flags and used an additional log line of attributes within the shared library. We found that the end-of-file (EOF) flag was set inside the JDK[7] resulting in the connection being closed later on[8]. This is expected behaviour if the client shuts down the connection [36].

When we analysed a Spring Boot application running with Tomcat, the findings about the logs were similar, except only one write() call occurs. There is only one write() call due to the response header and body being sent together. Even through Spring boot as well as net.httpserver using the same IOUtil.java class for the connection, the Tomcat server passes a SocketDispatcher[9] class instead of UnixNativeDispatcher, which uses the shared libraries differently.

We also tried a request from a Chromium-based browser, and the result stayed the same, except the second read() had a return value of minus one. We analysed the reason for that by printing the errno number that appeared within the shared library, and it showed an "EAGAIN or EWOULDBLOCK" according to the read() man page. This is another way a socket connection can be terminated, as analysed by Zhuang et al. [37].

This finding concludes that different traces might be seen even for the same language since the programmers decide what and how the shared methods are called. Most calls will eventually be called over similar libraries and, therefore, offer similar call behaviour, but a programmer is not forbidden to implement the socket connection themselves. Theoretically, all permutations of the available socket methods and their call order are possible.

5.3 Python Code Analysis

We analysed the Python application in the same way as described in the previous section and identified that Python with Flask calls different methods for communicating over sockets:

[5]https://github.com/openjdk/jdk/blob/jdk-17%2B8/src/java.base/share/classes/sun/nio/ch/Net.java

[6]https://github.com/openjdk/jdk/blob/jdk-17%2B8/src/java.base/unix/classes/sun/nio/fs/UnixNativeDispatcher.java

[7]https://github.com/openjdk/jdk/blob/jdk-17%2B8/src/jdk.httpserver/share/classes/sun/net/httpserver/Request.java#L288

[8]https://github.com/openjdk/jdk/blob/jdk-17%2B8/src/jdk.httpserver/share/classes/sun/net/httpserver/ServerImpl.java#L669

[9]https://github.com/openjdk/jdk/blob/jdk-17%2B8/src/java.base/unix/classes/sun/nio/ch/SocketDispatcher.java

$$bind(4) : 0 \rightarrow accept4(4) : 5 \rightarrow recv(5) : 77 \rightarrow send(5) : 174 \rightarrow send(5) : 38 \rightarrow recv(5) : 0 \rightarrow close(5) : 0$$

Through logging, we noticed that Python uses accept4() instead of accept(), recv() instead of read() and send() instead of write(). The difference between those three cases is that one method has an additional flag attribute that can be set to adjust the method behaviour for some cases. From the logical side, Python's recv() and send() method calls behave in the same way the net.httpserver calls do. They send the header and body of the response separated and receive a zero return value from the recv() method before the connection is closed.

During our analysis, we found the following:

- The socketserver.py file[10] calls the bind() method in line 466, the _accept() method in line 499 and the sendall() method in line 826.
- The server.py file[11] calls the readline() in line 400.
- The mentioned calls do not directly call the libc method but instead call the Python implementation of the shared methods. In case of accept() call from socketserver.py the _accept() from socketmodule.c file[12] will be called. The _accept() then specifies in in line 4909 that an _accept() call from python will be directed to the sock_accept() which then calls the sock_accept_impl() (line 2729) to decide then based on a preprocessor directive (line 2685) if the accept() or the accept4() original method of libc is called.
- Similar to the accept() method, the sendall() method gets resolved into one or more libc send() calls and the readline() method is resolved to libc recv() calls[13] that we traced with our ldpreload.so.

5.4 Discussion and Outlook

For addressing RQ1, we discuss the findings of the Java and Python analysis, as summarised in Figure 5.1 and go into the implication of each of the shared methods. Afterwards, we will give an outlook on how those findings can be applied to other languages.

No matter the language or tool used for starting an application, the **___libc_start_main** method will always be called. Based on the CLI arguments of this method, it is possible to identify the language or tool of the starting process. This can be useful to set functionality accordingly.

[10] `https://github.com/python/cpython/blob/v3.10.6/Lib/socketserver.py`
[11] `https://github.com/python/cpython/blob/v3.10.6/Lib/http/server.py`
[12] `https://github.com/python/cpython/blob/v3.10.6/Modules/socketmodule.c`
[13] `https://docs.python.org/3.10/library/socketserver.html?highlight=socketserver#module-socketserver`

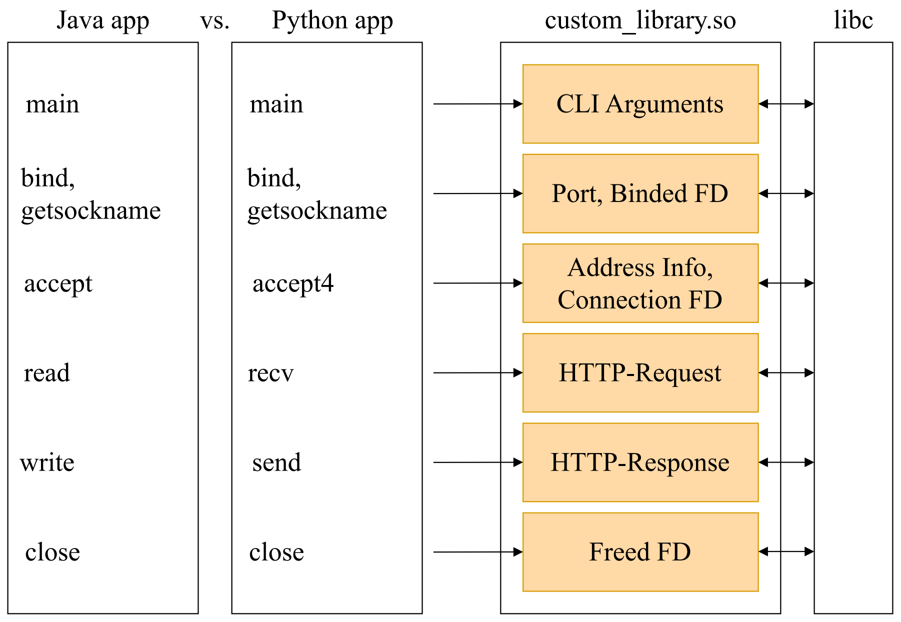


Figure 5.1: The called method from the Java and Python application, respectfully. A custom library (custom_library.so) in between can trace and manipulate parameters depending on the overwritten method.

Example scenario: One wants to create a custom shared library to trace arbitrary applications. For this, the Python application would need a different implementation than a Java application. Both implementations can be compiled to a shared library, and the ___libc_start_main method then decides during runtime which implementation should be used based on the CLI arguments.

The **bind** or **getsockname** as well as the **close** method are used by the Java and the Python application. Since the man page does not specify alternative functions for those methods, we assume that those functions will be used for all languages.

According to the man page, there are two ways of accepting an incoming request: with the **accept** or the **accept4** method. The difference between those two methods is that accept4 has an additional flag attribute. If the flags attribute is zero, the accept4 behaves the same as an accept call. Hence, all accept calls can be forwarded to the accept4 implementation within the ldpreload.so so that only the accept4 method needs to be maintained.

The main obstacles when tracing multiple languages are the requests and responses. Java application uses the **read and write**, whereas Python application uses **recv and send**. When looking at the man page, there are several other possible methods like recvmmsg, sendmmsg or sendfile. Those functions have different arguments and different behaviour, making support and maintenance for supporting multiple languages harder. In addition, no standardised protocol exists that requires specific methods. For example, when trying to inject deceptive tokens into a socket response, the response can be split and sent into multiple write calls, but it does not have to be.

This concludes how tracing socket connection can be performed. When creating a shared library, we suggest starting by supporting one specific application and iterative adapting the shared library to new applications or languages afterwards. We advise generating extensive tests to check if a language is fully supported with major server technologies and language libraries.

CHAPTER 6

Deception System Orchestration

After we outlined the hooking options with LD_PRELOAD in Chapter 5, we created a PoC which is publicly available at GitHub[1]. This chapter introduces the main architectural options encountered during development to answer RQ2: "What are architectural trade-offs when designing a deception system with LD_PRELOAD?"

Beginning with Section 6.1, an introduction is presented on how the LD_PRELOAD system can be automatically injected into arbitrary containers by utilizing a Kubernetes operator. After this, Section 6.2 will point out different options for updating the state of our PoC with a Kubernetes deployment in mind. Lastly, Section 6.3 will apply the architectural pattern from Section 6.1 and Section 6.2 onto the shared methods identified in Chapter 5. This section explains in detail what assumptions we made for the PoC and which architecture we chose with the performance of the shared library in mind.

6.1 Cloud Adaptation of LD_PRELOAD

For this subsection, we introduce options to distribute a custom-made shared library throughout Kubernetes. We name the shared library deception.so and we explain the technical details in Section 6.3.

To offer an LD_PRELOAD system across a cluster, each container that should be deceptive needs to have the deception.so file mounted and the LD_PRELOAD environment variable set. Due to the nature of the loading process of shared libraries, this must be done before a container's main process has started. To solve this problem, the Kubernetes deployment (e.g., YAML file) configuration or the container image (e.g., Dockerfile) configuration can be adapted. The latter would include that the image's configuration and source are available so it can be rebuilt. Additionally, rebuilding the images can be

[1]https://github.com/dynatrace-research/ld-preload-deception

a complex task. One needs to create and maintain multiple build registries, depending on how many different images and LD_PRELOAD versions one needs, and this does not even include the performance and time resources needed for rebuilding the images.

Instead of manipulating the image itself, changing the deployment can be easily achieved with Kubernetes operators. Operators can automate tasks within Kubernetes by using custom resources. One of those resource types is a mutating webhook, which can modify a deployment configuration before it is applied in a cluster. The mutating webhook is a service running on a container that processes configuration over a specified endpoint and, hence, can be arbitrarily programmed. In the case of an LD_PRELOAD system, the webhook can be programmed to set the environment variable and mount the deception.so. Further, the webhook gets the original YAML configuration and can apply the deception system only in a specified area. For example, the area can be a namespace or a specific label on a pod or container.

We depicted a simplified overview of the cluster architecture with the mentioned operator in place in Figure 6.1. The Figure contains the following parts that must be deployed before an arbitrary container can be made deceptive:

- Operator consisting of:
  - Mutating webhook configuration.
  - Container processing mutating admission webhooks.
- Static or Dynamic provisioned files.
  Note: Dynamic provisioning needs the operator to manage the files.
  - Persistent volume (PV) and Persistent volume claim (PVC) for the containers.
  - The operator (only needed if the operator actively manages the provisioning, which is necessary if the provisioning is dynamic).

The first part of the operator is the **mutating webhook configuration**, and this configuration is deployed so the Kubernetes control plane sends mutating admission webhooks before creating any resource. Apart from the configuration specifying the endpoint and the API that should be called, it also specifies the scope of when to call it. The scope is defined based on rules and other matching options with which one can trigger the webhook on specific conditions. For example, setting the rules "operations" to "CREATE" and "resources" to "pods" will only call the webhook if a prod gets created. With that, rules are elementary for an LD_PRELOAD use case since setting environment variables or mounting files is not supported by every Kubernetes resource. Hence, some resources would not be created at all. Optionally matching options like "objectSelector", "namespaceSelector", and "matchConditions" can further restrict the scope based on things like namespace, resource labels, or deployment parameters. This can be useful for a deception system when the exact scope is well-known and does not change often.

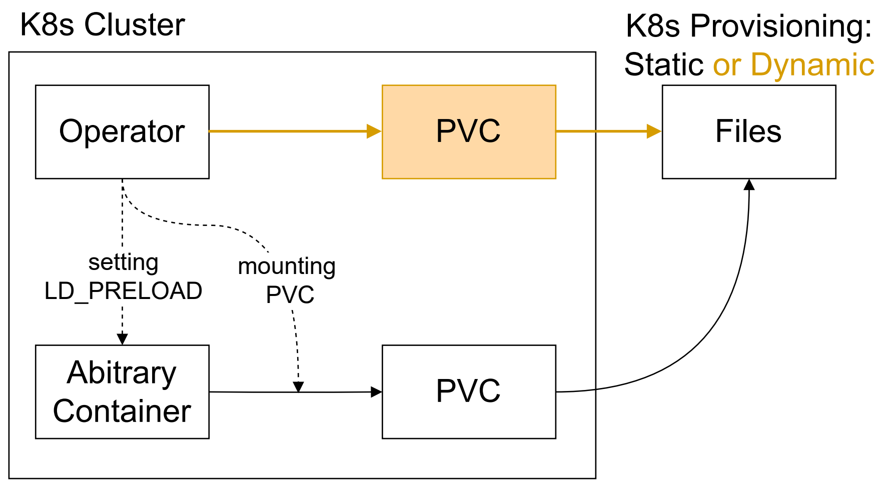


Figure 6.1: Operator including the deception system before the container gets created. Additionally, the Operator has the potential to manage the provisioning when provided with a PVC (orange).

The second part for the operator is the **container processing mutating admission webhooks**. This container can be deployed with a pod or a deployment configuration and, due to its flexibility, is taking a central role in managing the deception system.
Contrary to the mutating webhook configuration, there are no predefined attributes for restricting the scope. Instead, the container is a simple API service created with an arbitrary language, and the scope definition is more flexible by being defined programmatically. The container can also expose an endpoint for editing attributes used for narrowing the scope. With that, a scope change is possible without redeploying the operator. Nevertheless, this would only apply to newly created resources, and already existing ones would need to be redeployed.
A simple implementation of the API service is processing the deployment of an incoming webhook request by adding the LD_PRELOAD variable and mounting the deception.so with the PVC. An advanced system can differentiate between incoming deployments and mount different deception.so explicitly designed for a particular deployment. If the API service connects with a PVC to the provisioning itself, it could also be recompiling the deception.so, as elaborated in the next section.

Lastly, the **provisioned files** have to be accessible for the cluster so that the operator can mount them. This can either be done statically or dynamically. Compared to static provisioning, dynamic provisioning further automates the deployment by having more complexity within the operator. Hence, the dynamic approach is more time-consuming upfront.
When provisioned statically, the cluster administrator has to allocate PV and manually copy the files needed for the deception to the provisioning. The operator can then reference the provisioning with a PVC matching the exact PV. The provisioning can

also be dynamic to minimize the manual steps further, meaning that a PVC can claim a certain space if it stays within a specified memory size. For the dynamic allocation, the operator needs to have the deceptive file locally to copy it to the dynamically provisioned folder. Therefore, the operator would need its own PVC to access the provisioned folder. Independent of the kind of provisioning, it is important to consider the access mode supported by the underlying provisioning service. The access modes limit access to only one pod, one node, or multiple nodes, depending on where and how many containers must be served. Therefore, one has to check beforehand which access modes are supported by the provisioning service. For this reason, we selected Amazon EFS since the *ReadWriteMany* access mode to access a resource from multiple nodes at once is available.

6.2 External Systems State Updates

Since LD_PRELAOD is loaded before the process starts, updating the state while the process runs is not straightforward. Therefore, we elaborate on how the state can be managed within the deception.so and what needs to be changed during runtime within Subsection 6.2.1. Subsection 6.2.2 then considers an alternative approach by relaxing the constraint of runtime updates and letting the operator manage the deception system.

6.2.1 Internal And External State Management

We developed a state management system to allow external changes during the runtime. The different parts of the architecture within the deception are depicted in Figure 6.2 and will be explained in this subsection.

Starting with the main **hook** (abbreviated for ___libc_start_main), the functionality of the deception.so gets triggered. Here, we differentiate the main method from the other methods mentioned in Chapter 5 due to it being the only method that is called exactly once and before each other call. Therefore, the main hook can be used to initialize resources that are later needed. The hook of the other methods can read things like the wire specification from the global state to deceive its parameter accordingly. Besides that, both hook types call the original shared library method from libc and return the value to the application.

When we talk about the **global state**, we understand the global variable within a process. They play a central role in externally updating the deception. It is important to note that we split the global state into three different categories:

- During before or at compilation:
  Either using fixed compiled C macros[2] or setting constant global variables.
- During runtime by internal methods:
  The state will be modified and used by the shared methods but is not modifiable

[2]`https://gcc.gnu.org/onlinedocs/cpp/Macros.html`

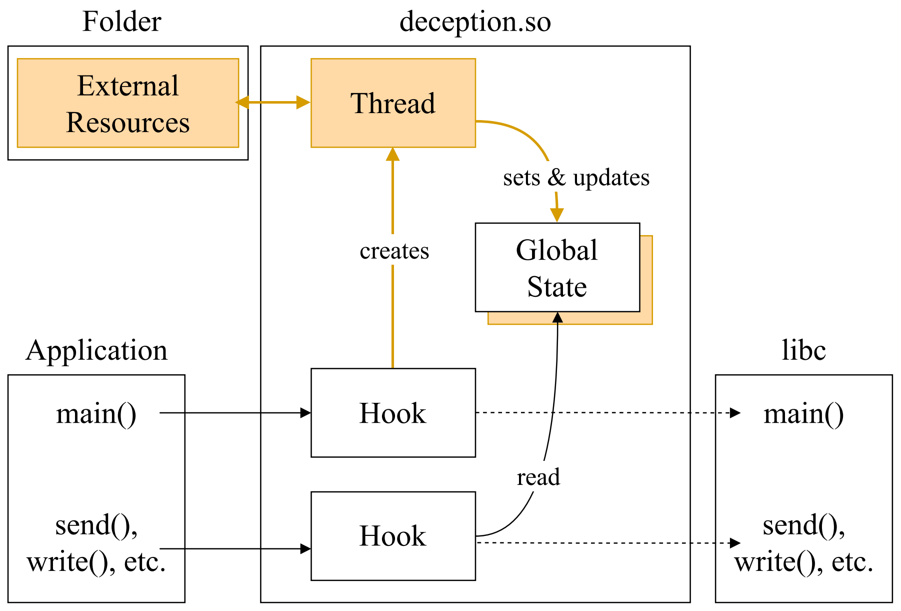


Figure 6.2: State management within deception.so. Compiled made modifiable during runtime (orange)

externally. For example, the methods save specific file descriptors (FD) to trace HTTP connections.

- During runtime with external resources:
  The state is externally modifiable during runtime by using a thread within the shared library. The shared method themselves will only read the state but not update it.

Since most shared methods are invocated from the application, updating an external state within those methods would greatly reduce performance due to the synchronousness of the operation and the potential parallelization when methods get called within different threads. To overcome those challenges, we created a single **thread** to manage the externally modifiable part of the global state. Since the main hook will be called for each process on a system, it is important to put an appropriate filter in place so that the thread gets only generated for processes that should be deceived.
Depending on the external resources, the thread can use a pull or a push-based communication channel. Apart from ensuring that no race condition is introduced by updating the global state, the thread might also have to lock out for race conditions introduced while reading from external resources.

External resources can be integrated in various ways that strongly impact the complexity of the resource as well as the thread. An example of a simple integration can be a text file that describes the state of attributes. The file can be mounted by the operator together with the deception.so. The thread can then periodically read the configuration from the local system and update the internal state. A more complex solution might use an external API accessible over TCP, which, in contrast, would imply that all the hooks can exclude this connection from being deceptive.

6.2.2 Recompilation

One place that was not considered until now is how to maintain and update the deception system, especially with offering the deception as a SaaS solution in mind. Therefore, this subsection discusses how maintenance and updating the system can be achieved by enhancing the operator's capabilities, as mentioned briefly in Section 6.1.

In contrast to SaaS solutions offerings like office tools or cloud storage, part of the deception cannot be offered over a website. For example, the operator or the deception must be deployed inside the consumer's infrastructure. Therefore, a differentiation must be made on what parts can be managed by the cloud service provider (CSP) and what parts must be available on the consumer side.

In the case of the operator, publishing the mutating webhook configuration and redeploying the container is inevitable. However, the CSP can offer the container image prebuilt over an image repository, reducing consumer expenses. The process of automatically updating based on an image repository is also supported by Kubernetes tools like Argo CD[3], further decreasing the maintenance on the consumer side.

For the provisioned resources like the deception.so, maintenance depends on whether the operator manages the files. For example, when static provisioning is chosen and the operator does not manage the deception.so, the system administrator must upload and update the deception.so manually. If the operator can access the resources, it can again be programmatically enhanced to update them. Therefore, the resources either already exist in the operator container image or the operator connects to an external API to derive the resources. The latter enables the CSP to compile the deception.so and send the operator the files to initialize or update the resources on the consumer side. With that, a consumer only needs to install an operator within a cluster to get deception as a service.

Updating the deception.so in an already existing process is still impossible due to LD_PRELOAD loading the files before the process starts. In this case, the operator can again be enhanced with privileges to modify the cluster and use the Kubernetes function to gracefully terminate and recreate all pods. Not redeploying pods without the deception system would require the operator to keep track of all the relevant deployments, but the operator can track these deployments. The operator must look at the incoming request sent from the Kubernetes control plane and conduct a lookup for the deployment that initiated the pod deployment with the pod configuration sent with the request.

By relaxing the constraint of runtime update, the thread within the deception.so could be removed as a hole, and the operator could process the external resources. If a change

[3]`https://argoproj.github.io/cd/`

occurs, the operator will issue a recompilation through the API of the CSP or recompile the deception.so itself if provided with the source code. Afterward, the operator would roll out the changes without the consumer having to interact with the system.
Choosing recompilation and redeploying instead of having a dedicated thread for checking updates would have the drawback that the deception.so could not be updated externally. At the same time, a potentially malicious TCP connection is active, but it would still be able to react to a multi-step attack [38]. A multi-step attack is an attack that spans different actions and interactions with a system to eventually find and use one or multiple vulnerabilities in those systems. By letting the operator compile different deceptive elements, LD_PRELOAD can create a deception framework like the one proposed by Kahlhofer et al.[19].

6.3 Trade-Offs within LD_PRELOAD Systems

To wrap up RQ2, this section discusses the design decisions for our PoC. The first subsection starts by describing the configuration format used to update the wires. The second subsection builds up on Chapter 5 and explains which filter methods we used in the shared methods. Finally, the last subsection discusses how the PoC preserves the characteristics of the cloud.

6.3.1 HoneYAML Configuration

The wires represent the individual traps placed to deceive the attacker. As shown in Figure 6.1, the decision was made to use a simple YAML definition to represent the wires state as introduced by Kahlhofer et al. [39]. This configuration is called honeYAML.

Listing 6.1: HoneYAML example that overwrites the "Server" response header and the status code when the /admin path is called.

```
honeywire:
  kind: response-code
  enabled: true
  name: status-code-admin-path
  description: Returns @value instead of the original status code when
       @path is requested
  operations:
    - op: replace-status-code
      value: 200
      condition:
        - path: /admin
---
honeywire:
  kind: http-header
  enabled: true
  name: http-header-server-replace
  description: Changes the Server attribute of @key to @value if the
      key exists.
  operations:
    - op: replace-inplace
      key: Server
      value: "Apache/1.0.3"
```

To update the configuration in the PoC, the consumer must manually change and save the file within the provisioning. We want to mention that the honeYAML can be managed by a service which offers a user interface for the consumer to make it more intuitive. However, this was not done in this thesis since it does not impact the PoC performance evaluation conducted in the next chapter.

For the wires it is possible to modify the following attributes:

- *kind*: The current PoC supports two different honewire types: "response-code" and "http-header".
- *enabled*: Either true for having the wire active or false if the wire should not be injected.
- *operations: replace-status-code*: The response-code wire can be specified with a value and a path attribute. During an HTTP connection, the HTTP response code will be overwritten by the string set in the value parameter if the requested resource's path contains a substring equal to the specified path parameter. Depending on the defined value attribute, the status code will be adjusted in length.
- *operations: replace-inplace*: The http-header wire offers a in-place string replacement. This means that the length of the original HTTP response attribute will be preserved. If the string provided at the value attribute is shorter, it will get padded with spaces. If the string is longer, it will get truncated.

6.3.2 Global State Management

Based on the findings of Chapter 5, Figure 6.3 was created to explain how the PoC is intercepting incoming requests by hooking the deception.so. The specific method and their usage of the global state are explained as follows:

1. *___libc_start_main*: As the main method will be processed for each process created, it is necessary to heavily filter requests only to run more extensive logic if needed. The string values that the PoC supports are "java", "python", and "python3". Those values are saved as a constant array within the global variables to filter out based on the first array entry within the argv parameter of the method. This input parameter resembles the CLI tool that was executed to start the process. It is fine if the values are just a sub-path of the input parameter (e.g., /usr/lib/java/java) to allow calling the tools by their relative or absolute path. In addition, the string comparison does not need to be case-sensitive. If the first parameter matches one of the supported tools, the PoC will initialize the global state and start the threads for updating the wires. During the initialization phase of the global state, a lookup for all original libc methods is performed, and their references are saved into the state to avoid multiple lookups during process runtime for each incoming or file manipulation.

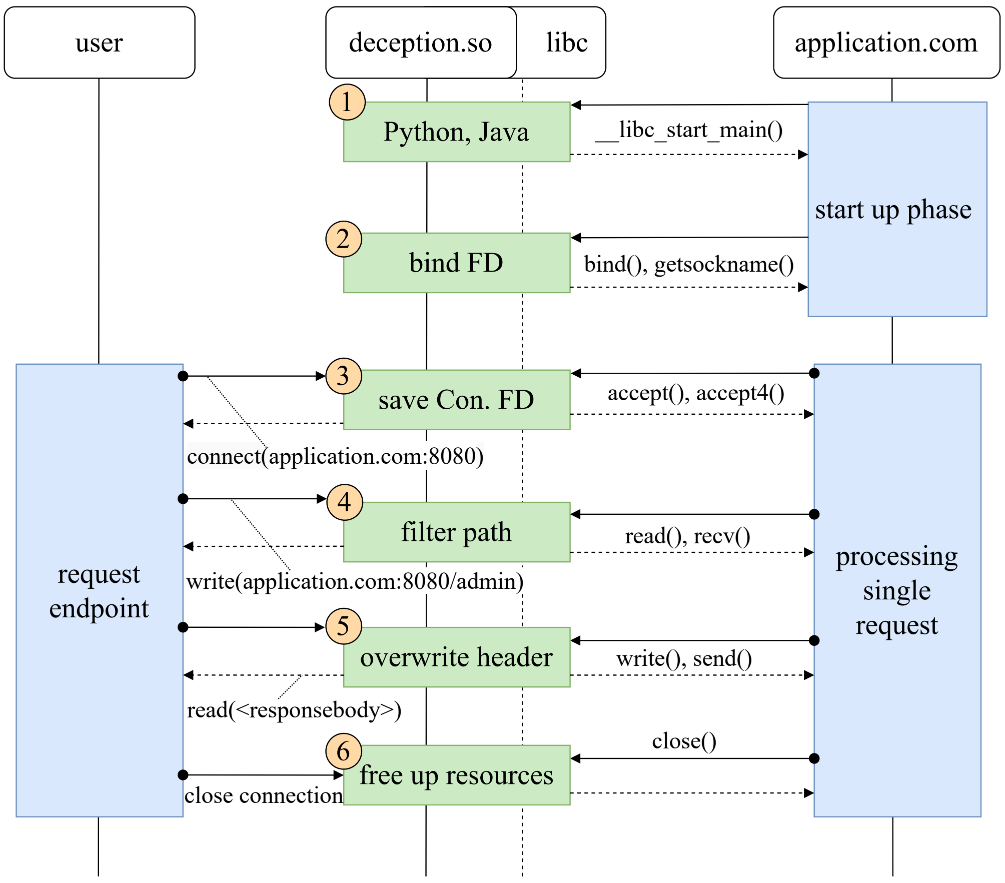


Figure 6.3: Example how an HTTP response can be made deceptive.

The first filter for all other hooks always checks if the global variables are initialized. If not, the hooks will immediately call and return the original libc method, to no further misspend time on checking other filters.

2. *bind* or *getsockname*: When the application starts, the port will be bound to an FD. This behaviour is used to filter out connections that should not be deceived. For this, an assumption was made that the application only serves content worth deceiving over a single port. For the PoC the ports 5000 and 8080 are saved again in a constant array within the global variables. If different ports should be supported, they can simply be added to the array, and after recompiling the PoC, requests based on these new ports can be deceptive. Important to note is that the ports address the port published by the process itself (e.g., targetPort when specified in a Kubernetes container YAML), which can vary from the port that a user accesses due to rerouting through proxies and services.
In addition to filtering out unwanted calls of the method itself, bind or getsockname will also save the FD that can be linked to the traced port within the global state.

This FD is used for the accept method for filtering in the next bullet point.

3. *accept* or *accept4*: As mentioned in the previous bullet point, the accept methods will filter out incoming requests that are not based on a specific port, tagged by the FD via the *bind* or *getsockname* method. Additionally, these methods filter out requests that are not based on IPv4 or IPv6. // Apart from that, this method also saves the newly defined FD for handling the incoming request. An active HTTP connection will get tracked inside the global state within a constant-sized array acting as a hash table to prohibit race conditions during an FD's usage time and optimize performance. The PoC uses the fact that the FDs are unique within a process to just set the value of array[FD] to true. Due to the benchmark being conducted sequentially, an array size of 1000 is sufficient.
The accept methods are the first method mentioned that will be called multiple times through the process lifetime. Therefore, it also checks if the wires inside the global state are currently not updated. If this is true, the global state will track that a method reads the variable to avoid race conditions. Before the methods return, the count will be reduced again. To avoid draining updates, a separate write flag prohibits new read access while an update is pending. The deception will be skipped if the global state is in this update phase. This functionality also holds for the next two bullet points directly below that are exchanging the data.

4. *read* or *recv*: If the incoming FD was tagged within one of the accept methods, the read or recv method checks if the response-code wire is set and the path specified within the wire matches the incoming requested resource. Regardless of whether the path fits or not, an object protocolling the type of the request will be allocated. Besides setting a flag if the path is a match, this object initializes an index needed later. The object is referenced in an additional size constant-sized array at the index of the FD, similar to what is done within the accept method. The object is necessary since it will be used within the read or send method in the next bullet point.

5. *write* or *send*: This method will get the response body as input and can now implant the deceiving elements based on the information the read method tracks. Since the Python implementation tested in Chapter 5 called the send method once for the response header and response body each, the index initialized in the *read* or *recv* method will indicate if the method was called the first time and therefore have to overwrite the header.

6. *close*: The close method will simply free all resources of the FD that are allocated throughout the connection lifetime.

6.3.3 Cloud Adaption

Considering the architecture presented in Section 6.1, the PoC only takes static provisioning without any interaction between the operator and the provisioning. This simplification of the PoC was made since it will not impact the performance of actively deceived requests when the configuration is not changed. The provisioned files consist of the deception.so and the honeyaml.yaml, both mounted to a container. The thread within the deception.so (as depicted in Figure 6.2) then checks periodically if the honeYAML file is updated (i.e., file has new last modified date) and if so, will put a read lock on the file for processing the data.

Since the PoC is proposed for a cloud environment, one must consider how the prototype applies to the cloud characteristics. Those characteristics have to hold when a CSP provides deception as a service, as presented by [8]. For offering deception as a service, it is important to differ to what extent the operator is implemented. In more detail, the characteristics of the PoC are presented in this thesis, as well as the advanced proof of concept (advanced PoC) that can use the operator to update the deception.so. // Beginning with the characteristic **on-demand self-service**, the consumer must not require human interaction on the CSP side but must set up at least the operator independently. After the operator is set up correctly, this characteristic is fulfilled for the current PoC as well as the advanced PoC
For **resource pooling**, one has to consider which part are located at the CSP offering deception as a service. For example, the honeYAML configuration per se can not be shared between multiple consumers. However, the CSP can allocate one persistent storage containing multiple different folders, each containing a single consumer's information. The operator itself has to be allocated by the consumers cluster. However, the recompilation can be outsourced to the CSP. Therefore, the resource pooling would work well by having a single API where an operator of the consumers can request a specific compilation of the deception.so.
Considering **rapid elasticity**, the current PoC has two parts that could generate scalability issues. The first is the operator. Since it is only processing configuration files, this should not generate any bottleneck until a high volume of permanent pod creation occurs. To scale even through that, the operator deployment can scale up with more pods. The second issue that could occur is because of the honeYAML file. It has to be ensured that the provisioning can handle multiple node access. Compared to the current PoC, the advanced PoC would most likely support multiple pods within the operator deployment due to the added processing needs. Apart from that, the advanced PoC is capable of rapid elasticity.
Specific metrics have to be defined to allow the PoC, and the advanced PoC for that matter, to be a **measured service**. For example, the responses are currently producing a local log if they get deceived. Instead of logging it locally, the information could be sent back to the CSP for generating statistics. This characteristic can be fulfilled but would need advanced evaluation from business analysts.
Lastly, the **Broad network access** can be fulfilled by the PoC and the advanced PoC by providing the services at multiple different availability zones of infrastructure provider.

CHAPTER 7

Performance Evaluation

A performance evaluation is presented in this chapter to conclude the evaluation of the capabilities introduced by LD_PRELOAD. This chapter addresses RQ3: "How does the deception system with LD_PRELOAD affect the performance of the original applications?". The setup of the benchmark, including the test bench (TB) and the system under test (SUT) with the different specifications, are outlined in Section 4.4. The artefacts created throughout the performance evaluation can be accessed on GitHub[1].

The rest of this chapter is split into two sections. Section 7.1 discusses adaptions to the TB to mitigate evaluation errors by the first benchmark runs. Afterwards, the benchmark results of the remodelled benchmarks are presented and discussed in Section 7.2.

7.1 Benchmark Design Flaws

We acknowledge that the initial benchmark had several flaws. The detailed analysis for identifying those flaws is presented in Appendix A. This section will shortly present the resulting adaptation.

The original purpose benchmark described in Section 4.4 resulted in the following adaptation:

- Using a local cluster (Kind) instead of Docker.
  Our VM deployment measured significantly higher variance when using Docker. Therefore, we switched to Kind, which stabilised the round-trip time (RTT) and made it better comparable to the AWS deployments since most of the deployment configurations (YAML file) can be used for both types of deployments.

[1]`https://github.com/dynatrace-research/ld-preload-deception/tree/main/benchmark/evaluation`

- SUT are redeployed before each benchmark.
  We saw no warm-up phase on the deployments where w/ wires=f are set. This is because we turned off the wires during runtime at our w/ wires=t deployment, resulting in the SUT already being warmed up.
- After analysing the lag plots, we found that 10,000 requests were insufficient for the warm-up phase. Therefore, we multiplied the warm-up and total request count by four, resulting in 40,000 requests for the warm-up phase and 200,000 total requests.

After each benchmark run we still occasionally encountered latency anomalies within the VM benchmarks. As recommended by [29], we monitored the SUT and could not recognise any limits inside the VM. Therefore, we think the reasons are other VMs running on the same CPU, also known as "noisy neighbours" [40]. Out of five affected benchmarks, the anomalies at VM Python w/o wires and the VM Python w/ wires=t benchmark had a significantly negative performance impact. For this reason we rerun those two benchmarks.

7.2 Results and Discussion

This section will present and interpret the data resulting from the second benchmark as explained in Section 7.1. The results of the 24 benchmarks are summarized in Table 7.1. Based on this table, the relative differences from the configurations w/ wires=f and w/ wires=t to the baseline w/o wires were concluded in Table 7.2. The relative differences were calculated with the formula $(withWires - baseline)/baseline * 100\%$. The median value will be chosen for comparison during the interpretation because it is more stable against outliers than the average value.

As a reference application for benchmarking microservices, we chose TeaStore to represent an average use case. The results of this SUT are discussed in Subsection 7.2.1. Additionally, a custom SUT was created for Java and Python to compare those two languages. These results are discussed in Subsection 7.2.2. A final discussion about the results and what they might bring for future LD_PRELOAD prototypes is given in subsection 7.2.3.

7.2.1 TeaStore

The TeaStore SUT represents a reference system for a cloud environment. With this SUT we want to see how a proof of concept system performs within an "average" microservice system. Figure 7.1 depicts the boxplots of each run.

When looking at the w/ wires=t benchmarks, the overhead is visually distinguishable for all benchmarks. At the relative comparison between the w/ wires=t and w/o wires, it is visible that the /admin use-case introduces more significant overhead (VM: 10.53%, AWS: 7.65%) than the /home/ use-cases (VM: 7.06%, AWS: 3.50%). These are the consequences of the /admin endpoint having near non-processing time since the requested

	min	max	median	average	relative
VM, /admin, w/o wires	1.15	248.51	2.92	3.26	-
VM, /admin, w/ wires=f	1.29	219.00	2.92	3.25	-0.04%
VM, /admin, w/ wires=t	1.43	229.03	3.23	3.58	10.53%
AWS, /admin, w/o wires	0.82	128.34	1.06	1.31	-
AWS, /admin, w/ wires=f	0.82	168.94	1.10	1.35	3.25%
AWS, /admin, w/ wires=t	0.86	141.11	1.14	1.40	7.65%
VM, /home/, w/o wires	4.41	5051.83	8.98	9.61	-
VM, /home/, w/ wires=f	4.29	170.00	9.06	9.71	0.95%
VM, /home/, w/ wires=t	4.28	5156.84	9.61	10.35	7.06%
AWS, /home/, w/o wires	2.51	183.13	3.32	3.89	-
AWS, /home/, w/ wires=f	2.48	200.17	3.29	3.96	-1.08%
AWS, /home/, w/ wires=t	2.59	219.37	3.44	4.12	3.50%
VM, Java, w/o wires	2.24	54.54	3.94	4.20	-
VM, Java, w/ wires=f	2.15	128.17	4.21	4.50	6.82%
VM, Java, w/ wires=t	2.39	5284.45	4.47	4.86	13.60%
AWS, Java, w/o wires	1.09	24.11	1.41	1.54	-
AWS, Java, w/ wires=f	1.09	30.44	1.43	1.55	1.13%
AWS, Java, w/ wires=t	1.14	32.99	1.48	1.62	5.12%
VM, Python, w/o wires	7.26	140.72	10.96	11.32	-
VM, Python, w/ wires=f	6.31	228.54	11.15	11.49	1.68%
VM, Python, w/ wires=t	7.70	138.03	11.37	11.74	3.76%
AWS, Python, w/o wires	2.53	38.73	3.25	3.79	-
AWS, Python, w/ wires=f	2.50	46.68	3.24	3.79	-0.08%
AWS, Python, w/ wires=t	2.59	83.81	3.33	3.85	2.52%

Table 7.1: Results of the 24 benchmarks. The Teastore SUT (*/admin* and */home/* configuration) is also visualized in the form of boxplots by Figure 7.1. The custom SUT (*Java* and *Python*) are depicted in Figure 7.2. Relative overhead (of median) compared to the w/o wires configuration.

	w/o wires	w/ wires=f
Minimum	-1.08%	2.52%
Maximum	6.82%	13.60%
Average	1.58%	6.72%

Table 7.2: Average of the relative performance overhead of w/o wires compared to wires=f and wires=t, as presented in Table 7.1.

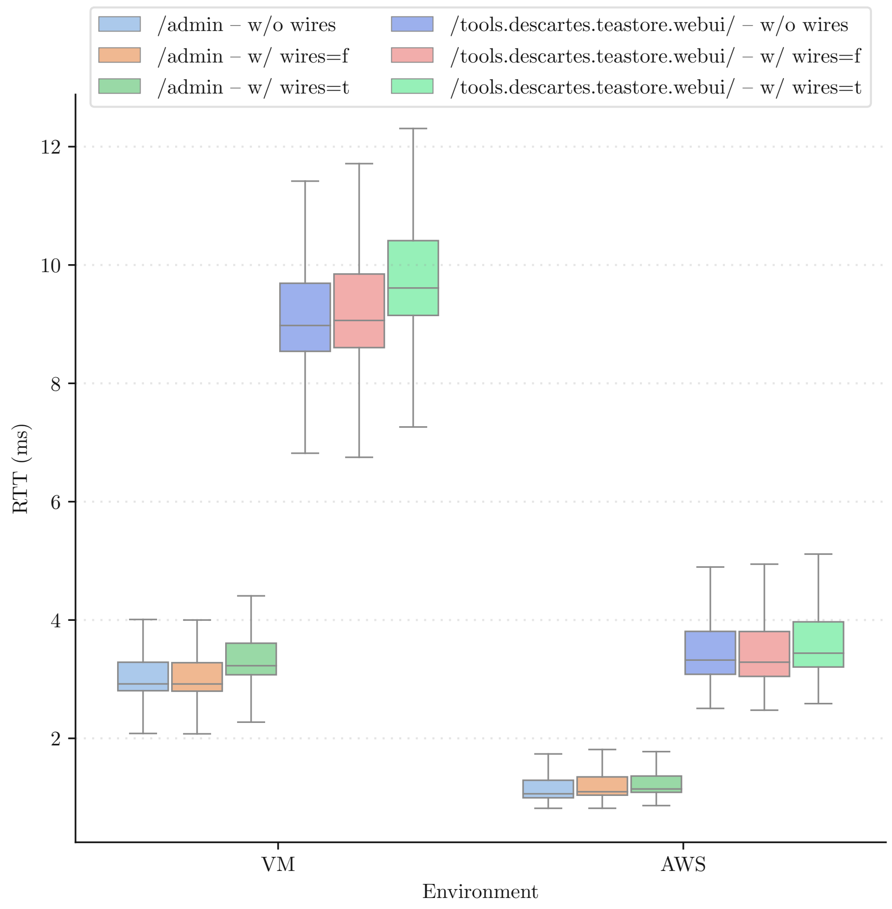


Figure 7.1: Boxplot of the 12 benchmarks conducted with TeaStore as SUT.

endpoint is not implemented. Contrary to that /home/ calls different services and processes their responses. **When designing deceptive wires, one has to consider that the deception system has greater relative overhead at endpoints with less processing time.**

As expected, the w/ wires=f benchmarks introduce near to no overhead compared to w/o wires. **We expected the overad of w/ wires=f is neglectable**, as indicated by the negative numbers for w/ wires=f, like VM /admin with -0.04%.
The benchmark result of AWS /admin w/ wires=f is an exception to our expectations. This benchmark is not visibly different by much in the boxplots. However, when we look at the relative overhead of 3.25%, it is significantly higher than the equivalent VM benchmark, which had only -0.04%. This result arises due to the /admin endpoint having near to no processing time, and the overall processing time in AWS is the smallest out of all our tests. Therefore, the overhead of the deception system is more visible.

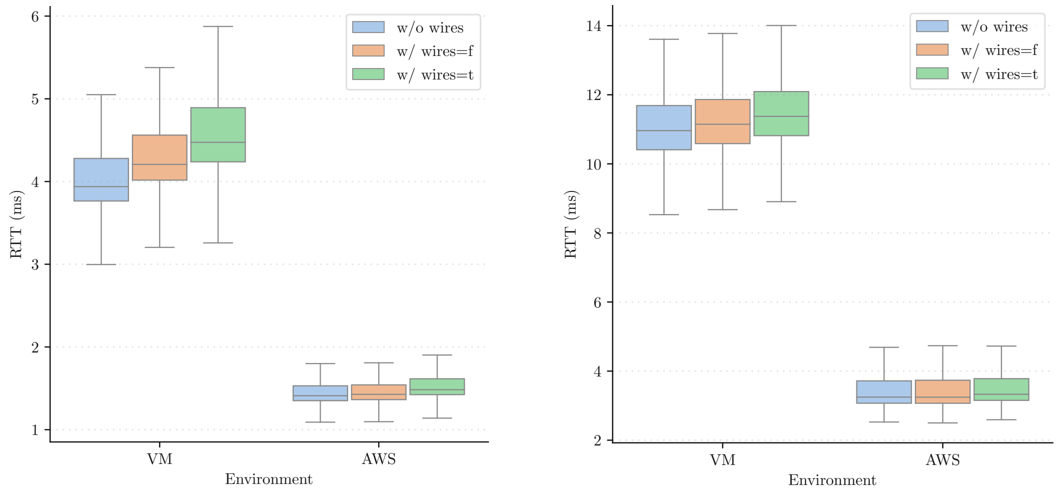


Figure 7.2: The boxplots with the six benchmarks per graphic conducted on the custom-created SUT specifically designed with Java (left) and Python (right).

7.2.2 Custom SUT

The custom test SUTs were created to compare Java and Python within a similar workload setting. To our surprise, the performance for finishing those tasks differs significantly. As seen in 7.2, the Java boxplots (left figure) range from 1 to 6 milliseconds, whereas the Python boxplots (right figure) range from 2 to 14 seconds. We will now go into more detail of the results.

Surprisingly, the VM Java results with 6.82% at w/ wires=f and with 13.60% at wires=t resembles the largest performance overhead out of all benchmarks in their categories. When looking at the Java w/ wires=t configuration, the VM (13.60%) also took over double the relative time compared to the AWS (5.12%). In contrast, the other benchmarks stayed about double the performance or below. Apart from a small performance spike in w/ wires=t, nothing special stood out in the VM Java lag plots analysis. However, we argue that the low maximum number for w/o wires (54.54 milliseconds) compared to w/ wires=f (128.17 milliseconds) and w/ wires=t (5256.59 milliseconds) could be an indicator that the w/o wires had an exceptionally well-going run, whereas the other two configurations had an average to wors run.

Comparing Pythons overhead (VM 3.76%, AWS 2.52%) against Javas overhead (VM 13.60%, AWS 5.12%), the deception significantly impacts the performance of the Java benchmarks more. Considering the results of TeaStore, we argue that the programming language itself does not introduce this difference but rather the Java program, the used framework or the runtime itself through its optimizations. When looking at the median baseline, the Python application took more than double the time to complete the task compared to the Java application. **Overall, we found that endpoints that need less processing time, either through less processed complexity or better optimization, have more impact than the chosen language.**

7.2.3 Final Discussion And Outlook

To conclude the benchmark results, we took the average of performance overheads. The overhead for w/ wires=f is 1.58%, and for w/ wires=t, 6.72% on average. It is worth noting that the performance of the presented PoC can be improved further by setting more parts of the global state during compile-time by recompilation of the PoC as discussed in Subsection 6.2.2. We further acknowledge that performance improvements can be leveraged using appropriate data structures like hash maps instead of arrays. Hence, those numbers do not represent the full potential of our LD_PRELOAD PoC.

Nevertheless, having a 6.72% performance overhead per request can lead to attackers gasping a potential deception being in place and can directly affect the generated costs. For example, when a service is insufficient, additional resources must be allocated. The argument can be brought up that wires are only active for a handful of services. Hence, the overall cost can be minimized. Still, those granular adjustments create additional complexity, leading to a generally high upfront cost to create deception as a service.

Finally, the above results show that LD_PRELOAD can be used for deception within cloud environments and can provide deception as a service. Nevertheless, one has to keep in mind that it is either necessary to optimize the system appropriately or to create a rich feature set to customize which deployment should be made deceptive to make the solution feasible in practice. Both parts create much complexity, so we advise future work to compare the LD_PRELOAD option with other possible options.

CHAPTER 8

Conclusion

This study investigates hooking shared libraries for cyber deception inside cloud environments.

We started with analysing to what extent socket connections can be traced to answer RQ1. We concluded that the major shared method needed for socket communication can be tracked and linked together by one shared library without requiring extensive adaption to support a broad spectrum of applications. Only reading and writing data with a socket can have multiple implementations in libc with small adaptations in their arguments and functionalities, as we saw when comparing a Python with a Java application. Therefore, development with the LD_PRELOAD approach requires iterative advancement for supported technologies and extensive testing.

For RQ2, we developed a proof of concept shared library to inject wires within HTTP-response messages. We encountered multiple trade-offs when designing such PoC with LD_PRELOAD. The main decision one has to consider is the capabilities of the system to update itself versus the amount of performance overhead introduced by the system. The system update itself can be further separated into updating the whole shared library binary by a Kubernetes operator and updates during the runtime, which requires a built-in mechanism within the binary.

Finally, we conducted 24 benchmarks to evaluate the performance overhead introduced by our PoC to answer RQ3. Compared with a baseline system without our PoC, we predicted that the median increase was about 1.58% on average when we injected the PoC and deactivated the wires. However, deactivating the wires means that no deceptive tokens get injected. When we activated the wires, the median increased by 6.72% on average. This concluded that more performance optimisation would be necessary for our PoC.

For future work, we encourage researchers to extend our system further. We want to highlight the following possible enhancements:

- We consider that the next step is to further automate our deception system by extending the operator to build shared libraries as needed. This can also minimize the performance overhead of the wires and the system in general.
- Speaking of wires, we implemented only two types of wires. Therefore, the next step could be to implement additional deceptive wires. Existing vulnerabilities and their exploits can guide the use case for those wires.
- We argue that implementing multiple different wires can also enable managing them automatically. Not only would such automation match with the proposed deception system by Kahlhofer et al. [19], but it also resembles what state-of-the-art deception systems like SODA [8] are already capable of.
- A general evaluation of different layers, like runtime modules for specific languages, kernel patches with eBPF or a proxy solution within the same pod, can help researchers in the future to decide which technologies best fit their use cases.

APPENDIX A

Benchmark Flaws

The initial proposed setup had flaws, considering the low request count and warm-up number, as well as the SUT management throughout testing and the Docker deployment within the VM. Section A.1 will address these flaws occurring in the initial benchmark suite. After that, we rerun the benchmarks and the deficiencies of the second benchmark were corrected. Still, some virtual machine (VM) instances had high maximum response time in their benchmarks. In total, five out of the 24 benchmarks were affected, and we rerun two of them. A detailed description of how external factors influence the benchmarks at the VM and why we rerun those two tests are described in Section A.2.

A.1 Benchmark Suite 1

After the first benchmark suite, the first results looked distrustful. This distrustfulness was mainly because the boxplot of the SUT with wires set to false is equally or even faster than on the system without any deception injected in all instances. Therefore, a lag plot analysis was conducted where the problems can be seen. Figure A.1 represents the initial VM results of the benchmark with SUT set to /home/ with wires=f. The primary analysis was guided by the /home/ SUT since it was the most complex request with multiple communications with other services that had to be done by the SUT. Hence, the most noticeable warm-up was to be expected. However, as seen in the figure, no warm-up was visible. This was most likely because the wires were set to false during runtime without the SUT being restarted. Therefore, the SUT was already warm-up during the previous run, where the wires were set to true.

Benchmark alteration: *SUT are redeployed before each benchmark.*

Another thing noticeable within Figure A.1 is that the round-trip time (RTT) spread between four and 500 milliseconds occurs too regularly, and the spread itself should not be that big. This behaviour was only noticeable in the VM benchmarks. We reasoned

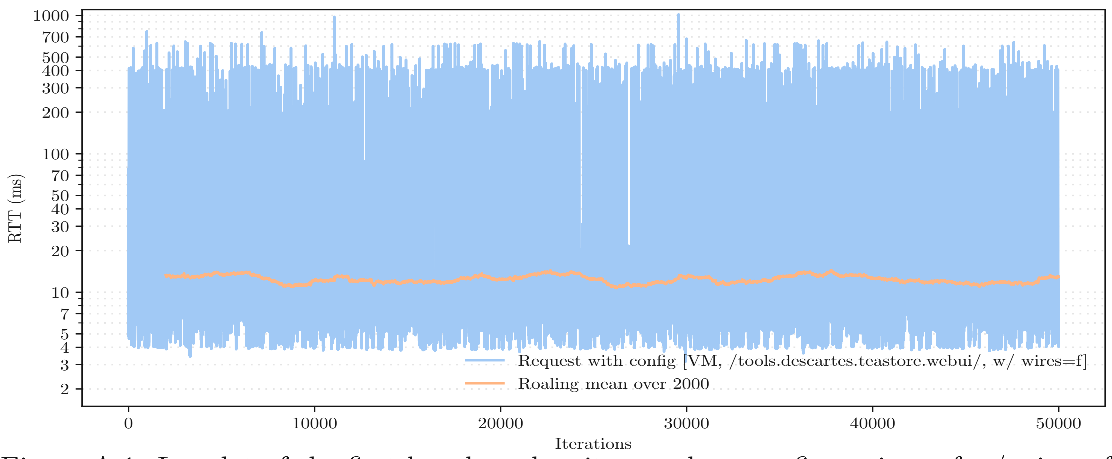


Figure A.1: Lagplot of the first benchmark suit run where configurations of w/ wires=f showed no warm-up.

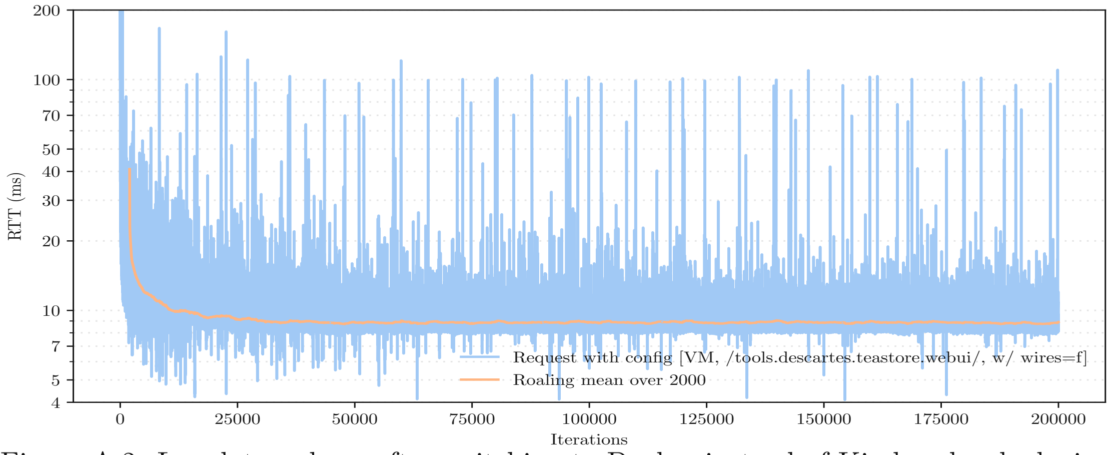


Figure A.2: Lagplot analyse, after switching to Docker instead of Kind and redeploying w/ wires=f configuration before testing and running a total of 200,000 instead of 50,000 tests.

that this is because the container is deployed with docker, and the network is rerouted over localhost, which introduces considerable network overhead and delays.

Benchmark alteration: *For the VM, a local cluster (Kind) is used to reduce the test variance and make the setup better comparable to the cloud benchmarks.*

Apart from the previously mentioned flaws, the defined warm-up phase of 10,000 requests was insufficient for some of the other analysed lag plots. Therefore, the improvements mentioned above were implemented, and the configuration of VM, /home/, and wires=f false was tested again but with 200,000 requests instead of 50,000. In the resulting lag plot depicted in Figure A.2, it can be empirically observed that the RTT stabilizes between the 30,000 and 35,000 RTT mark.

Benchmark alteration: *The warm-up and total request count are multiplied by four to 40,000 and 200,000 requests, respectively, to mitigate the impact created during caching initialization.*

As seen in Figure A.2, the maximum RTT after a warm-up phase of 40,000 was around 100 milliseconds, and the frequency of such relatively slow requests was significantly lower than in Figure A.1. This behaviour indicates that the deployment in Kind stabilizes the RTT. The warm-up with wires=f configuration is also visible and comparable to other analysed lag plots. Last but not least, no benchmark showed a destabilized RTT after 40,000 requests.

A.2 Benchmark Suite 2

After the second run, the results of the benchmarks mainly looked as expected, except for five VM runs. Those test runs have significantly higher maximum RTTs. Therefore, we conducted a lag plot analysis to verify why the spikes occurred.

As mentioned above, only the VM runs were affected, suggesting that the error might be related to the VM itself. Two effected benchmark configurations had the deception set to w/o wires, and three to w/ wires=t. Therefore, we assumed that the error does not correlate with the deception system or the benchmark system but rather with external factors such as other VMs sharing the CPU with our VM. The high maximum value occurred due to performance spikes at all benchmarks. We looked at the lag plots with an additional line representing the rolling mean of 2,000 entries for the inspection. Those rolling mean spikes lasted for around 4,000 requests, concluding that the spike affected only 2,000 requests. The performance decrease of the RTT was from 50-100%. Therefore, those spikes only increased the average performance minimally, with the median staying stable.

Out of the five benchmarks, two with the Python SUT configuration looked different. The first benchmark w/o wires shows a significant increase in the RTT at the end, as visible in Figure A.3. The affected area is about 30,000 out of 160,000 requests, with the average RTT double compared to the requests sent before the spike. Since the VM for the benchmark allocates only four out of the twelve cores of the CPU, the assumption was made that this behaviour is due to a heavy load occurring at a different VM that uses the same CPU.
The second benchmark w/ wires=t is depicted in Figure A.4. Within the figure, we saw that the RTT after the first spike had not returned to the RTT before the spike. This implies that this benchmark's median and average will be increased by about one second.

The benchmark of the repeated tests compared to their initial results are shown in Table A.1. The new tests are considered as a baseline since the old datasets introduced a performance overhead through their spikes. Hence, we used the formula $(initial - rerun)/rerun * 100\%$ to calculate the difference. For the configuration w/o wires, the median with spikes was slightly better at -1.09%, but the average performance worsened by 17.05%. For the configuration w/ wires=t, the spikes worsened the median by 10.20% and the average by 10.65%. Only the rerun benchmarks are considered in the next section.

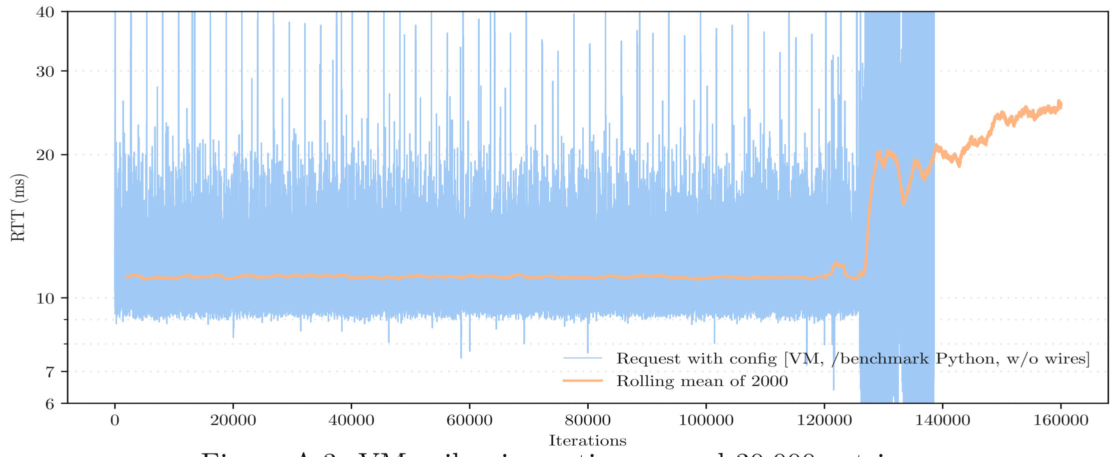


Figure A.3: VM spikes impacting around 30,000 entries.

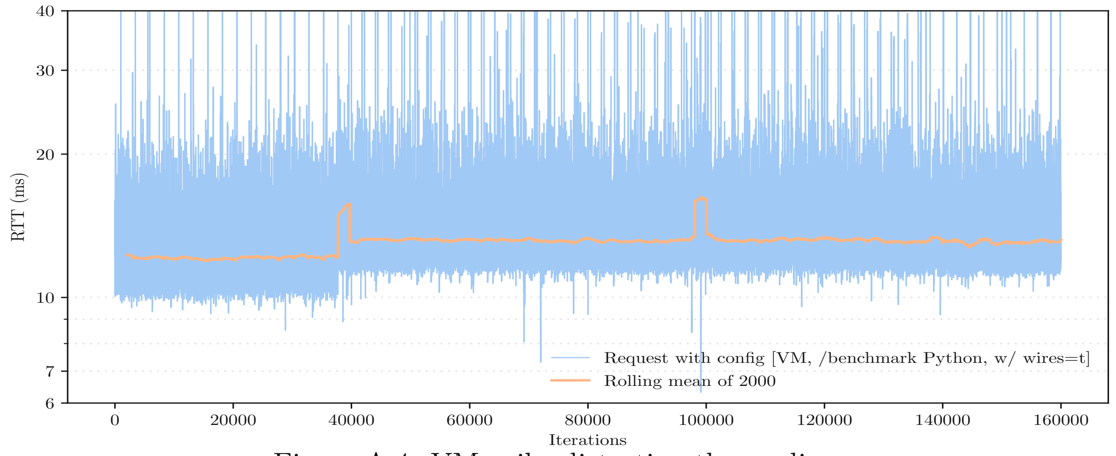


Figure A.4: VM spike distorting the median.

	min	max	median	average
VM Python w/o wires	3.93	958.94	10.84	13.25
VM Python w/ wires=t	6.32	5256.59	12.53	12.99
Rerun: VM Python w/o wires	7.26	140.72	10.96	11.32
Rerun: VM Python w/ wires=t	7.7	138.03	11.37	11.74

Table A.1: Comparison of rerun benchmarks conducted due to significant performance spices.

List of Figures

Listings

List of Tables

Bibliography

[1] K. J. Ferguson-Walter, M. M. Major, C. K. Johnson, and D. H. Muhleman, “Examining the efficacy of decoy-based and psychological cyber deception,” in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 1127–1144. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ferguson-walter

[2] A. Chuvakin, “Will deception fizzle … again?” [Online]. Available: https://blogs.gartner.com/anton-chuvakin/2019/03/01/will-deception-fizzle-again/ Last visited 2023-11-24.

[3] M. M. Islam, A. Dutta, M. S. I. Sajid, E. Al-Shaer, J. Wei, and S. Farhang, “Chimera: Autonomous planning and orchestration for malware deception,” in *2021 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2021, pp. 173–181. doi:10.1109/CNS53000.2021.9705030

[4] V. D. Priya and S. S. Chakkaravarthy, “Containerized cloud-based honeypot deception for tracking attackers,” *Scientific Reports*, vol. 13, no. 1, p. 1437, 2023. doi:10.1038/s41598-023-28613-0

[5] X. Han, N. Kheir, and D. Balzarotti, “Evaluation of deception-based web attacks detection,” in *Proceedings of the 2017 Workshop on Moving Target Defense*, 2017, pp. 65–73. doi:10.1145/3140549.3140555

[6] D. Fraunholz, D. Reti, S. Duque Anton, and H. D. Schotten, “Cloxy: A context-aware deception-as-a-service reverse proxy for web services,” in *Proceedings of the 5th ACM Workshop on Moving Target Defense*, ser. MTD ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 40–47. doi:10.1145/3268966.3268973

[7] J. Lopez, L. Babun, H. Aksu, and A. S. Uluagac, “A survey on function and system call hooking approaches,” *Journal of Hardware and Systems Security*, vol. 1, pp. 114–136, 2017. doi:10.1007/s41635-017-0013-2

[8] M. S. I. Sajid, J. Wei, B. Abdeen, E. Al-Shaer, M. M. Islam, W. Diong, and L. Khan, “Soda: A system for cyber deception orchestration and automation,” in *Annual Computer Security Applications Conference*, 2021, pp. 675–689. doi:10.1145/3485832.3485918

[9] P. Mell and T. Grance, "The nist definition of cloud computing." Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011. doi:10.6028/nist.sp.800-145

[10] D. Rountree and I. Castrillo, *The Basics of Cloud Computing: Understanding the Fundamentals of Cloud Computing in Theory and Practice*. Rockland, MA: Elsevier Science, 2014. ISBN 978-0-12-405932-0. doi:10.1016/C2012-0-02521-5

[11] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019. doi:https://doi.org/10.1145/3295739

[12] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond)," *IEEE Access*, 2023. doi:10.1109/ACCESS.2023.3281480

[13] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225. doi:10.1109/SOSE.2019.00026

[14] N. C. Rowe, J. Rrushi *et al.*, *Introduction to cyberdeception*. Springer, 2016. doi:10.1007/978-3-319-41187-3

[15] K. E. Heckman, F. J. Stech, R. K. Thomas, B. Schmoker, and A. W. Tsow, "Cyber denial, deception and counter deception," *Advances in Information Security*, vol. 64, 2015. doi:10.1007/978-3-319-25133-2

[16] L. Spitzner, *Honeypots: tracking hackers*. Addison-Wesley Reading, 2003, vol. 1. ISBN 978-0-321-10895-1

[17] J. Nazario, "Repository: awesome-honeypots." [Online]. Available: https://github.com/paralax/awesome-honeypots Last visited 2023-11-24.

[18] I. Mokube and M. Adams, "Honeypots: concepts, approaches, and challenges," in *Proceedings of the 45th annual southeast regional conference*, 2007, pp. 321–326. doi:10.1145/1233341.1233399

[19] M. Kahlhofer, M. Hölzl, and A. Berger, "Towards reconstructing multi-step cyber attacks in modern cloud environments with tripwires," in *Proceedings of the 2020 European Interdisciplinary Cybersecurity Conference*, 2020, pp. 1–2. doi:10.1145/3424954.3424968

[20] M. N. Alsaleh, J. Wei, E. Al-Shaer, and M. Ahmed, "Gextractor: Towards automated extraction of malware deception parameters," ser. SSPREW-8. New York, NY, USA: Association for Computing Machinery, 2018. doi:10.1145/3289239.3289244

[21] M. S. I. Sajid, J. Wei, M. R. Alam, E. Aghaei, and E. Al-Shaer, "Dodgetron: Towards autonomous cyber deception using dynamic hybrid analysis of malware," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9. doi:10.1109/CNS48642.2020.9162202

[22] C. Husse and J. Stenning, "Easyhook - the reinvention of windows api hooking." [Online]. Available: https://easyhook.github.io/ Last visited 2023-11-24.

[23] M. S. I. Sajid, J. Wei, E. Al-Shaer, Q. Duan, B. Abdeen, and L. Khan, "Symbsoda: Configurable and verifiable orchestration automation for active malware deception," *ACM Trans. Priv. Secur.*, vol. 26, no. 4, nov 2023. doi:10.1145/3624568

[24] D. D. Nano, "Pixie." [Online]. Available: https://ebpf.io/summit-2023-talks/ Last visited 2023-11-24.

[25] D. Reti, T. Angeli, and H. D. Schotten, "Honey infiltrator: Injecting honeytoken using netfilter," in *2023 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, 2023, pp. 465–469. doi:10.1109/EuroSPW59978.2023.00057

[26] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 942–953. doi:10.1145/2660267.2660329

[27] F. Araujo and T. Taylor, "Improving cybersecurity hygiene through jit patching," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1421–1432. doi:10.1145/3368089.3417056

[28] T. Alyas, K. Alissa, M. Alqahtani, T. Faiz, S. A. Alsaif, N. Tabassum, and H. H. Naqvi, "Multi-cloud integration security framework using honeypots," *Mobile Information Systems*, vol. 2022, pp. 1–13, 2022. doi:10.1155/2022/2600712

[29] M. Kahlhofer, P. Kern, S. Henning, and R. Stefan, "Threats of a replication crisis in empirical computer science," 2023. doi:10.48550/arXiv.2310.12702

[30] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, September 2018, pp. 223–236. doi:10.1109/MASCOTS.2018.00030

[31] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?" in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, 2020, pp. 138–149. doi:10.1145/3358960.3379124

[32] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2021. doi:10.1109/TSE.2019.2927908

[33] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering.* Springer, 2012. ISBN 978-3-642-29043-5. doi:10.1007/978-3-642-29044-2

[34] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," vol. 16, no. 3, apr 2016. doi:10.1145/2885497

[35] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* IEEE, 2011, pp. 104–113. doi:10.1109/CCGrid.2011.22

[36] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol. III: Client-server programming and applications.* Prentice-Hall, Inc., 1993. doi:10.5555/138416

[37] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, "{NetCheck}: Network diagnoses from blackbox traces," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* Seattle, WA: USENIX Association, Apr. 2014, pp. 115–128. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zhuang

[38] J. Navarro, A. Deruyver, and P. Parrend, "A systematic survey on multi-step attack detection," *Computers & Security*, vol. 76, pp. 214–249, 2018. doi:10.1016/j.cose.2018.03.001

[39] M. Kahlhofer, S. Achleitner, S. Rass, and R. Mayrhofer, "Honeyquest: Rapidly measuring the enticingness of cyber deception techniques with code-based questionnaires," 2024, unpublished manuscript under submission.

[40] L. Gkatzikis and I. Koutsopoulos, "Migrate or not? exploiting dynamic task migration in mobile cloud computing systems," *IEEE Wireless Communications*, vol. 20, no. 3, pp. 24–32, 2013. doi:10.1109/MWC.2013.6549280