University of Applied Sciences and Arts Northwestern Switzerland
School of Computer Science

Zero Trust for Processes

Master Thesis

Windisch, 16 August 2026



[1]

Student Esra Siegert

Expert Romano Roth

Supervisor Sebastian Graf

University of Applied Sciences and Arts Northwestern Switzerland,
School of Computer Science

[1] Title image generated with ChatGPT on 14 September 2025.

Notice on AI-Assisted Translation

This document was translated from German into English using artificial intelligence.

1 Abstract

Modern IT infrastructures increasingly rely on the zero-trust principle. This principle requires a control mechanism to authenticate and authorize every interaction. At present, the principle is primarily applied at the network or workload level. At the Linux level, and specifically at the Linux process level, there is no control mechanism that authenticates the executable file used when loading a new program image on the basis of a verifiable identity. By default, no check verifies where the file originates from or whether the file is trustworthy. As a result, unwanted or malicious programs can be started quickly.

This thesis addresses this issue by investigating how the idea of zero trust can be implemented practically at process level with eBPF and how such an implementation affects performance. In its implementation, the thesis follows the zero-trust maxim "Never trust, always verify." with the "Process Policy Enforcer" (PPE) and only permits the execution of files whose corresponding SHA-256 hash value a PPE user classifies as trustworthy. PPE goes one step further and restricts trusted processes in their access to security-relevant kernel operations.

PPE operates in two phases. In a first phase, PPE records which processes the system requires and which security-relevant kernel operations these processes use. During this phase, PPE checks against an external source whether the executable file used when loading a new program image is trustworthy according to the user's trust model. In a second phase, PPE enforces the recorded rules. PPE uses authentication in an eBPF program and Tetragon tracing policies for authorization.

The performance analysis indicates that PPE affects performance particularly when workloads frequently execute new programs or place a heavy load on precisely those kernel paths at which PPE and Tetragon perform checks. For NGINX, Redis and a microservice demo application, the measurement data show only limited deviations in non-overloaded ranges.

The thesis shows that zero trust at process level is feasible with certain performance penalties. However, zero trust at process level requires active management of the external source for the trustworthiness of executable files, repeated execution of PPE's first phase after software updates, and deletion of potentially compromised hash values from the policies.

Keywords:

Zero Trust, process authentication, process authorization, eBPF, Linux security, Tetragon, SPIFFE, SPIRE, Rekor

Contents

1 Abstract **iii**

List of Figures **vii**

List of Tables **vii**

2 Introduction **1**

3 Motivation **1**

4 Problem **1**

4.1 Scope 2

5 Fundamentals **2**

5.1 Zero Trust 2

5.2 Access Control Models 3

5.3 Linux Kernel 4

5.4 proc File System 5

5.5 Linux Processes 5

5.6 Linux Namespaces 6

5.7 eBPF (extended Berkeley Packet Filter) 7

6 Related Work and Tools **8**

6.1 Related Work 8

6.2 Related Tools 9

7 Need for Process Authentication **12**

8 Process Policy Enforcer Concepts **13**

8.1 Generation of Authentication Rules 13

8.2 Generation of Authorization Rules 14

8.3 Enforcement of Authentication and Authorization Rules 15

8.4 Classification of the Access Control Model 16

9 Threat Model: Target Environment **17**

10 Technical Solution Spaces **17**

10.1 Variant 1: Standalone with eBPF/Kernel Extension 17

10.2 Variant 2: Integration of Existing Tools/Technologies 18

10.3 Variant 3: Glue Code 18

10.4 Selected Variant: Integration of Existing Tools/Technologies 18

11 Process Policy Enforcer **19**

11.1 Challenges 19

11.1.1 Process Identification 19

11.1.2 Checking Trustworthiness 20

11.1.3 Detection of Executable Files 20

11.1.4 Time-of-Check-to-Time-of-Use Problem 21

11.1.5 Time-to-Live 21

11.1.6 Diversity of Syscalls and Kernel Paths 21

11.1.7 Linux Namespaces 22

11.2 Key Decisions 22

11.2.1 Identification of Files 22

11.2.2 Checking Trustworthiness 22

11.2.3 Authentication 23

11.2.4 Authorization 25

11.2.5 Relevant eBPF Program Types and Attach Points 27

11.2.6 Persistence of Rules 27

11.2.7 Git Authentication 28

11.2.8 Detection of Executable Files 28

11.3 Solution Approach 29

11.3.1 Rule Generation 29

11.3.2 Rule Enforcement 29

11.4 Architecture 30

11.4.1 Static Rule Generation Procedure 30

11.4.2 Daemon for Dynamic Rule Generation/Rule Enforcement 31

11.5 Availability and Failure Behaviour 33

12 Evaluation **34**

12.1 Performance 35

12.1.1 Syscall: stress-ng 36

12.1.2 Reference Applications: NGINX & Redis 37

12.1.3 Microservice Application: Online Boutique 40

12.2 Threat Model: Target Environment 42

13 Performance Analysis **42**

14 Discussion **44**

14.1 Goal Achievement 45

14.2 Limitations 47

14.2.1 Runtime Environment 47

14.2.2 Trust Model 48

14.2.3 Kubernetes 48

14.2.4 Managed Kubernetes: Azure AKS 49

14.2.5 Argument Checking 49

14.3 Security Effect 49

14.4 Outlook 50

15 Conclusion **50**

References **51**

A Appendix **54**

A.1 Rekor Entry 54

A.2 Tetragon Tracing Policy 55

A.3 Security-Relevant Syscalls 58

A.4 Results 59

A.5 Source Code 64

List of Figures

4.1 Use of server operating systems 2
5.1 Zero Trust 3
5.2 Kernel as interface 4
5.3 Process states 5
5.4 Process hierarchy 6
5.5 Namespace types 6
5.6 How eBPF works 8
6.1 Tool comparison 11
8.1 Generation of authentication rules 14
8.2 Generation of authorization rules 15
8.3 Enforcement of the rules 16
11.1 TOCTOU 21
11.2 Authentication in userspace 24
11.3 Authentication with eBPF 25
11.4 Tool selection 26
11.5 Process 29
11.6 Procedure for rule generation 31
11.7 Architecture 32
12.1 stress-ng benchmark 37
12.2 NGINX benchmark 38
12.3 Redis GET benchmark 39
12.4 Redis SET benchmark 39
12.5 Online Boutique benchmark 41

List of Tables

5.1 Namespaces 7
6.1 Related tools 10
6.2 Decision on related tools for PPE 12
10.1 Evaluation of solution spaces 19
11.1 Failure behaviour of PPE 34
12.1 Hardware specification 35
12.2 Software versions of the test environment 35
12.3 stress-ng test configuration 36

12.4 NGINX test configuration 38
12.5 Redis test configuration 38
12.6 Validation of security assumptions 42
13.1 Interpretation of performance measurements 44
14.1 Traceability of goal achievement 46
14.2 Not fully achieved goals 47
A.1 Security-relevant syscalls 58
A.2 Performance measurement with stress-ng 59
A.3 NGINX results 60
A.4 Redis GET results 61
A.5 Redis SET results 62
A.6 Online Boutique results 63

2 Introduction

Developed as a communication network between universities and the military, the Internet initially relied on mutual trust. This prevailing trust is still reflected today in many protocols widely used on the Internet. However, as the number of Internet participants increased, it became apparent that malicious actors can also abuse open communication protocols. Participants on the Internet therefore need to protect themselves. This is often done by isolating entire networks, that is, through so-called perimeter security. Within the perimeter, however, participants continue to trust each other. Because this approach is insufficient, the concept of zero trust has existed for several years. In the zero-trust approach, participants within a perimeter no longer trust one another by default. They only trust one another if they can mutually prove their identity. This thesis transfers this principle to the process level of Linux systems. The focus on processes is relevant because local files cannot be considered trustworthy in certain threat scenarios. One example is the attempt to integrate a manipulated version of the xz library into the important SSH tool. If this attempt had succeeded, official package managers would have installed a corrupt version of SSH on many Linux systems. Under normal circumstances, no further line of defence would then have been available. The approach pursued here changes this on two levels. First, the developed tool "Process Policy Enforcer" (PPE) authenticates the file to be executed in the path of `execve`. Authentication is performed on the basis of the SHA-256 hash value of the file underlying the process. For the trust check, PPE compares this hash value with externally managed authentication rules. Second, PPE prevents unnecessary access to system resources such as files. Final performance measurements with and without PPE show the overhead caused by this protection. The analysis of the measured values therefore provides a basis for deciding whether the overhead is acceptable in relation to the benefit.

3 Motivation

As explained in Chapter 2, this thesis focuses on increasing the security of Linux systems. Their security can be endangered for various reasons. The central concern is preventing the execution of unauthorized programs. Such programs can enter a system in different ways. An uninformed user or, where applicable, a malicious actor with access to the system could install such a program. In addition, a program may be legitimate at a certain point in time and only become insecure later. Reasons for this could include newly discovered vulnerabilities. Such vulnerabilities, in turn, may be introduced deliberately or integrated unintentionally as bugs. Deliberate introduction can occur either through the program's developer or through a software dependency supplied by a third party. If the introduction occurs through a third party via a software dependency, it is referred to as a supply-chain attack. Mitigating such attacks is one of the main motivations of this thesis. An example of a supply-chain attack is the attempt, disclosed in 2024, to introduce a manipulated version of the `xz` library into SSH. If the attempt had been successful, the official distributions would likely have distributed the malware through package managers and users would have used the software. [1] This leads to the approach of transferring zero trust to the process level: process authentication should prevent the execution of unauthorized programs, while process authorization should restrict already running malicious programs through constraints.

4 Problem

Executable files always entail risk because they may contain malware, be implemented incorrectly, or be outdated. In all cases, they increase a system's attack surface. Users should therefore

only start programs that are actually necessary and whose origin is known. In addition, running programs should only be able to access the resources they need for their task. Operating systems, however, provide only limited means and possibilities for this by default.

Against this background, this thesis investigates how the executable file used when loading a new program image can be authenticated before execution. Building on this, it examines whether the same approach can also be transferred to programs with runtime environments, such as Java or Python. It further considers how authenticated programs can be restricted in their use of critical kernel operations, which system resources can thereby be protected, and what effects the approach has on availability and failure behaviour. Since such protection mechanisms are likely to affect system performance, the thesis quantifies this influence. In addition, it examines whether the approach can also be transferred to other platforms or directly to hypervisors and how an initial rule set can be created.

4.1 Scope

This thesis concentrates on solving the problems described in Chapter 4 for the Linux operating system. It does not consider other operating systems because they are less relevant as server operating systems. Figure 4.1 shows the distribution of server operating system usage.

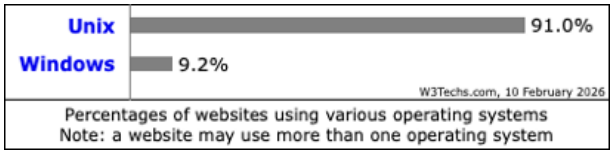


Figure 4.1: Use of server operating systems (Source: [2])

The focus is on conceptually showing how the problems formulated in the research question can be solved. Individual implementations and examples serve to demonstrate the technical feasibility of the developed concepts by way of example. The thesis does not claim to implement or evaluate all conceivable variants of the mechanisms used in full. This applies, for example, to the validation of all possible types of Rekor entries. A complete protection of all system resources is likewise outside the scope of this thesis.

5 Fundamentals

To support understanding of the problem described in Chapter 4, this chapter introduces the relevant terms and concepts.

5.1 Zero Trust

Zero trust is a term from the networking domain and describes a security concept that differs from classical perimeter security. Figure 5.1 compares the two concepts. As the graphic shows, a perimeter-based security approach protects against access from outside, but it does not restrict communication within the perimeter. An attacker is therefore likely to face few remaining obstacles after successfully breaching the perimeter. Zero trust relativizes the central role of this perimeter. Communication within the perimeter is no longer possible automatically; every participant must identify itself when establishing communication. In addition, communication with the communication partner must be explicitly permitted. [3] In practice, service meshes are often used for this purpose; in principle, they implement the concepts described by Rose et

al. in [4]. Transferred to the process level, one could argue that access to a host corresponds to perimeter protection in a network. The individual possibilities of a process to access system resources, by contrast, correspond to the communication possibilities of a service in a network. The authentication and authorization of processes therefore also apply the maxim “never trust, always verify”.

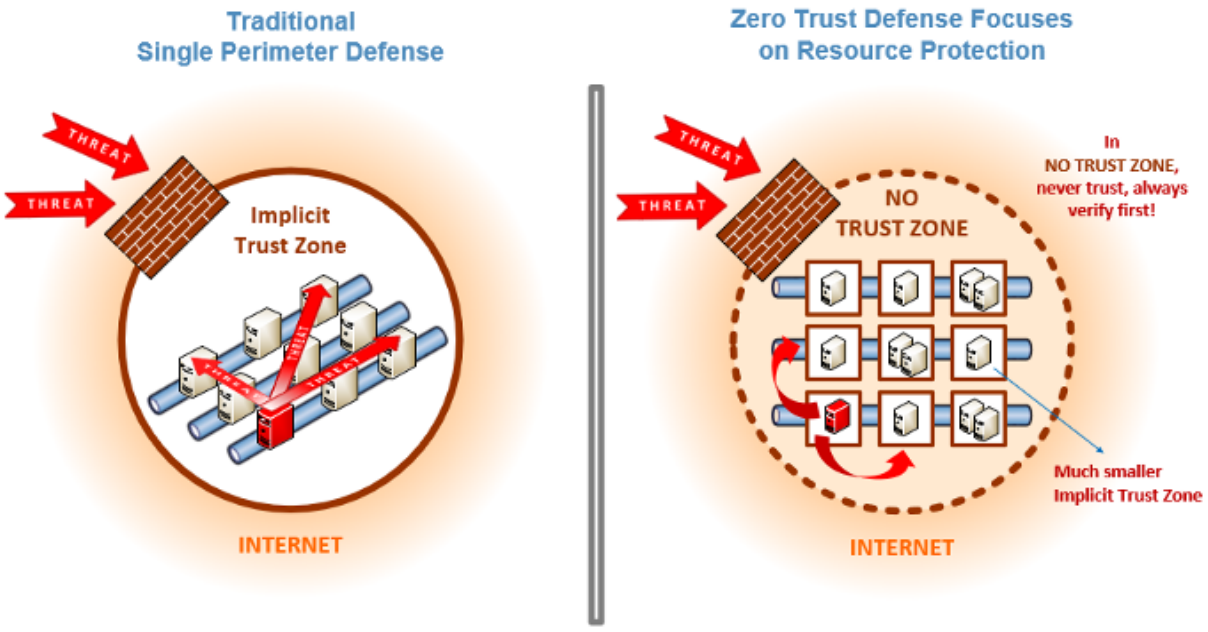


Figure 5.1: Zero Trust (Source: [5])

Zero trust therefore requires not only authentication, but also an explicit decision about permitted access. PPE needs an access control model for this purpose.

5.2 Access Control Models

Different access control models exist to control access to objects. One model used under Linux for file access is Discretionary Access Control (DAC). In DAC, the owner of an object assigns the access permissions for that object at their own discretion. When accessing an object, the system takes into account, among other things, the identity of the accessing user. [6]

A more restrictive model is Mandatory Access Control (MAC). In MAC, a central security policy assigns security labels to subjects and objects. The label of a subject describes its security clearance, while the label of an object describes its security classification. The MAC policy decides on the basis of this security information whether a subject may access an object. Unlike DAC, a user cannot modify the predefined MAC policy themselves. [6]

In Role-Based Access Control (RBAC), the system assigns permissions not directly to individual users, but to roles. A central administration assigns users the roles intended for their tasks. The permissions assigned to the roles then determine which objects a user may access. RBAC therefore simplifies permission management, especially in larger systems. [6]

In Attribute-Based Access Control (ABAC), the access decision is based on attributes of subjects and objects as well as on environmental conditions. The attributes of a subject may include, for example, its name, role, or organizational function. Environmental conditions such as time, date, or location can also be included in the decision. An access control mechanism evaluates

these attributes, the environmental conditions, and the defined policies, and decides on this basis whether access is permitted. [6]

This list of access control models is not exhaustive. Concrete access control systems may combine different models. [6]

Since PPE is intended to transfer this access control to Linux processes, the next chapter examines the interface between processes and the kernel.

5.3 Linux Kernel

The Linux kernel manages system resources and provides them through an interface that programs address via so-called system calls, or syscalls. System resources include, for example, CPU time, files, or devices. Devices are an abstraction for various system resources such as printers. Figure 5.2 schematically shows how programs access system resources via the kernel. The figure also shows that the kernel separates applications from system resources. The applications are located in so-called userspace. The system resources and the kernel itself, by contrast, are located in the protected kernelspace. [7]

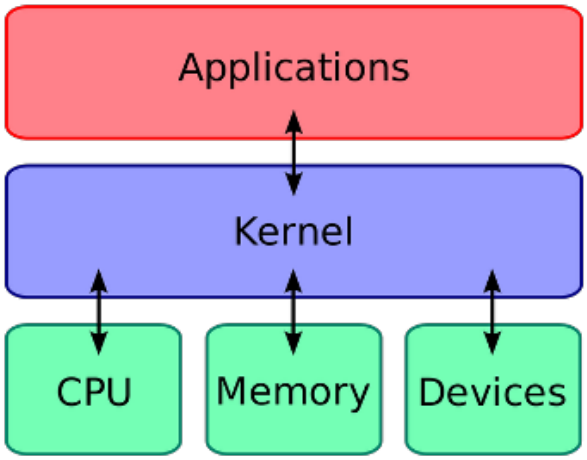


Figure 5.2: Kernel as interface (Source: [8])

Protection of system resources depends on the respective kernel operation and the active security modules. Linux Security Modules (LSM) provide a framework for integrating different Mandatory Access Control and other security modules at defined security-relevant points in the kernel. The kernel does not call LSM hooks indiscriminately for every syscall, but at security-relevant points within kernel processing. A single syscall may therefore reach no LSM hook, one LSM hook, or several LSM hooks. Conversely, several syscalls may reach the same internal kernel function and thus the same LSM hook. Which LSMs are active depends on the kernel configuration, boot parameters, distribution, and installed modules. [9], [10]

Bovet and Cesati [7] describe in detail how the Linux kernel manages system resources and how it functions internally in general. Since this thesis considers the protection of system resources, the interface to the Linux kernel is relevant, especially the control of access to this interface.

In addition to syscalls, Linux provides another important interface through the proc file system, which allows users to read process and kernel information.

5.4 proc File System

The proc file system is a virtual file system. Linux mounts it like a normal file system, but without an underlying physical device. The proc file system offers users a simple way to read kernel data structures and process the information derived from them further. [7]

Since the proc file system also provides process information, the next section introduces the process concepts on which PPE later relies.

5.5 Linux Processes

Linux processes represent a running program instance and are the users of the interface to system resources provided by the Linux kernel. With the exception of the init process, they each have a parent process and may have several child processes. From the kernel's perspective, a process is an entity to which the kernel allocates resources. A process is always in a specific state. Figure 5.3 shows the possible states. The figure shows that a process may be executing, that is, in the running state, or sleeping. If the process is sleeping, the kernel can make it runnable again depending on the sleep state and the triggering event. A zombie process is an already terminated process whose exit status has not yet been collected by the parent process using `wait()` or a related function. If, by contrast, the parent process terminates first, any child process remains as an orphaned process and is typically assigned to a subreaper or PID 1. The stopped state does not mean that a process has terminated. A stopped process continues to exist but temporarily executes no code, for example because a signal such as `SIGSTOP` or `SIGTSTP` has suspended it. [7]

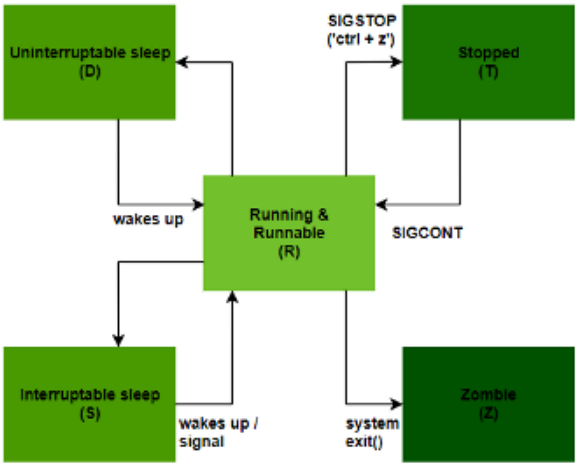


Figure 5.3: Process states (Source: [11])

As the preceding description and Figure 5.4 show, processes under Linux are organized hierarchically. Child processes therefore inherit certain attributes from their parent processes.

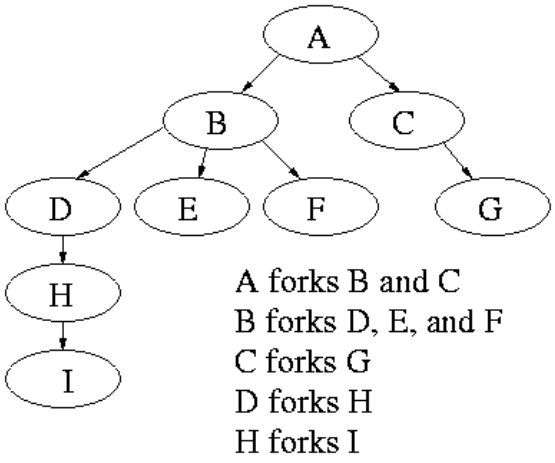


Figure 5.4: Process hierarchy (Source: [12])

The preceding discussion initially describes processes from a general Linux perspective. For PPE, however, this view is insufficient because processes may have different views of PIDs, file systems, and other resources depending on their execution context. Linux namespaces address this problem.

5.6 Linux Namespaces

Namespaces allow the kernel to isolate processes from one another through an abstraction layer. Containers are a well-known example of this use. If, for example, one logs into a container and checks the running processes, one can observe that the init process with PID 1 is not `/sbin/init`, unlike in a virtual machine, but the process with which the container runtime started the container. Linux has therefore started the container, among other namespaces, in a separate PID namespace. Within this namespace, from the internal perspective, the container's init process therefore has PID 1. Since PID namespaces are hierarchical and processes at lower hierarchy levels are visible at higher levels, the container's init process is visible in hierarchically higher namespaces, but not vice versa. Figure 5.5 illustrates this relationship for better understanding. The figure also shows that the container's init process has a different PID when viewed from outside than when viewed from inside. [13]

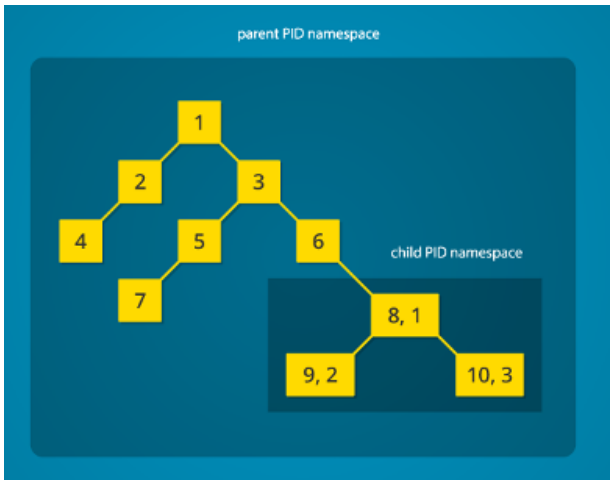


Figure 5.5: Namespace types (Source: [14])

According to the man pages [15], however, Linux namespaces are not limited to PIDs. There are further important abstractions. Table 5.1 provides an overview.

It is important to note that not every namespace is hierarchical. Every namespace has its own properties, which a solution must take into account. Their common denominator remains the goal of isolating processes and resources from one another.

Namespace	Description
Cgroup	Virtualizes the view of the cgroup hierarchy and hides parent cgroups.
IPC (Inter-Process Communication)	Isolates IPC resources such as shared memory, semaphores, and message queues.
Network	Provides an independent view of network interfaces and routing tables.
Mount	Provides an independent mount table of the file system.
PID	Provides virtual PIDs for processes. The init process has PID 1 within the namespace.
Time	Virtualizes certain timers.
User	Isolates the UID and GID table. Maps the UID or GID from the namespace to UIDs on the host.
UTS (Unix Timesharing System)	Provides independent host and domain names.

Table 5.1: Namespaces (Source: [15])

Namespaces therefore show that PPE must not interpret PIDs, file paths, and resources independently of a process's namespace context. To control the associated kernel events technically, rather than merely describing them, PPE requires a mechanism in the kernel.

5.7 eBPF (extended Berkeley Packet Filter)

Anyone who wants to add an additional capability to the kernel normally has to go through a lengthy process, often lasting years and with an uncertain outcome. The eBPF mechanism offers an alternative for extending the kernel at defined points without having to take the code through this complete process. eBPF allows developers to write programs that run in kernel context and can observe or influence specific kernel events there. The corresponding privileges are required to load the written code into the kernel. If these privileges are present, the application can generate bytecode from the eBPF code and then load it into the kernel, as Figure 5.6 shows. In a first step, the kernel verifies the code using the verifier. These protection mechanisms are intended to prevent the application from loading malicious code, or code that could crash the system, into the kernel. After being loaded into the kernel, the eBPF program can attach to defined attach points such as KProbes, tracepoints, LSM hooks, or network hooks. As a result, the kernel executes the eBPF program when kernel processing reaches the respective attach point. What the eBPF program may do at that point depends on the respective program type and attach point. A tracing program primarily observes events, whereas, for example, XDP or LSM

programs can influence decisions about network packets or security-relevant kernel operations. As the figure shows, processing an event does not have to take place entirely in the kernel. The loaded eBPF program can communicate with applications in userspace using BPF maps, ring buffers, or perf event output. This allows developers to keep the part running in the kernel small while implementing additional application logic in a higher-level programming language in userspace. [16]

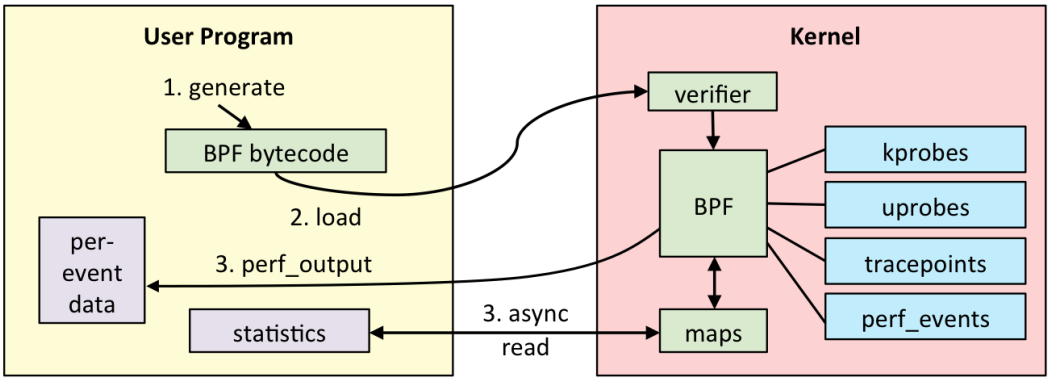


Figure 5.6: How eBPF works
(Source: [17])

This introduces the technical fundamentals central to the remainder of the thesis: processes as the units to be controlled, syscalls as the access path to system resources, namespaces as the context of such processes, and eBPF as the mechanism for control in the kernel. On this basis, the following chapter classifies which existing work and tools already address parts of this problem.

6 Related Work and Tools

Several products and research works cover partial aspects of the present thesis. This chapter classifies these approaches.

6.1 Related Work

In addition to existing products, work that examines process authentication or policy-based access control to system resources is particularly relevant.

PPE focuses on process authentication and the process authorization built on top of it. Almohri et al. [18] pursue a similar approach. Unlike PPE, however, the authors use a challenge-response mechanism instead of external verification of hash values. For this purpose, they implement a kernel module that controls syscalls and only permits them after successful authentication of a process. This approach requires modifications to the applications to be protected, because the application itself must manage the secret required for challenge-response. This creates an additional trust assumption, since a compromise of the process may also endanger the associated secret. Managing and, where necessary, rotating the secrets also increases administrative effort.

Sharma et al. pursue a Java-specific approach in their paper [19]. They extend the Java class loader so that it loads only classes already recorded by the software build in the Software Bill of Materials (SBOM). The classes are compared with the SBOM, similarly to PPE, via a hash

value. Through this mechanism, the authors prevent an attacker from loading additional Java classes at runtime. However, no external verification of the origin or trustworthiness of the classes contained in the underlying SBOM takes place.

With regard to policy enforcement and process authorization, PPE can be compared with the approach by Mazzocca et al. In [20], the authors present an eBPF-based policy manager and policy enforcer. This makes it possible to control access to system resources, such as files, by means of policies. In contrast to PPE, however, the approach does not implement hash-based authentication of the executable file used when loading a new program image. In their performance evaluation, the authors also find that the use of the eBPF programs causes only a small additional overhead. Since PPE also uses eBPF to enforce policies, these results suggest that the additional performance overhead of eBPF-based enforcement may be limited.

These works show individual solution approaches for process authentication, dynamically loaded code, and policy-based access control. PPE must additionally consider which existing tools already support these tasks in practice.

6.2 Related Tools

Several existing tools pursue security goals similar to PPE. The comparison considers these tools not only as possible alternatives, but also as possible complements to PPE. This is relevant because PPE combines process authentication, process authorization, and rule management. A tool may therefore be unsuitable for PPE, partially replace PPE, or take over a subtask and thereby meaningfully complement PPE. Table 6.1 lists the most important representatives for this thesis.

Tool	Classification Access Control	Description
Tetragon	Policy-based	eBPF-based observability and runtime enforcement tool. Configuration is flexible through policies. Excellent integration into the Kubernetes container orchestration platform. [21]
SECCOMP	Syscall filter	Kernel mechanism for restricting syscalls. The restriction is applied by the process itself, and the process passes the restrictions on to child processes. Systemd, Docker, or Kubernetes use this mechanism to restrict service processes or containers. In the case of systemd, the user can, for example, choose from a list of sensible defaults for a service process. [22], [23], [24]
AppArmor	MAC	AppArmor is an LSM-based security module that uses profiles to control which paths an application can access. If no profile exists for an application, AppArmor does not restrict its behaviour. AppArmor profiles can be recorded manually or created automatically using a learning mode. [25]
SELinux	MAC	SELinux is an LSM-based security module that complements and overlays Linux's standard DAC access control model. SELinux stores the labels it needs for access checks as extended attributes in the file system. If a subject wants to access a resource, such as a file, SELinux grants or denies access based on the context of the subject and the object. [26], [27]

Table 6.1: Related tools

Figure 6.1 shows where the tools mentioned apply in the flow of a syscall. It can be seen that both AppArmor and SELinux register as LSM modules and that the LSM framework calls them accordingly when required. Tetragon can, when required and based on the policies used, load eBPF programs at LSM hooks, KProbes, tracepoints, or other attach points. SECCOMP, by contrast, is applied early in the path of a syscall if configured.

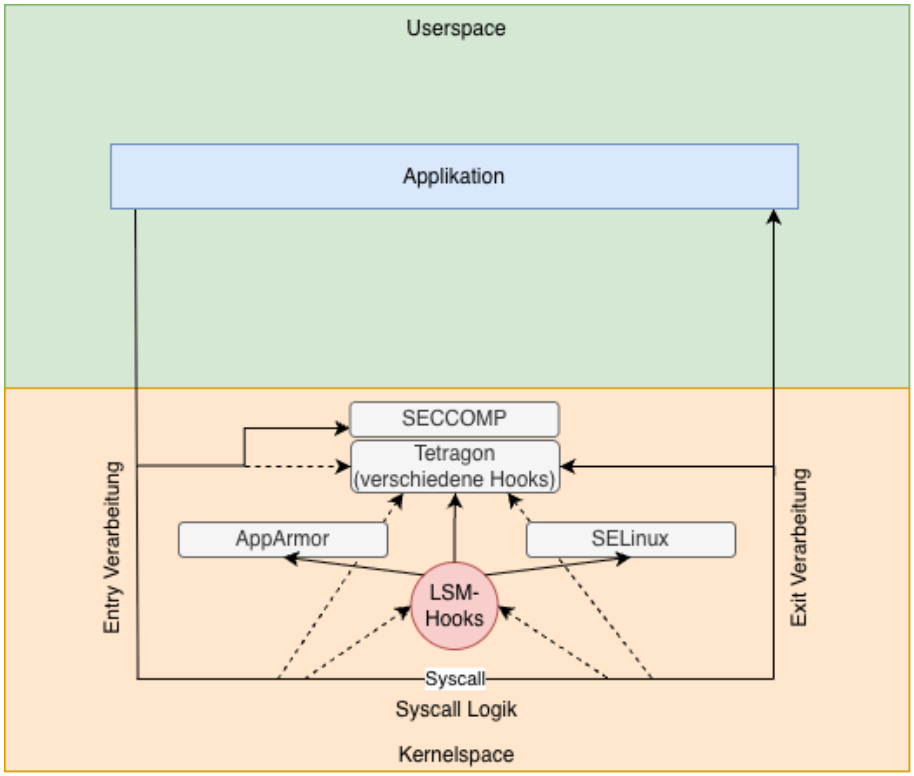


Figure 6.1: Tool comparison (Source: own representation)

With its eBPF-based approach, Tetragon is the tool closest to PPE. However, creating complex Tetragon policies can be time-consuming because of the possibilities it offers. These possibilities are precisely what make Tetragon a useful complement to PPE for implementing enforcement at security-relevant syscalls. This allows PPE to save a considerable amount of eBPF code and thus complexity. PPE complements Tetragon with hash-based authentication of the file to be executed.

SECCOMP cannot restrict processes from the outside and is therefore more suitable for service managers or container runtimes. PPE, by contrast, requires external control and is itself intended to be neither a service manager nor a container runtime. SECCOMP therefore follows a different approach and is only of limited suitability for PPE's architecture.

SELinux focuses on access control using security contexts. PPE, by contrast, authenticates the executable file used when loading a new program image on the basis of its hash value and protects resources by restricting security-relevant syscalls. The two tools therefore follow a different approach, which is why SELinux is rather unsuitable as a complement to PPE.

AppArmor focuses on path-based access control. If file system protection were the main motivation, AppArmor's profile-based configuration approach would broadly correspond to that of PPE. However, AppArmor contradicts PPE's approach insofar as AppArmor is limited to processes that run with an AppArmor profile. PPE aims to monitor processes independently of existing profiles. AppArmor therefore follows a different approach and is only conditionally suitable as a complement to PPE.

Overall, Tetragon appears to be the most suitable complement to PPE because of its architecture. Table 6.2 summarizes this decision. It does not evaluate the tools as general security solutions, but shows which role they could assume in the PPE approach and why this thesis uses or does not use them.

Tool	Relevant capability	Limit in the PPE context	Decision
Tetragon	Observes and blocks security-relevant kernel operations through declarative policies.	Tetragon does not authenticate the executable file that a process uses at program start on the basis of an externally verified hash value.	PPE uses Tetragon for authorization and complements it with its own hash-based authentication.
SECCOMP	Restricts syscalls for processes whose start environment sets or inherits the filter.	PPE would have to wrap the start of all relevant programs in order to launch arbitrary host processes with SECCOMP profiles.	PPE does not use SECCOMP as a core building block.
AppArmor	Restricts access path-based through profiles.	AppArmor requires suitable profiles and replaces neither hash-based authentication nor the dynamic derivation of PPE rules.	AppArmor can additionally harden the host; however, PPE does not integrate AppArmor.
SELinux	Restricts access through security contexts and policies.	SELinux follows its own label and policy model and would not directly simplify the PPE architecture.	SELinux can additionally harden the host; however, PPE does not integrate SELinux.

Table 6.2: Decision on related tools for PPE

7 Need for Process Authentication

The comparison with existing work and tools shows that authentication of an executable file before its execution in particular is not sufficiently covered. This chapter therefore derives why Linux's own permission mechanisms are insufficient for this goal and why PPE requires an additional trust anchor.

Classical Linux permissions specify, among other things, whether a user may in principle execute a file. The owner of the file or a correspondingly privileged process sets these permissions. If no additional protection mechanisms are active, the user or administrator therefore essentially decides whether they consider an executable file trustworthy enough for execution. Only when the process wants to access resources through a syscall for which the corresponding user is not authorized does Linux prevent access. As described in Chapter 6, this is DAC.

This approach requires a high level of competence from the user and assumes that the user can assess the risk of an executable file. Even experienced administrators, however, cannot simply see from a file whether it is dangerous.

A check before execution is therefore necessary. A common approach is to compare a file with a published hash value. Download pages for software often provide a second file containing the expected hash value alongside the executable file. After the download, the hash value of the downloaded file can be compared with this reference value. This makes it possible to determine whether third parties have modified the file in transit to the user's computer.

This approach assumes that the source of the published reference hash value is trusted. If this is not the case, the software should not be used.

If such a mechanism is triggered automatically on every program execution, Linux's own mechanisms continue to apply. In addition, however, a layer is created on which it can be defined which manufacturers the system trusts.

It is important that the trust anchor – in this case the expected hash value and the associated trust model – is not located exclusively on the local system and is secured separately. Otherwise, someone could potentially manipulate the reference value, making the check at startup pointless. This need raises the question of which concrete attack types the additional control mechanism should address. Before this thesis examines that question in Chapter 9, the next chapter shows how PPE conceptually implements process authentication.

8 Process Policy Enforcer Concepts

The previous chapter explained the benefits of additional process authentication. This chapter provides an overview of how PPE implements it and which additional components PPE requires for this purpose. In principle, PPE operates in two phases. In a first phase, PPE generates the required authentication and authorization rules. In a second phase, PPE applies these rules to authenticate and authorize processes. PPE can theoretically skip the first phase if another component generates the rules. However, the rules must then correspond to the format expected by PPE.

8.1 Generation of Authentication Rules

Figure 8.1 shows the components that PPE requires to generate authentication rules. As its central element, PPE scans the file system for executable files. If the chosen trust model requires external validation of the hash value of an executable file found, PPE queries Rekor, a transparency log, for associated metadata. If PPE finds a matching entry for an executable file and its metadata meets the requirements of the configured trust model, PPE considers the file trustworthy. In the appendix, Chapter A.1 shows an example of such metadata for the `kubectl` tool, including a decoded Fulcio certificate. Fulcio is a certificate authority operated by Sigstore that issues short-lived signing certificates for identities authenticated via OpenID Connect (OIDC). The certificate binds the public key used for the signature to the identity authenticated via OIDC. With the associated private key, for example, a maintainer can sign a build artifact or its digest and make the associated signature information traceable through Rekor. [28], [29] PPE decides on the basis of a list of permitted combinations of OIDC issuers and email addresses whether the identity contained in such a certificate corresponds to the administrator's trust model. [30] If the identity corresponds to the trust model, PPE creates a SPIRE registration entry with the hash value of the executable file as selector. For this purpose, PPE requests its own workload identity from the local SPIRE agent. After successful workload attestation, PPE receives a workload identity and uses it for authenticated communication with the designated SPIRE interfaces. PPE then stores the authentication rule in the SPIRE server. Listing 1 shows such a registration entry with the hash value used as selector.

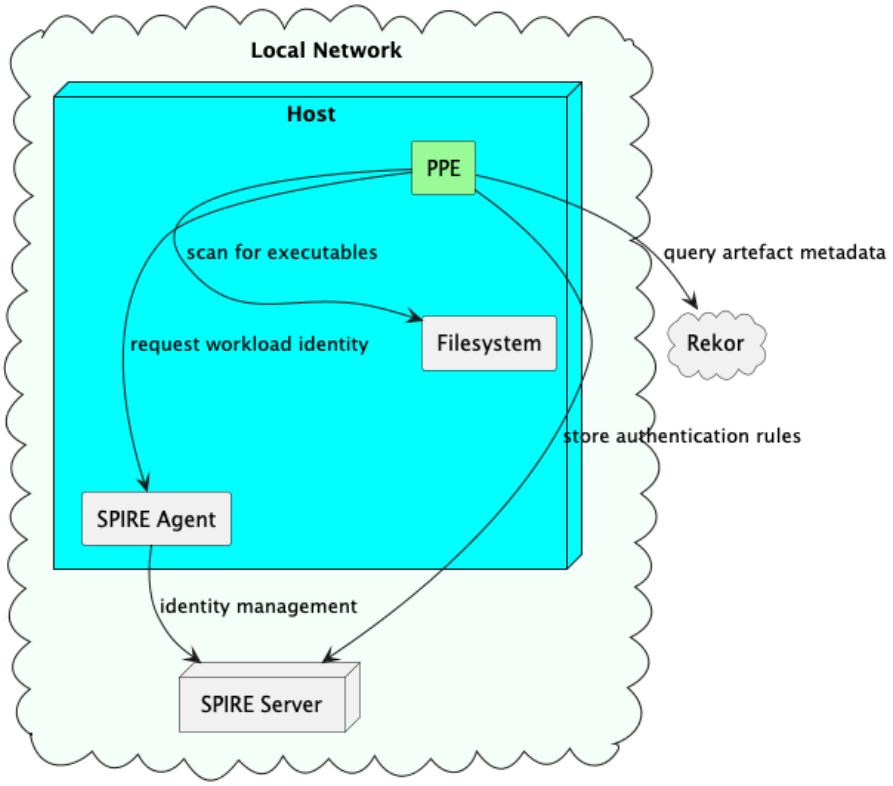


Figure 8.1: Concept for generating authentication rules (Source: own representation)

Listing 1: SPIRE Registration Entry

```
Entry ID         : 1f1516a5-0317-4022-9dea-de023f73353b
SPIFFE ID        : spiffe://example.org/ppe/kubectl
Parent ID        : spiffe://example.org/ppe
Revision         : 0
X509-SVID TTL    : default
JWT-SVID TTL     : default
Selector         : unix:sha256:316
   d712726d857c20744c57cb36aa47cfc79bc0af7e82cebf3780244b654c073
```

The authentication rules therefore answer which executable files may start. For zero trust at process level, however, this is not sufficient, because PPE must also restrict the behaviour of processes that have already started.

8.2 Generation of Authorization Rules

PPE uses the components shown in Figure 8.2 to generate authorization rules. In this mode, PPE adds policies for observing system behaviour to Tetragon. Listing 4 in the appendix shows such a monitoring policy. After PPE has added the monitoring policies to Tetragon, PPE opens a streaming connection to the Tetragon API. Tetragon delivers the events captured by the monitoring policies to PPE over this connection. PPE initially buffers incoming events in memory. As soon as PPE receives a stop signal, PPE generates the authorization rules from the recorded events and transfers them to the Git service. Before PPE terminates, it removes the policies previously added to Tetragon.

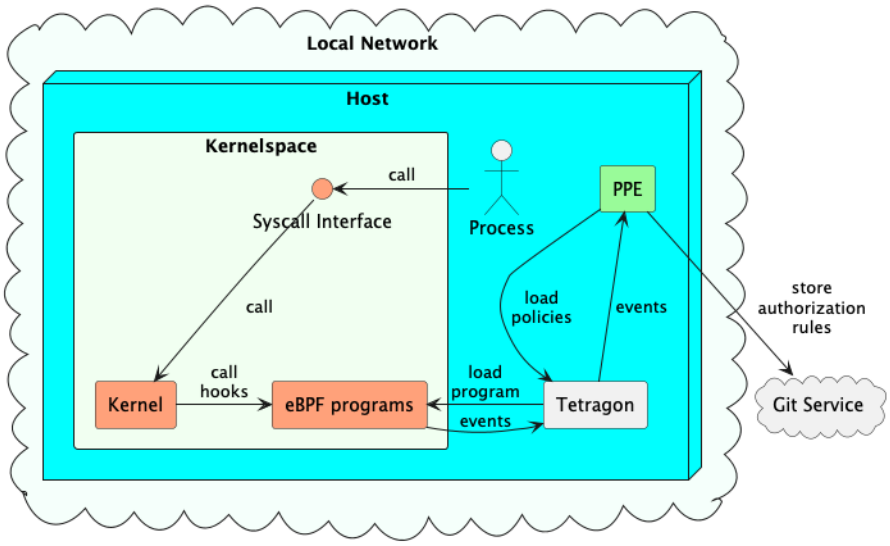


Figure 8.2: Concept for generating authorization rules (Source: own representation)

The authorization rules thus complement the authentication rules with specifications for the behaviour of running processes. During operation, PPE must apply both types of rules together.

8.3 Enforcement of Authentication and Authorization Rules

Figure 8.3 shows the components of the second phase, the enforcement phase. As already in the generation of authentication rules, PPE requests a workload identity from the local SPIRE agent. With this workload identity, PPE then loads the previously generated authentication rules from the SPIRE server. PPE reads the hash values of the authenticated executable files from the selectors of the registration entries. It then loads the eBPF program required for authentication and makes the previously loaded hash values available to it. From this point on, PPE only permits loading a new program image if the hash value of the underlying executable file is contained in the loaded authentication rules. To this end, the eBPF program calculates the hash value of the file intended for execution and compares it with the known hash values. After initializing process authentication, PPE starts authorization. For this purpose, PPE loads the authorization rules from an external Git service. PPE then adds the authorization rules to Tetragon through the Tetragon API. Listing 5 in the appendix shows such an authorization rule. Tetragon enforces these rules in the kernel using the eBPF programs loaded for this purpose.

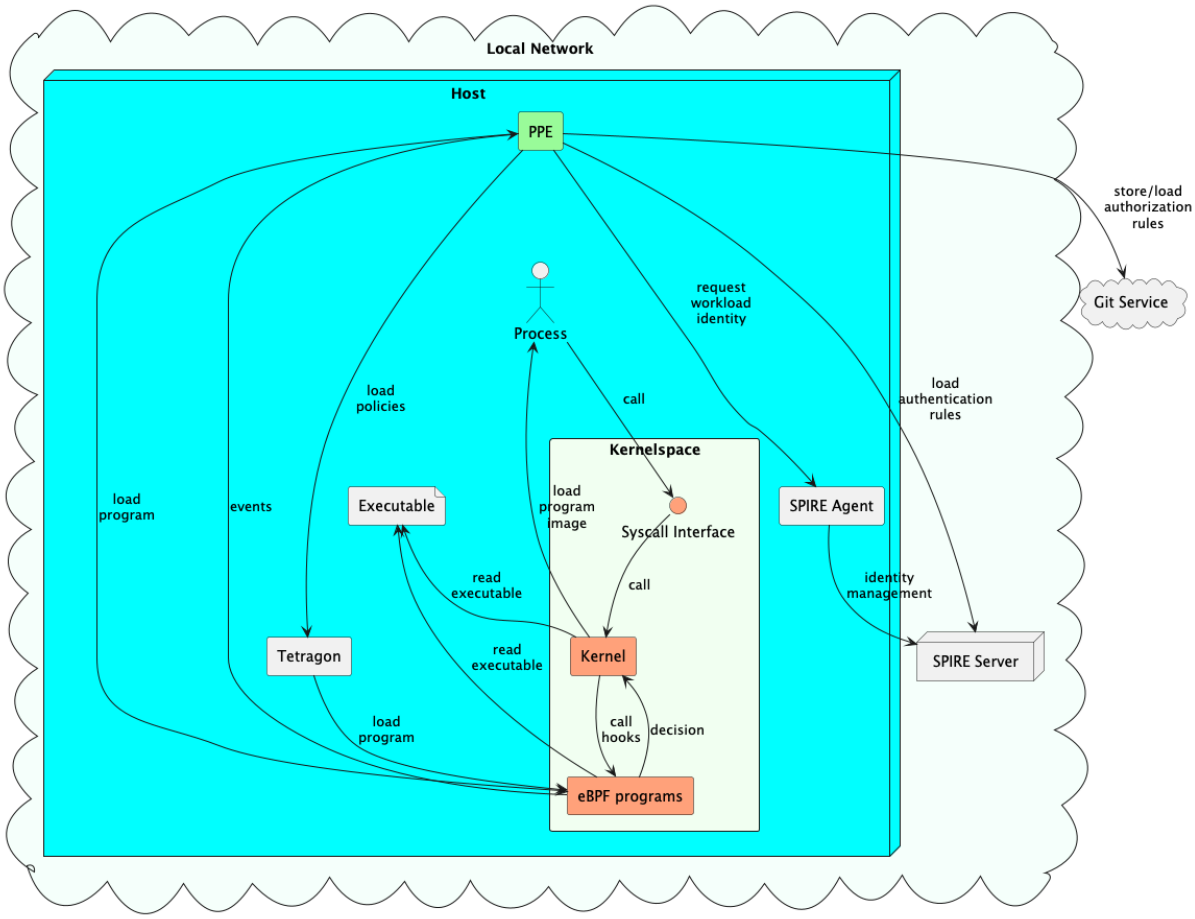


Figure 8.3: Concept for enforcing the authentication and authorization rules (Source: own representation)

This overview of how PPE works shows which components create and enforce rules. From this, the thesis derives which access control model PPE is conceptually closest to.

8.4 Classification of the Access Control Model

If only the enforcement of access rules is considered, PPE largely follows the MAC principle. A central instance specifies both the authentication and the authorization rules. The controlled processes and the owners of the affected objects cannot modify these rules themselves. If the view is expanded to include the features on the basis of which PPE and Tetragon make access decisions, the system additionally exhibits properties of ABAC. For authentication, PPE uses the hash value of the executable file as an attribute for identifying the program to be executed. For authorization, Tetragon can include different characteristics of a process, its execution context, and the observed kernel operation in the evaluation of a policy. PPE therefore has properties of both MAC and ABAC.

This conceptual classification determines which decisions PPE makes and according to which access control principle this thesis classifies them. Building on this, the threat model can determine which attack types these decisions should address.

9 Threat Model: Target Environment

A threat model normally analyses possible dangers to a system. The present thesis, however, does not develop a classical system; it investigates security mechanisms for existing systems.

As Chapter 4 explained, process authentication and process authorization when accessing system resources are central. These mechanisms are not an end in themselves, but are intended to prevent concrete attacks. The STRIDE model [31] is therefore used to classify the relevant attack types. The thesis assumes that an attacker succeeds in executing a program in the userspace of a system.

PPE does not combat all attack vectors described in STRIDE. PPE concentrates on those attack vectors that process authentication and authorization can mitigate. PPE can therefore contribute especially in the areas of spoofing, tampering, information disclosure, and elevation of privilege.

Process authentication prevents the re-execution of a manipulated file if an attacker replaces a legitimate executable file and PPE does not permit the new hash value. This reduces the risk that a manipulated file can start again as a legitimate file.

Process authorization is intended to restrict the capabilities of processes. Depending on how process authorization restricts a process, or which security-relevant kernel operations it grants to the process, it helps reduce malicious modification of files (tampering) or prevent reading sensitive files (information disclosure). By restricting such operations, process authorization also reduces the risk of an elevation-of-privilege incident, because in the best case it prevents the kernel operations required for it.

Although PPE cannot mitigate all attack vectors described in STRIDE, it reduces the risk of individual relevant attack types.

This threat model shows that PPE requires both decisions in the kernel path and management of the associated rules. The following chapter therefore considers which technical solution spaces can meet these requirements.

10 Technical Solution Spaces

To solve the problem described in Chapter 4, the thesis considers three technical solution spaces. Variant 1 investigates an implementation with no or minimal use of existing tools. Variant 2 uses existing tools where this is useful and complements their functionality with concrete added value. Variant 3 largely avoids custom code and examines how far the problem can be solved by combining existing tools.

10.1 Variant 1: Standalone with eBPF/Kernel Extension

Variant 1 provides for developing the solution, where possible, without dependencies apart from the Linux kernel itself. Since intercepting security-relevant kernel operations without additional tools requires a mechanism in kernel context, a custom kernel extension would be an option. Such an extension could be implemented as a kernel module or through eBPF; both paths would be demanding. In particular, a generally usable kernel module, or one that could even be integrated into the upstream kernel, would require substantial time. Such an approach might already be obsolete by the time it achieved a certain distribution. It must also be considered that developers usually have to write kernel code in the languages supported by the kernel build system and under its restrictions. In practice, C dominates, with Rust becoming increasingly important. On the other hand, this approach offers great flexibility because it can intervene more deeply in

kernel mechanisms than would be possible through the syscall interface alone. Nevertheless, this variant also contains parts that should run outside the kernel. In particular, comparing the hash with an external source should not take place inside the kernel. eBPF provides one possible way to do this. In short, eBPF allows verified programs to be loaded from userspace into the kernel in a comparatively simple way and to be executed there at defined attach points. The eBPF verifier checks and limits which code the kernel can execute. Defined attach points trigger the code. Chapter 5.7 describes in more detail how eBPF works.

Variant 1 thus shows the maximally independent but costly solution space. The second variant reduces this effort by selectively integrating existing tools.

10.2 Variant 2: Integration of Existing Tools/Technologies

This variant provides for a combination of custom development and the integration of existing tools. Since the variant must also be able to react to syscalls when integrating existing tools, the use of eBPF appears useful in this variant. This combination offers great flexibility to develop new functionality without having to forgo proven solutions. In this variant, it is therefore possible to draw on and integrate the tools considered in Chapter 6.2. By integrating existing tools, the codebase of the resulting solution is reduced, and the maintenance effort decreases accordingly.

Variant 2 thus retains custom functionality while using existing components. The third variant shifts the emphasis even more strongly toward existing tools.

10.3 Variant 3: Glue Code

This variant depends on the tools used. It is therefore not yet possible at this point to define whether eBPF, a kernel module, or another approach would be used. The major advantage of this variant is that PPE requires little to no new code. This reduces maintenance effort considerably. The disadvantage of this variant, however, is correspondingly that it can create hardly any new functionality. New functionality could at most result to some extent from the skilful combination of existing tools. The variant therefore also offers little flexibility, because it is completely dependent on external tools.

Having described the three solution spaces, the thesis now compares their properties.

10.4 Selected Variant: Integration of Existing Tools/Technologies

Table 10.1 lists the advantages of the variants presented. The evaluation uses the assessment criteria flexibility, dependence on third parties (tools, vendors, and so on), maintenance effort, and development effort. The most important criteria for the thesis are flexibility and development effort. The criteria dependence on third parties and maintenance effort are more relevant for later practical use of the solution. The flexibility criterion describes the extent to which the variant offers possibilities to implement functionality for this thesis. Development effort, by contrast, describes the expected development effort resulting from the choice of variant. Higher development effort can only be justified if it offers considerable advantages in flexibility.

Criterion	Variant 1	Variant 2	Variant 3
Flexibility	high	high	low
Development effort	high	medium	low
Dependence on third parties	low	medium	high
Maintenance effort	high	medium	low

Table 10.1: Evaluation of solution spaces

Table 10.1 shows that variants 1 and 2 offer considerably more flexibility than variant 3. Variant 2 differs from variant 1 in that it incorporates existing tools where useful. As the table shows, this comes at the cost of independence. In contrast, maintenance and development effort are lower than in variant 1 while flexibility remains the same. Table 10.1 could suggest that variant 3 is even more advantageous in terms of effort and maintainability. However, variant 3 forgoes a high degree of independence and flexibility. The decisive factor for choosing variant 2 is the balanced compromise with regard to the applied criteria. It offers a high degree of flexibility without foregoing the advantages of existing tools.

After selecting the technical solution space, the following chapter describes the concrete Process Policy Enforcer, including its challenges, decisions, and components.

11 Process Policy Enforcer

In response to the problem statement in Chapter 4, this thesis implements the "Process Policy Enforcer" (PPE). The solution variants examined lead to the key decisions in Chapter 11.2 and to the architecture in Chapter 11.4. Chapter 11.1 first describes the challenges addressed in the process.

11.1 Challenges

Developing PPE involved several conceptual and practical challenges.

11.1.1 Process Identification

To authenticate executable files and authorize processes, both processes and the associated executable files must be uniquely identifiable. Both are less trivial than they may appear at first glance.

Files

An obvious approach would be to identify files by their path. A path, however, is not unique, because multiple paths can point to one and the same file, or the content of the referenced file can change. In addition, the path depends on the currently set root in the file system (chroot) and, as already described in Chapter 5.6, on the current mount namespace.

Another possibility would be to identify a file by its inode and device. In addition to a unique identifier, the inode would also provide further metadata such as file size, UID, or permissions. In contrast to the path, the inode would give PPE a unique identifier, but the inode likewise says nothing about the content of the linked file. Even if the content of the file were checked initially, the content could change later without the inode changing. [7]

For PPE, it is therefore insufficient to determine the reference to a file uniquely; what matters is the verifiability of the file content. One way to check the content of a file is to compare it with an exact copy. This is not practicable for PPE, however, because files can be very large and PPE would therefore have to keep an exact copy for comparison for every legitimate file. This can be avoided by calculating cryptographic hash values of files. A hash function maps the content of a file to a fixed-length value. SHA-256 is a modern and widely used hash function. In the case of SHA-256, the length is 256 bits. [32]

For PPE, using cryptographic hash values for identification is therefore the most appropriate approach. With high probability, they uniquely identify the checked file content independently of its reference. Because of their constant length, PPE can easily persist them as identifiers in a file or an external system. The only disadvantage compared with paths and inodes is the comparatively high computational effort. This could lead to performance penalties.

Processes

PPE must be able to identify processes in order to assign them to the authenticated executable files and subsequently authorize them. Under Linux, a process is often referenced by its process ID (PID). PPE can use this to reference a process within a specific PID namespace at a specific point in time. PPE must, however, observe the concept of Linux namespaces (see Chapter 5.6). For a system-wide view, PPE must therefore run in the host's initial PID namespace and use, for identifying other processes, the PID as seen from this initial PID namespace. The PID from the perspective of the process usually differs from the PID from the perspective of the initial PID namespace if the process runs in its own PID namespace. PPE must also take into account that a PID is not a persistent identity and that the assignment of a PID to an executable file may therefore change at a later point if Linux reuses the PID. For more robust identification, additional information such as the process start time or an internal kernel process reference would be more meaningful. [7]

PPE can further use the PID to obtain additional information about a running process. For example, PPE can access the directory structure from the process's perspective via the proc file system using the PID, or read the command used for program execution (see Chapter 5.4).

In summary, PPE must reference processes by their PID as seen from the host's initial PID namespace and take into account that a PID is not a persistent identity.

11.1.2 Checking Trustworthiness

PPE cannot infer from the mere presence of a file whether it is trustworthy. Instead, PPE can use a trust source to check whether a file is known there. At this point, the PPE user must trust this trust source or the information provided by it. How far this trust extends and what it looks like depends primarily on the user and their conditions. Mere presence need not mean that the user trusts a file. A user might, for example, trust a file only if the metadata provided by the trust source satisfies their requirements. The trust model may range from very restrictive to very permissive. Between these extremes there are moderate trust models, such as trusting only certain originators or only internal trust sources. Which trust model is useful depends, as already mentioned, on the conditions. PPE's task is to cover as broad a spectrum of conditions as possible.

11.1.3 Detection of Executable Files

During authentication rule generation, PPE must detect whether a file is an executable file. Under Linux, this cannot readily be read directly from the file. Detection is important so that

PPE checks only executable files for trustworthiness. The reason is that checking trustworthiness is relatively expensive. If every single file were checked for trustworthiness, rule generation would not scale, especially on systems with many non-executable files, and would unnecessarily take a great deal of time.

11.1.4 Time-of-Check-to-Time-of-Use Problem

The Time-of-Check-to-Time-of-Use problem (TOCTOU) describes, in principle, the problem that a resource can change between checking and use. In [33], Bishop and Dilger describe this using file access. In their example, they show that the file name can change between checking the access rights and opening the file. This can potentially result in an unauthorized file access.

PPE has a similar problem if authentication of an executable file takes place in userspace. Strictly speaking, the problem here is not that a resource changes between checking and use; rather, a time window between loading the program image and making the decision allows an unauthenticated program to perform activities. Figure 11.1 marks this time window in red. The figure also shows that an eBPF program can detect the loading of a new program image, but in a normal, non-sleepable eBPF program it cannot suitably determine the hash value of the executable file and thus lacks the basis for an authentication decision. PPE can only stop the process if authentication in the userspace program fails. In the meantime, the process can perform activities without control, which constitutes a security problem.

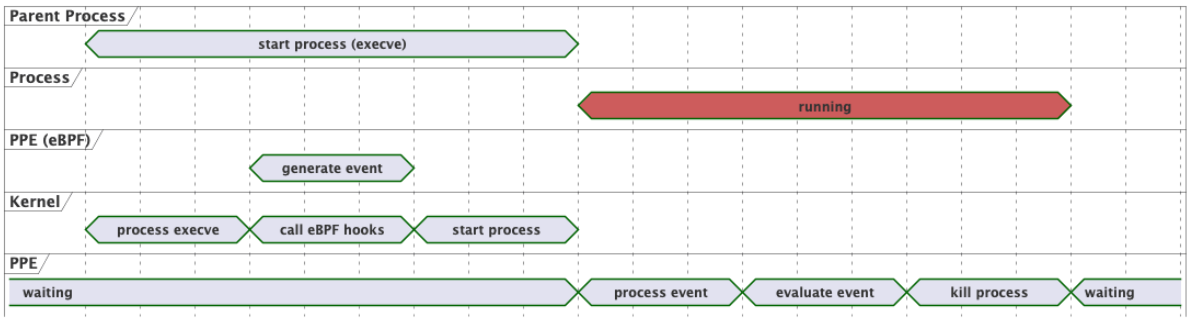


Figure 11.1: TOCTOU-like problem in PPE (Source: own representation)

11.1.5 Time-to-Live

It is important that PPE regularly reconciles the hash values to be used for authentication with the stored rules if it keeps these hash values in memory. If a user decides, because of a security incident, for example because a supply-chain attack has affected a certain program and that program is no longer trustworthy, that a hash value is no longer valid, PPE should remove it from memory. PPE must therefore regularly reconcile rules in memory with the stored rules.

11.1.6 Diversity of Syscalls and Kernel Paths

The Linux kernel contains a large diversity of syscalls and internal kernel paths whose purposes could hardly be more different and which sometimes even overlap. From PPE's perspective, not all syscalls and kernel paths are equally relevant. PPE is concerned with security-relevant kernel operations whose use could pose a security risk to the system. PPE must therefore determine which operations are relevant and then create authorization rules for them. This gives rise to the next challenge: PPE must decide how fine-grained these rules should be. The question is whether PPE must authorize only an operation itself or also its arguments. If PPE must also authorize the arguments, the complexity of the rules increases.

11.1.7 Linux Namespaces

Linux implements the concept of namespaces to control what a process can see. For this purpose, the kernel adds a process to different types of namespaces. Each process belongs to exactly one namespace of each namespace type. Linux provides various namespace types. For PPE, the PID namespace and the mount namespace are primarily relevant.

The mount namespace controls which files a process can see, and the PID namespace abstracts the PID in the Linux namespace from the PID of the host namespace. Containers of a container runtime, for example, do not see the host's entire file system, but the files of the overlay file system. Likewise, a container always sees PID 1 for the container's root process, regardless of the PID that the root process has in the host namespace. [13]

PPE must take this Linux namespace behaviour into account and handle it correctly.

These challenges lead to the central technical decisions for PPE.

11.2 Key Decisions

Before and during implementation, the thesis had to make directional decisions. This chapter describes the most important decisions and the corresponding trade-offs.

11.2.1 Identification of Files

A file can be identified either by its content or by a reference. Chapter 11.1.1 has already explained why the reference is insufficient for authenticating a file. Nevertheless, a path as a reference to a file is easier to handle than the file's hash value. By using the hash value for authentication, PPE can check whether the system executes only approved executable files. Since authentication of the executable file always takes place before any authorizations, PPE can in principle regard the resulting process and the underlying executable file as approved within its own trust model. The reference to a file can therefore be used later for authorizations. In principle, it does not matter what the reference looks like as long as authentication was performed correctly. If the content behind a reference changes, this is irrelevant as long as the system does not execute the file with the new content again. If it does, PPE authenticates the executable file again on the basis of the hash value. In theory, from a security perspective, re-authentication at every security-relevant kernel operation would be desirable, but from a performance perspective it is not realistic because of the computational effort for hash calculations. Since PPE records the authorization rules beforehand in the rule generation process step and uses the path associated with the process for this purpose, it would theoretically be possible to replace the executable file behind such a path. This effectively constitutes a risk. Nevertheless, PPE must authenticate the replacement executable file on the next program execution. PPE accepts this risk for authorization because authentication is based on the hash value of the executable file. PPE therefore uses the hash value for authentication and the path for authorization.

11.2.2 Checking Trustworthiness

PPE must check executable files on the system with regard to their trustworthiness. One approach is to assume all files on a system to be trustworthy at the time of rule generation. Another approach is to verify the trustworthiness of a file's origin externally.

The first approach has the advantage that it is easy to implement and has no dependency on surrounding systems or external providers. Its disadvantage is that it provides no possibility to update the list of trustworthy files. The only way to update this list is to run rule generation

again. It must be noted, however, that every rule generation under these circumstances entails risks, because it assumes that no malware is present on the system at the time of rule generation. In this case, the trust anchor lies in the time of rule generation.

In the second approach, the trust anchor lies not in time, but in an external source. How strongly PPE can trust this external source depends on its trustworthiness. Rekor is suitable as a possible public source; users can also operate their own Rekor instance if needed. The public Rekor instance offers transparency, traceability, and cryptographically verifiable log entries. Maintainers of software artifacts can thereby make signature or provenance information for their artifacts traceable in Rekor. Since Rekor offers an API, PPE can query, using the hash value, whether a signed and traceable entry for a file exists in the transparency log. Such an entry does not automatically prove that an artifact is secure or legitimate. Whether the signer is trustworthy and whether the artifact may be approved depends on the user's trust model. A disadvantage of this approach is its dependence on a functioning network connection and the availability of Rekor at the time of rule generation. [34]

Both variants have in common that PPE creates the rules at a specific point in time. The second approach, however, adds an additional security layer to the trustworthiness check, because in this variant PPE actively checks, in addition to the point in time, whether the executable file has a traceable entry in Rekor's transparency log that matches the trust model. The price is that rule generation takes more time, because PPE requires external queries for this purpose. Since rule generation takes place at most at intervals, PPE accepts this disadvantage in favour of a higher level of trust.

11.2.3 Authentication

Chapter 11.1.1 has already derived how a file can be identified and assigned to a process. To authenticate the executable file used when loading a new program image, PPE must compare its unique identification with the identifiers recorded during rule generation. PPE could do this in userspace or directly in the kernel in the eBPF program. If PPE performs the comparison in userspace, it could use the Rego policy language. Regardless of where the decision is made, PPE must intercept the loading of a new program image in an eBPF program in the kernel. PPE must also ensure that authenticated hash values are not valid without limit for the reasons stated in Chapter 11.1.5. PPE must therefore no longer use deleted hash values for authentication.

Authentication in userspace has the advantage that calculating the hash value of the executable file does not block the syscall. PPE can also use policy frameworks in userspace to make a decision. One such framework is the Open Policy Agent (OPA) with the Rego policy language. [35] PPE can therefore generate Rego policies with the permitted hash values and use a library provided by OPA to make decisions. Decision-making takes place in parallel with program execution, as Figure 11.2 shows. If the decision is negative, PPE must terminate the process. Since decision-making and program execution are independent, PPE has no control over the activities of the process until the kill point. PPE receives the information that the kernel is about to load a new program image through an eBPF program loaded into the kernel (shown in the figure as "PPE (eBPF)"). If the kernel calls this program as part of the corresponding syscall, "PPE (eBPF)" writes this event to a ring buffer. PPE, which according to the figure runs in userspace, can wait for events in the ring buffer and react to them. It is important that PPE accesses the executable file via the proc file system, because this allows PPE to avoid problems with Linux namespaces that may differ for the process. In addition, PPE must periodically check whether the Rego policies in the source have changed. If this is the case, PPE must reload the rules and must no longer use any invalid hash values for authentication.

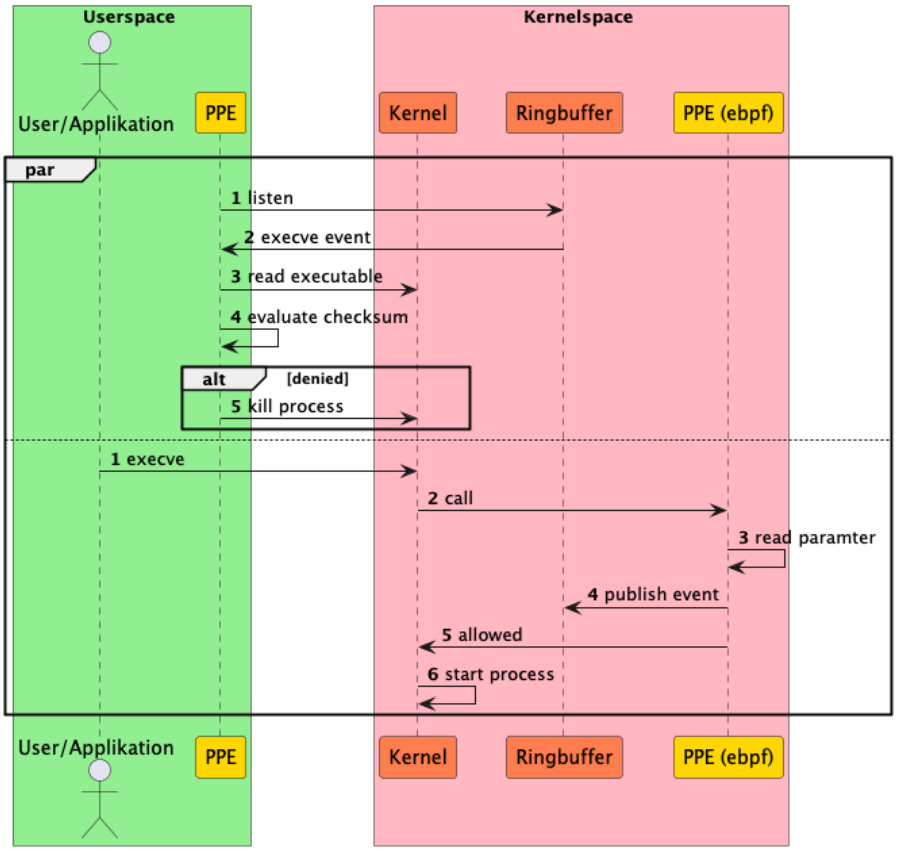


Figure 11.2: Authentication in userspace (Source: own representation)

PPE can also move decision-making into the kernel in an eBPF program. In this case, the eBPF program performs the decision-making, as Figure 11.3 shows. This delays the corresponding syscall, because the hash value of the executable file is calculated in the path of the kernel decision. Not every eBPF program type may block a kernel operation or use helper functions that can sleep. The eBPF verifier and the capabilities of the respective program type limit which operations an eBPF program may execute. To make determining the hash value in the path of the kernel decision possible, PPE must use a sleepable eBPF program. [36] In this special eBPF program, individual helper functions that can sleep are permitted and are therefore suitable for the required check of the file content. PPE depends on suitable kernel mechanisms or eBPF helper functions to determine the hash value of the executable file in kernel context. “PPE (eBPF)” can then compare this hash value with hash values previously stored in a BPF map. If the hash value is missing from the BPF map, “PPE (eBPF)” can prevent the kernel from loading the program image via the return value. In this case, PPE writes the negative decision to the ring buffer. The PPE main program can receive this event from the ring buffer and log it. The central advantage of this variant is that the kernel does not load an unauthenticated program image in the first place. The unauthorized program can therefore perform no activities before PPE would have to terminate it subsequently. As with authentication in userspace, PPE must ensure during authentication in the kernel that the hash values stored in the BPF map match those present in SPIRE. If invalid hash values are present in the BPF map, PPE must remove them so that the eBPF program no longer uses them for authentication.

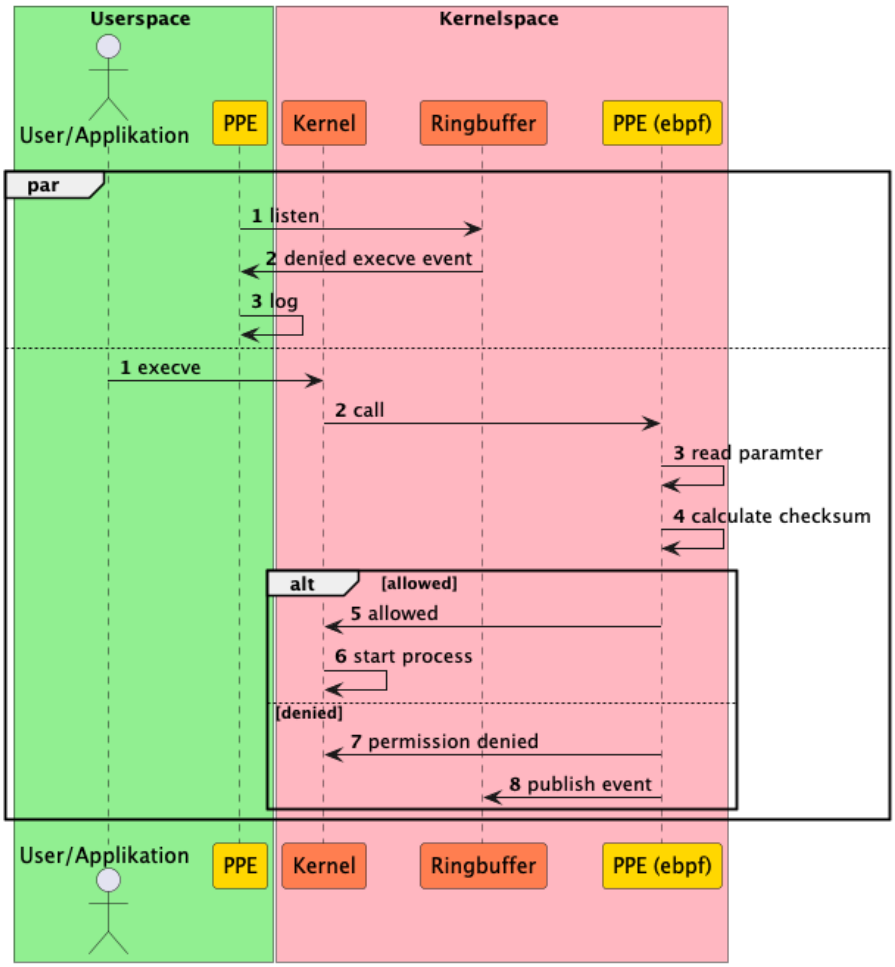


Figure 11.3: Authentication with eBPF (Source: own representation)

Since activities by unauthenticated programs constitute a major security risk, only authentication in the kernel is suitable. PPE deliberately accepts a delay in program executions and, at this point, weights security higher than performance. In the userspace variant, the possible activities of a process before PPE kills it create a problem similar to the Time-of-Check-to-Time-of-Use problem (TOCTOU). Chapter 11.1.4 describes this problem in more detail.

11.2.4 Authorization

Chapter 6.2 already describes tools that resemble PPE or may serve as complements. From an architectural perspective, Tetragon is an obvious complement. SECCOMP is also an option for filtering syscalls. A custom implementation of an authorization mechanism remains as a further option.

For generating authorization rules, the authorization mechanism must also provide a monitoring mode. PPE uses the monitoring mode to observe events during the rule generation phase.

A custom authorization mechanism has the advantage that PPE can design it freely. To control security-relevant kernel operations in kernel context with reasonable effort, eBPF is an obvious choice for such an implementation. This would allow PPE to consider the relevant syscalls or kernel paths individually. At the same time, this variant requires PPE to create, compile, and integrate a suitable eBPF program or attach-point configuration for every supported kernel path.

In addition, every supported kernel path requires a monitoring mode. This entails considerable complexity and maintenance effort.

For SECCOMP to be active, a process must restrict itself independently or start with restrictions inherited from the parent process. SECCOMP does not provide for later restriction of an already running, arbitrary process from the outside. A SECCOMP-based approach therefore requires PPE to control how programs start. One possibility would be a wrapper script. In this wrapper script, PPE could set the SECCOMP profile and then start the actual process. This approach assumes that PPE enforces that every process wishing to execute another program uses the provided wrapper script. In general, this approach therefore appears impracticable.

Tetragon uses policies in YAML format as input. Depending on the tracing policy, Tetragon attaches to LSM hooks, KProbes, tracepoints, or other attach points. Tetragon therefore does not authorize syscalls as abstract units, but the kernel operation visible at the respective attach point. In these YAML files, PPE can configure, based on the file path associated with a process, whether such a kernel operation is permitted. If the operation is impermissible, Tetragon can block access. However, Tetragon blocks only if it has loaded a policy in enforcement mode. Otherwise, Tetragon merely generates events that a consumer can intercept and process. For PPE, Tetragon therefore assumes the task of providing suitable eBPF-based controls at the respective attach points from declarative policies. Tetragon abstracts the creation, compilation, and integration of the required programs. [21] The number of usable selectors in a policy remains limited. This restriction reduces in particular the number of arguments that can be checked and thus influences the possible granularity of policies. However, this limitation is not a Tetragon problem per se, but is based on restrictions for eBPF programs. [37]

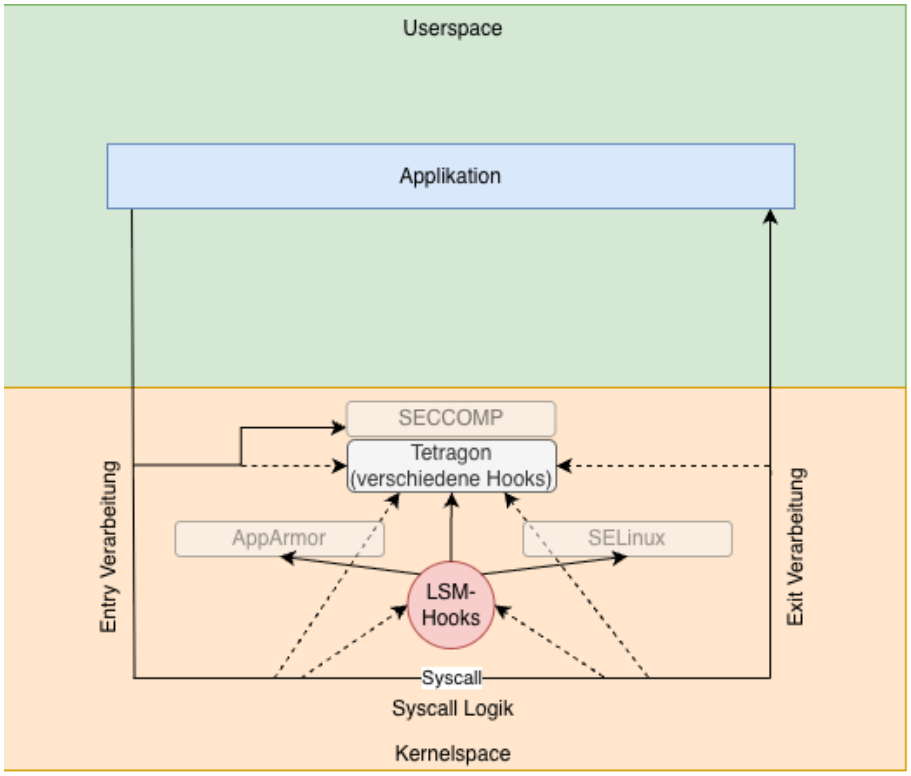


Figure 11.4: Tool selection (Source: own representation)

Figure 11.4 shows that Tetragon offers the broadest range of functions and the greatest flexibility. Accordingly, PPE can rely on Tetragon both for generating authorization rules and for enforcing

them, thereby reducing the complexity that would arise from implementing its own eBPF programs. The SECCOMP approach, by contrast, is more specific and does not fit PPE's approach because control is applied from the inside; PPE therefore does not pursue this approach further.

11.2.5 Relevant eBPF Program Types and Attach Points

PPE implements security functionality using eBPF. The focus is therefore on security-relevant hooks. According to Rice [16], LSM hooks are relevant for this purpose. The advantage of LSM hooks compared with KProbes, for example, is that they influence the kernel's security decision and can therefore permit or deny the operation requested by a syscall. Concentrating on LSM hooks massively restricts the number of available hooks. Nevertheless, at the time of writing this thesis, 267 different LSM hooks were still available on the development machine used. This thesis therefore narrows down the relevant hooks for its own context. The selection concentrates on security-relevant syscalls described by Bernaschi et al. in [38]. Table A.1 lists the corresponding syscalls and a mapping to suitable LSM hooks. A security-relevant syscall is, for example, `chmod`, which allows the permissions of a file to be changed. This syscall can be intercepted by registering at the LSM hook `security_path_chmod`. With an eBPF program, such registration is possible using the attach point `bpf_lsm_path_chmod`.

The use of an LSM hook allows the eBPF program to decide on further processing of the syscall and to block it if necessary [10]. Attaching directly to a KProbe would be less suitable for this, because KProbes primarily serve to observe kernel paths and do not provide the same direct return value for the LSM security decision.

11.2.6 Persistence of Rules

PPE must persist the rules created during rule generation. Different variants are available to PPE for this purpose. It must be noted that authentication and authorization rules exist in different formats. While PPE only has to store the hash value for authentication, it must persist Tetragon policies for authorization.

Both formats can be stored locally on the file system. External storage is likewise possible for both formats. The concrete implementation differs depending on the format. A Git repository, for example, is an option for both formats. For hash values, there is also the option of using the identity management system SPIRE and the underlying SPIFFE framework.

Local storage has the advantage that there are no external dependencies. Linux can also protect the corresponding files locally from unauthorized access, apart from the "root" user. A process running as the "root" user could therefore modify the file.

Different storage solutions are suitable for external storage. Depending on the solution, the storage solution supports a version history, which keeps changes to policies or hash values traceable. To prevent unauthorized access, it must also support an authentication and authorization mechanism. A storage solution that supports both requirements and is particularly suitable for files in text form is the Git version control system together with an external Git service. Versioning is a basic functionality in Git, while the authentication and authorization mechanisms depend on the external Git service. Chapter 11.2.7 describes in more detail which authentication mechanism PPE uses.

In addition to an external storage solution, SPIRE is also suitable for assigning permitted hash values. SPIRE offers workloads the possibility of receiving an external, signed identity, the so-called workload identity, in the form of a JWT token or a certificate after successful

workload attestation. For this purpose, SPIRE stores registration entries that specify under which conditions it issues a workload identity. One such possibility is identifying a file by its hash value. PPE uses SPIRE in this sense to connect permitted hash values with SPIRE registration entries and its own authentication mechanism. Since SPIFFE works with trust domains, PPE can relate the rules to administratively separated trust areas. For example, it can be defined whether PPE accepts a hash value only within a specific trust domain. [39]

PPE uses Git as the storage solution for authorization rules because of automatic versioning. Since SPIRE already provides registration entries and workload identities as security primitives, PPE uses SPIRE to manage the hash-based authentication rules.

11.2.7 Git Authentication

As described in Chapter 11.2.6, PPE relies on SPIRE and Git to store data externally. While SPIRE can issue a workload identity, different authentication options exist for Git services.

Which authentication options are available for Git depends primarily on the external Git service used. PPE must authenticate to the Git service in order to read or write data. It must either store or request the information required for this locally. As described in Chapter 11.2.6, local storage has the disadvantage that not only PPE can read the data. The Git service still offers the advantage of versioning, but PPE loses control over who writes data with the locally stored credentials.

A SPIRE-based extension can avoid local Git credentials by allowing PPE to identify itself with its hash value in order to obtain a workload identity. PPE can use this workload identity to authenticate to the Git service.

Unfortunately, common Git services do not directly support SPIRE. An intermediate proxy can, however, allow PPE to authenticate to the proxy with the SPIRE identity while the proxy handles access to the Git service with an access token for PPE. Implementing such a proxy is not part of PPE and constitutes a separate tool.

PPE deliberately refrains from implementing a proxy. PPE therefore accesses the Git service directly, and there is no proxy encapsulating the accesses. PPE thus accepts the risks arising from local storage of the SSH key. However, PPE prepares the use of a proxy for authentication with SPIRE in order to make it possible in the future to benefit from the shared use of SPIRE for storing hash values and as a source of a workload identity.

11.2.8 Detection of Executable Files

Since Linux provides no unambiguous feature for potentially executable files, PPE must rely on indicators when generating authentication rules. One indicator can be the presence of execute permission. The presence of such permission, however, is not sufficient to classify a file as executable. Likewise, a non-executable file may erroneously have execute permission, while an effectively executable file may lack it. PPE must therefore analyse the content of the file to determine whether it is potentially executable. Executable files have characteristic headers, such as the ELF header. [40] PPE can therefore identify executable files on the basis of their characteristic headers. PPE can implement this identification itself or rely, for example, on the established Linux tool `file`. Based on a file's header, such as the ELF header, `file` can determine the MIME type of a file. MIME types describe a wide range of possible file types using certain rules. PPE can derive from the MIME type whether a file is potentially executable. PPE therefore relies on the established Linux tool `file` to determine a file's MIME type instead of analysing the header itself. [41]

These decisions determine PPE's concrete process.

11.3 Solution Approach

PPE uses a two-stage approach to authenticate and authorize processes. Figure 11.5 shows that PPE first creates the rules and then enforces them in a second step. In this context, a rule specifies either whether a file may be executed or whether a process may trigger certain security-relevant syscalls. The different time axes of the two process steps are not shown in the illustration. Rule generation takes place initially and, if necessary, periodically after updates. Rule enforcement, by contrast, is a continuous process that takes place in parallel with the actual workload of a system.

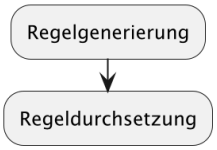


Figure 11.5: Process (Source: own representation)

11.3.1 Rule Generation

Before PPE can enforce rules, it must generate them. PPE supports two types of rule generation for this purpose. For authentication, it analyses static files; for authorization, it examines relevant processes at runtime. Static analysis determines which files on a system are executable and classifies them. Classification includes, on the one hand, unique identification and, on the other, checking a file's origin. By checking the origin of a file, PPE can determine whether there is a trust attestation for a file that matches the trust model and is traceable.

The decisions on rule enforcement in Chapter 11.3.2 determine the format in which the rules must be available. PPE requires different approaches to obtain the necessary information. It must analyse both the files on the system and their runtime behaviour. The analysis of existing files serves to decide, for every executable file, whether its execution should be permitted according to the trust model used. Runtime analysis records which security-relevant kernel operations an executable file triggers.

Because of the broad approach to rule generation, the rules are likely to be too permissive in many practical cases. In such cases, operators must refine them manually or on the basis of other information sources.

11.3.2 Rule Enforcement

As with rule generation, rule enforcement also distinguishes between authentication and authorization. During authentication, PPE decides on the basis of the information obtained through static analysis whether execution of a file is permitted. If execution of a file is prohibited, PPE blocks its execution.

After PPE has permitted execution of a file, the resulting process can access system resources. For this purpose, the process requests system resources from the kernel through the syscall interface. If the process makes unexpected requests, PPE should react accordingly and block the request or terminate the requesting process.

11.4 Architecture

The two-stage solution approach described in Chapter 11.3 is also reflected in the architecture. Static rule generation corresponds to a procedure, while dynamic rule generation and rule enforcement correspond to a daemon.

11.4.1 Static Rule Generation Procedure

To generate authentication rules, PPE follows the process shown in Figure 11.6. PPE scans the file system for executable files by analysing a file's MIME type (Chapter 11.2.8). If PPE finds an executable file, it calculates the file's SHA-256 hash value. Using the SHA-256 hash value, PPE can search Rekor for a matching entry. If PPE finds a matching entry, it can check whether the user considers the author trustworthy. As an additional trust model, the user can define that a matching Rekor entry alone is sufficient to approve an executable file. Alternatively, the user can regard all executable files present on the system at the time of rule generation as trustworthy. Regardless of which trust model a user chooses, PPE creates an authentication rule in the form of a SPIRE registration entry if it classifies the executable file as trustworthy (Chapter 11.2.6). Once PPE has finished scanning the file system, an initial set of authentication rules is available.

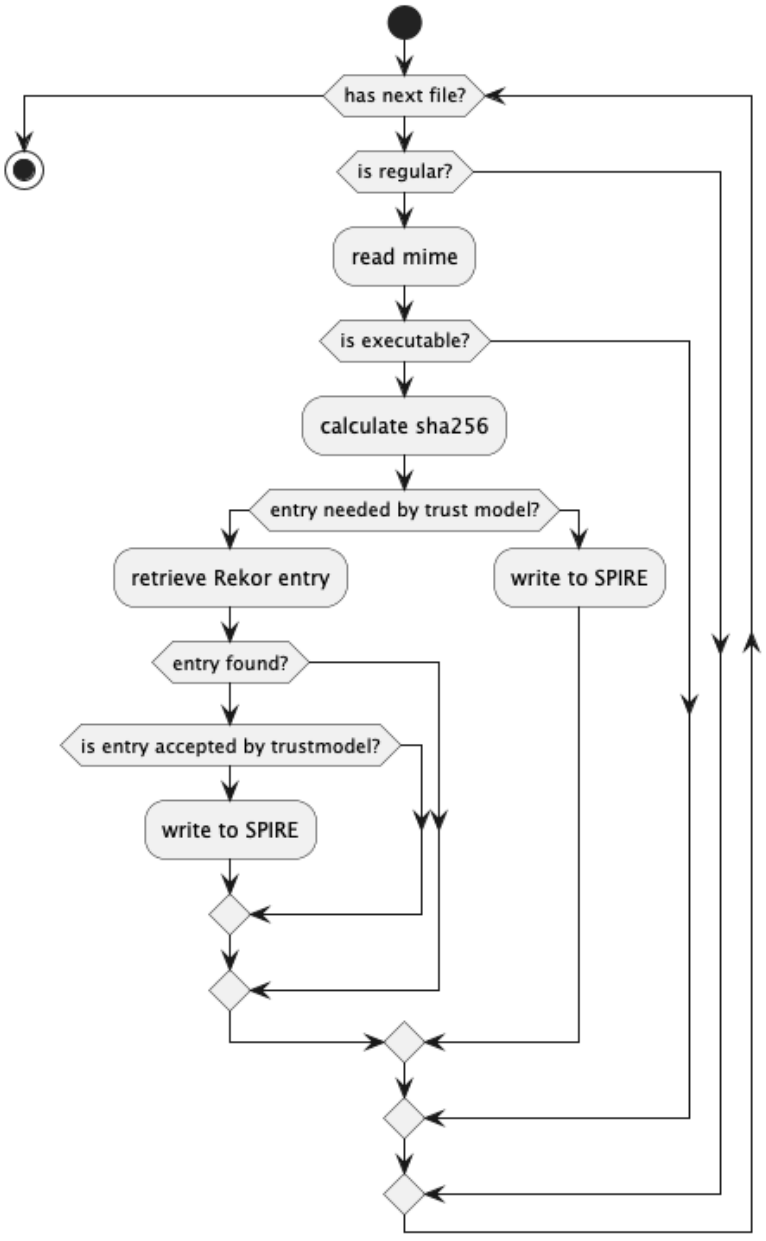


Figure 11.6: Procedure for rule generation (Source: own representation)

11.4.2 Daemon for Dynamic Rule Generation/Rule Enforcement

In contrast to the procedure described in Chapter 11.4.1, generating authorization rules and enforcing rules require a component that runs permanently. For this purpose, PPE implements the daemon shown in Figure 11.7. The daemon can be configured to record authorization rules, merely log any rule violations, or enforce the rules. In addition, the daemon provides a server through which executable files can subsequently be added to the authentication rules, provided they meet the requirements. In the case of such subsequent registration, the same procedure is used as during rule generation.

Figure 11.7 shows how the decisions to use Tetragon for authorization and a custom mechanism for authentication have directly shaped PPE's architecture. When starting, the daemon therefore adds the authorization rules to Tetragon on the one hand and loads the eBPF program for

authenticating the executable files on the other. For this purpose, PPE reads the valid hash values from SPIRE. PPE repeats this regularly to avoid invalid hash values in the BPF map.

As Figure 11.7 also shows, the largest part of PPE runs in userspace. The eBPF program for authentication is the only PPE component that runs directly in the kernel. PPE creates all other eBPF programs through Tetragon.

Which policies PPE loads through Tetragon also depends on the selected mode. When PPE starts in the mode for generating authorization rules, it adds only policies in monitoring mode to Tetragon so that it can process the events registered by Tetragon. PPE and Tetragon communicate through a socket created by Tetragon.

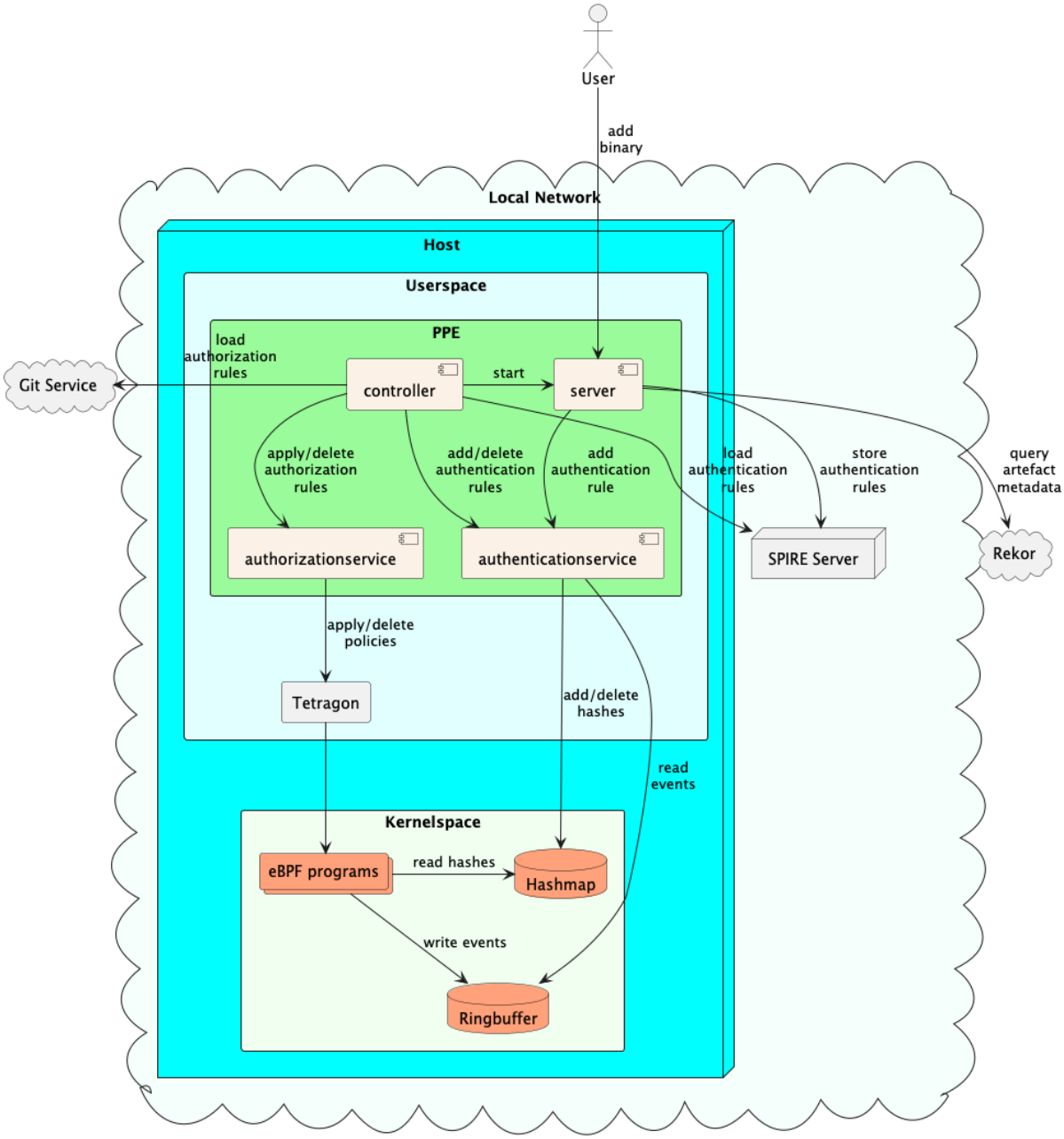


Figure 11.7: Architecture (Source: own representation)

The architecture thus describes how PPE technically combines process authentication and process authorization. Whether this protection is acceptable in practice, however, depends not only on functionality, but also on the overhead caused and on failure behaviour. The following section therefore adds the availability perspective before Chapter 12 examines performance.

11.5 Availability and Failure Behaviour

PPE's availability depends on the phase in which a failure occurs. For rule generation, PPE requires access to the file system, Rekor, SPIRE, Git, and Tetragon in monitoring mode. For enforcement, by contrast, PPE primarily requires the loaded eBPF program for authentication, the populated BPF maps with the permitted hash values, and the enforcement policies loaded by Tetragon. Git and Rekor are not in the critical path during ongoing enforcement. SPIRE and the PPE userspace daemon remain relevant for updates, policy management, and logging, but not necessarily for every individual authentication decision as long as the eBPF program is loaded and the BPF maps contain valid entries.

Table 11.1 summarizes the most important failure cases. The columns on operational decisions and measures against single points of failure describe recommendations for production operation. Not all recommendations are functions of PPE implemented within the scope of this thesis. These recommendations lead to the operational goal of generally running PPE fail-closed in production environments: if PPE cannot load valid authentication or authorization rules, it must not permit new program executions and must not load permissive fallback rules. Fail-open may be useful for pure observation phases or to avoid operational interruptions, but it weakens the protection promise.

Failure case	Effect	Recommended operational decision	Possible measure against SPOF
PPE userspace daemon fails	Loaded eBPF programs can continue to use existing BPF maps; updates, logging, and policy management fail.	Continue in the short term, but raise an alarm.	Service manager with restart policy, health checks, and persisted rules.
Tetragon agent fails or removes programs	New policies cannot be loaded; already loaded enforcement programs may continue to operate or be removed depending on lifecycle and pinning.	Fail-closed for new policy load operations; monitoring alarm.	Run Tetragon as a monitored system service, treat policy load errors as blocking, and check program status.
SPIRE fails	PPE cannot update hash values; already loaded BPF maps may remain usable depending on the TTL implementation.	Do not accept empty maps as a valid state.	Local cache, TTL, SPIRE redundancy, and controlled degradation mode.
Git service unavailable	Authorization rules cannot be read or written.	Use the last validated state; do not publish new rules.	Local read-only cache and versioned releases of policies.
Rekor unavailable	Rule generation cannot check transparency log entries.	Abort rule generation or accept only manually approved hash values.	Cache Rekor queries or operate a private Rekor instance.
BPF maps outdated	New approvals or blocks do not take effect; revoked hash values remain active until the update.	Enforce TTL and raise an alarm on refresh failure.	Periodic reconciliation, atomic map update, and expiry time per entry.
eBPF program or Tetragon policy does not load	Authentication or authorization is not active.	Abort enforcement start.	Startup sequence validates loaded programs and policies before workloads start.
Host restart	BPF maps and eBPF programs are not loaded persistently.	Start PPE and Tetragon before dependent workloads.	Systemd dependencies, persisted rules, and boot health check.
Faulty or empty rules	New program executions may be completely blocked or permitted too broadly; running processes remain affected until further syscalls.	Never implicitly interpret empty rules as permitting.	Schema validation, minimum rules, staging, and rollback to the last valid state.

Table 11.1: Failure behaviour of PPE and possible measures against single points of failure

PPE does not retrospectively evaluate already running processes on the basis of a new hash value. Faulty or empty authentication rules therefore primarily affect new program executions. Authorization rules, by contrast, can affect running processes during later security-relevant kernel operations if the corresponding Tetragon policies are loaded. For a robust protection promise in production operation, this results in the requirement to validate the rule state and to version the last valid state traceably.

Having described architecture and failure behaviour, the thesis evaluates the implemented solution on the basis of the defined goals.

12 Evaluation

This chapter examines how PPE behaves in the implemented test scenarios. The evaluation considers two dimensions: the performance impact and the security-relevant behaviour according to the threat model. The performance measurements show the overhead caused by PPE in synthetic load tests, reference applications, and a microservice application. The security validation then checks, using the risks described in Chapter 9, whether PPE makes the expected decisions.

This chapter describes the test design and results; Chapter 13 then classifies the performance measurements.

12.1 Performance

The performance measurement is intended to show the overhead caused by PPE. Since PPE controls security-relevant kernel operations, the test selection deliberately considers scenarios with high syscall intensity. At the same time, the tests cover different abstraction levels. They show both PPE's influence on individual syscalls and its effects on real applications.

The evaluation does not aim to create a statistically generalizable benchmark study. The focus is instead on the relative comparison of identical test configurations with and without PPE. The measurements are intended to make the orders of magnitude and tendencies of the overhead caused by PPE visible. The evaluation therefore covers several levels: synthetic load tests related to syscalls, individual reference applications, and a microservice application. This breadth is intended to show whether overhead visible in tests related to syscalls is also reflected in more realistic workloads.

All performance tests ran on the same hardware. Table 12.1 lists the hardware specification.

Component	**Configuration**
Processor	Intel Core i5-7400 (4 cores, 4 threads, 3.0 GHz)
Memory	16 GiB DDR3-1600
Mass storage	120 GB Kingston SATA SSD
Network interface	Intel I211 Gigabit Ethernet (1 Gbit/s)
System architecture	x86-64 (64 bit)
Linux kernel	Linux 7.0.0-27-generic

Table 12.1: Hardware specification

Table 12.2 documents the software versions traceable in the measurement artifacts.

Component	**Documented version**
Distribution	Ubuntu 26.04 LTS
Linux kernel	Linux 7.0.0-27-generic
stress-ng	0.20.01
Nginx	1.28.3
Redis	8.0.5
Tetragon	1.7.0
Spire	1.14.6

Table 12.2: Software versions of the test environment

To obtain a baseline, this thesis first ran all load tests without PPE. In a second step, the load tests ran while PPE was in the enforcement phase. The load tests did not consider the generation

phase, because this phase is time-limited and therefore not operationally relevant. For each test configuration, the evaluation reports an independent measurement run over a specific period or a certain number of requests. Observed differences may also be influenced by measurement noise, scheduling, or background activity. The results should be interpreted as indicative measured values and not as statistically validated general statements.

12.1.1 Syscall: stress-ng

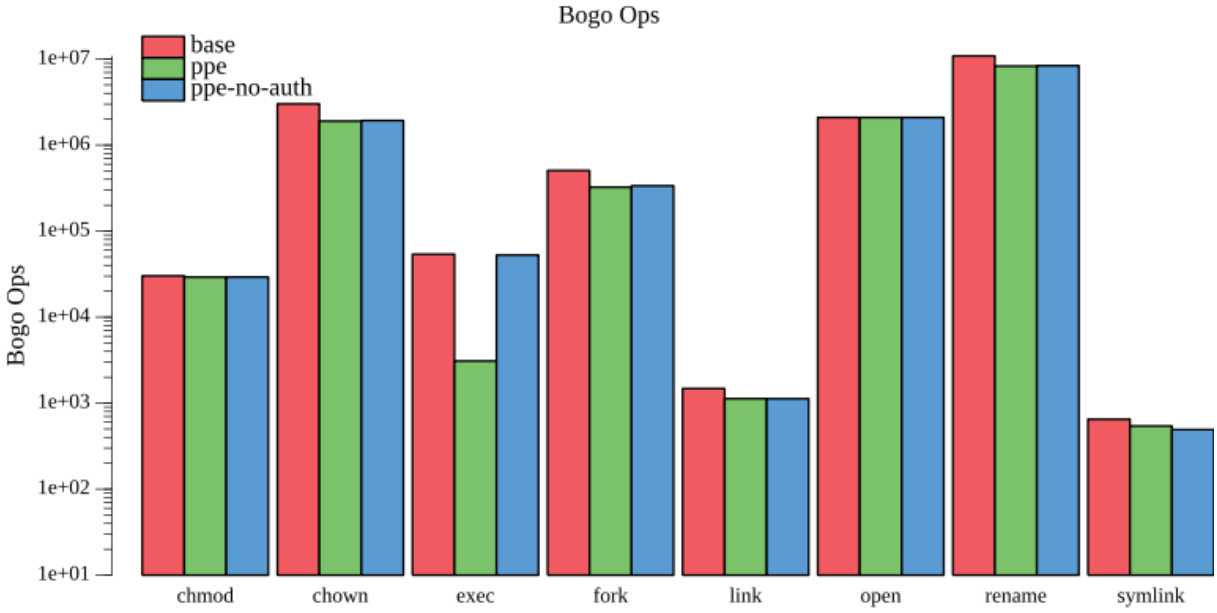
The stress-ng tool generates local load on a system and provides a broad range of stressors for this purpose. A stressor is a type of specific load generator. The broad selection of stressors allows a load test tailored to the kernel paths protected by PPE. [42]

The test configuration used is shown in Table 12.3. Each stressor was started four times in parallel, corresponding to the number of CPUs in the test system.

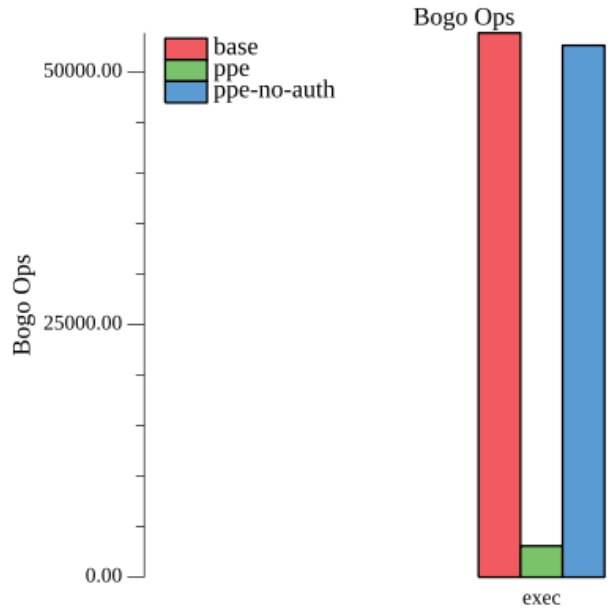
Test property	**Value**
Test duration	60 s
Parallel stressors	4
Operations	open, rename, link, symlink, chmod, chown, fork, exec
Metrics	Bogo Ops

Table 12.3: stress-ng test configuration

Figure 12.1a shows the results of all stressors in the unit Bogo Ops. Bogo Ops is an artificial measurement unit that differs from stressor to stressor. The results can therefore only be compared within a stressor group. To present the large differences between the measured values appropriately nevertheless, the figure uses a logarithmic scale. Since the results for the `exec` stressor are particularly interesting, the results for this stressor are shown with a linear scale in Figure 12.1b.



(a) Bogo Ops (log scale)



(b) Bogo Ops Exec

Figure 12.1: stress-ng benchmark (Source: own representation)

Figure 12.1a makes visible that the stressors reach different orders of magnitude and can therefore only be compared within the same stressor group. Figure 12.1b highlights the `exec` stressor separately because it stresses the `execve` path, which is particularly relevant for PPE, and shows the influence of hash calculation there most clearly.

12.1.2 Reference Applications: NGINX & Redis

As Zhou et al. show in [43], NGINX and Redis are suitable reference applications. Zhou et al. use these applications to evaluate their approach for accelerating syscalls. Unlike Zhou et al., however, this thesis does not expect an improvement in the relevant metrics, but rather a deterioration due to overhead.

Since NGINX is a web server and Redis is a key-value database, the evaluation measures their performance with different tools. For NGINX, it uses the load testing tool Vegeta [44]; for Redis, it uses the Redis tool redis-benchmark [45].

Table 12.4 and Table 12.5 list the exact test configurations.

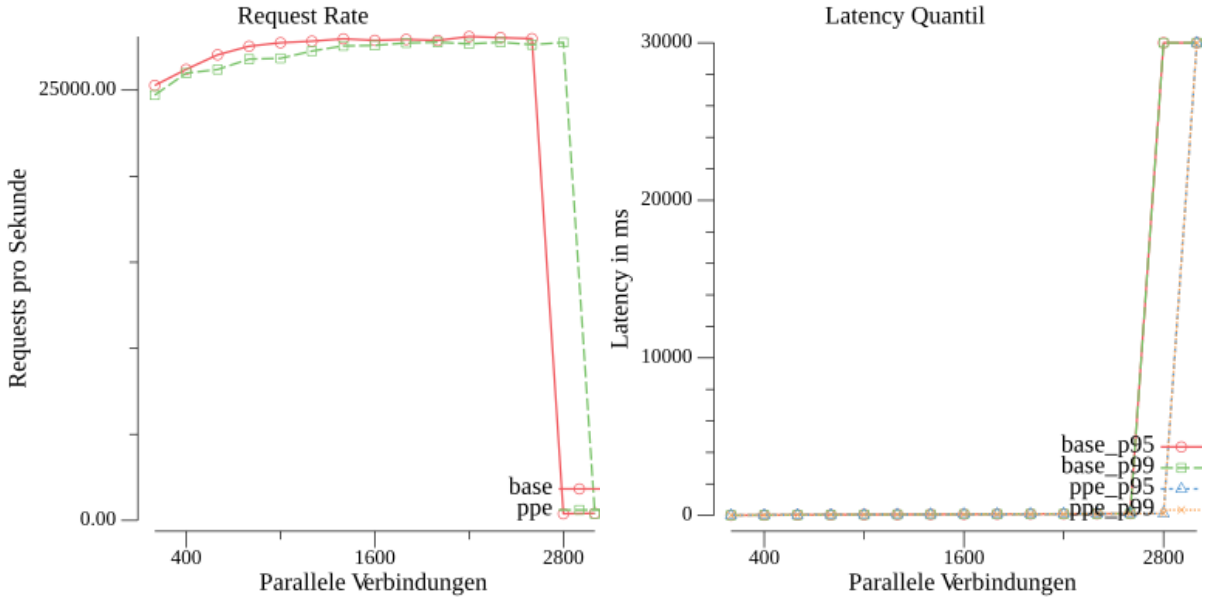
Test property	**Value**
Test duration	60 s
Number of parallel connections	$\{200, 400, 600, \ldots, 3000\}$
Operations	GET
Metrics	Requests per second, success rate, and latency

Table 12.4: NGINX test configuration

Test property	**Value**
Number of requests	1'000'000 per test run
Number of parallel connections	$\{100, 200, 300, \ldots, 1000\}$
Operations	GET and SET
Metrics	Requests per second and latency

Table 12.5: Redis test configuration

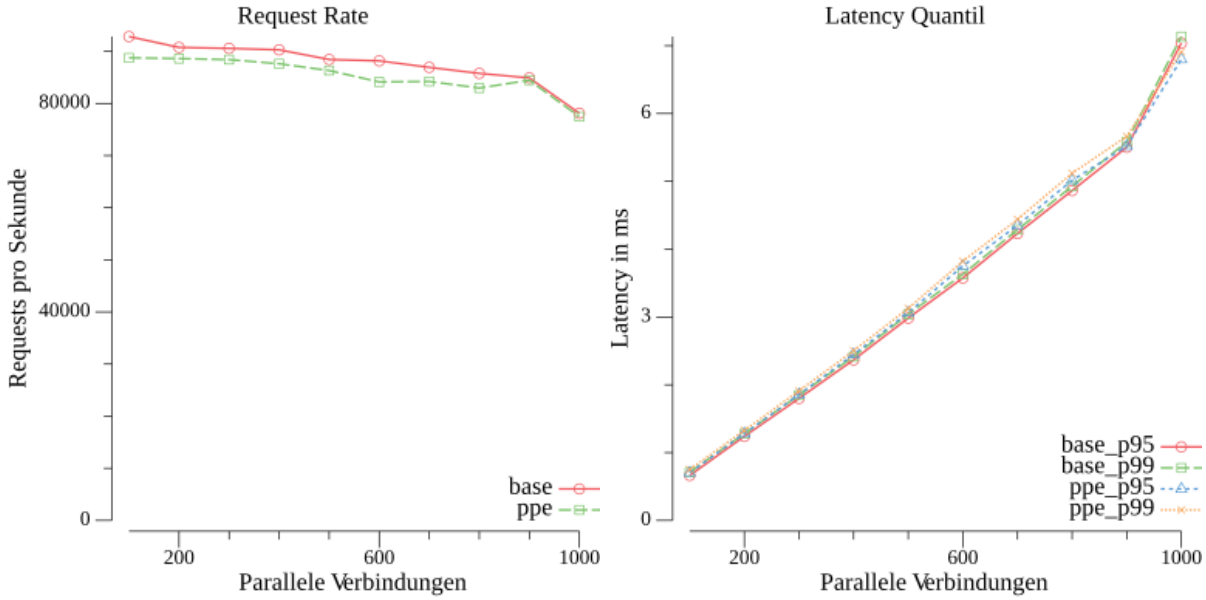
Figure 12.2 shows the resulting measured values for NGINX, Figure 12.3 for Redis GET, and Figure 12.4 for Redis SET. The subfigures show the throughput or rate (Figures 12.2a, 12.3a, and 12.4a) as well as the latency percentiles (Figures 12.2b, 12.3b, and 12.4b).



(a) Rate **(b)** Latency percentiles

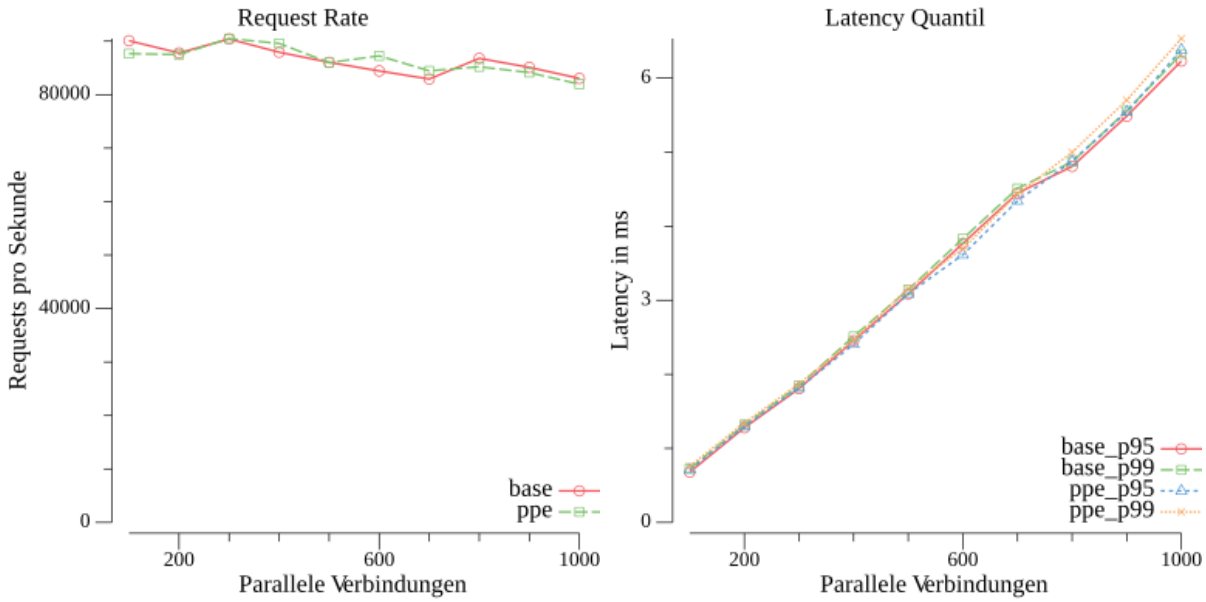
Figure 12.2: NGINX benchmark (Source: own representation)

From 2800 parallel connections onward, the measured values show overload or timeout effects: the success rate drops in individual configurations and the percentile latency is close to 30 s. These points are therefore not suitable for a statement about normal PPE overhead, but mark the overload range of the test setup. Up to 2600 parallel connections, Figure 12.2a shows similar throughput with and without PPE, while Figure 12.2b shows an increase in latencies with increasing parallelism. In this range, the measured values show no clear PPE-specific latency jump.



(a) Rate **(b)** Latency percentiles

Figure 12.3: Redis GET benchmark (Source: own representation)



(a) Rate **(b)** Latency percentiles

Figure 12.4: Redis SET benchmark (Source: own representation)

The Redis measurements in Figure 12.3 and Figure 12.4 show a similar progression for GET and SET. As the number of parallel connections increases, the rate tends to decrease slightly, while the latency percentiles increase. The curves for baseline and PPE remain close together in both Redis tests, so the figures show no pronounced PPE-specific effect.

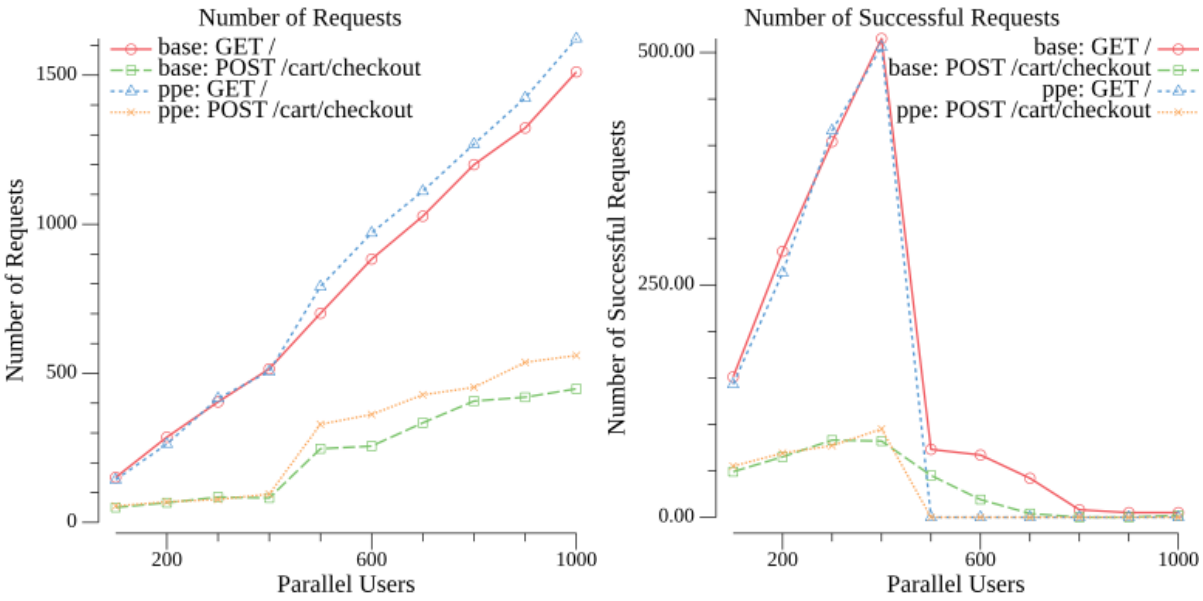
NGINX and Redis consider individual reference services. In addition, Online Boutique examines a distributed microservice application.

12.1.3 Microservice Application: Online Boutique

"Online Boutique" is a microservice demo application by Google [46]. The application represents an online shop consisting of 11 microservices. The demo application is deployed on a Kubernetes cluster. The test system therefore requires a Kubernetes installation. Kubernetes is a container orchestration platform that allows containers to be managed across multiple hosts [47]. Since Kubernetes normally runs distributed across multiple hosts and the hosts partly fulfil different roles, a normal Kubernetes installation is not useful for the present test case. The Kind tool offers a simple alternative [48]. Kind makes it possible to create a Kubernetes cluster with a single command based on a container runtime. Instead of individual hosts, Kind uses containers as hosts.

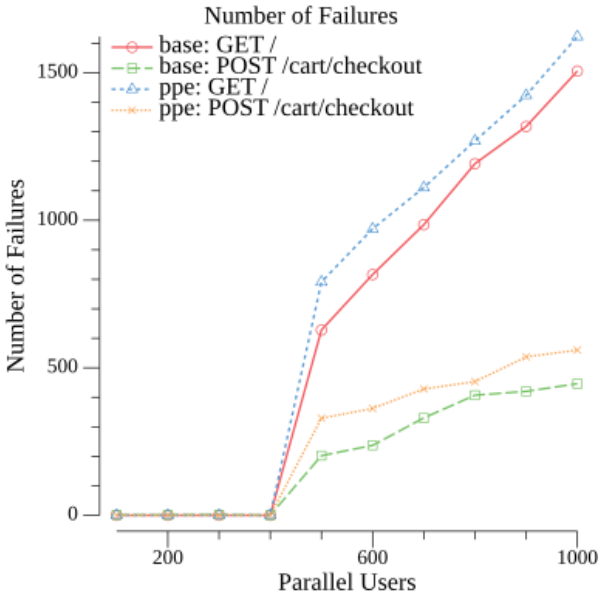
One microservice of the demo application is a load generator based on Locust [49]. Locust allows load tests to be executed and the associated statistics to be recorded. The load generator integrated into the demo application is preferable for the present load test because it already contains different types of requests to the demo application and therefore addresses different components.

Figure 12.5 shows the measured values of the Online Boutique evaluation. The subfigures show the number of requests (Figure 12.5a), the successful requests (Figure 12.5b), and the failures (Figure 12.5c). The representations therefore show not only the total load, but also whether the application processed additional requests successfully or whether these resulted in errors.



(a) Number of requests

(b) Number of successful requests



(c) Number of failures

Figure 12.5: Online Boutique benchmark (Source: own representation)

The Online Boutique measurements show strong overload from around 500 parallel users. Figure 12.5a continues to show many requests, but Figure 12.5b and Figure 12.5c show that failures increase significantly in this range and successful requests collapse. The data therefore do not allow the conclusion that PPE causes no negative overall effect in this workload. The only permissible conclusion is the more cautious finding that the selected measurement in the overloaded range does not separate a statistically validated, isolated PPE effect from the general overload effects of the test system.

The performance evaluation thus provides the measured values required for the later assessment of PPE's overhead. As a supplement to the visualized measured values, Chapter A.4 in the appendix contains the complete tabular measured values in Tables A.2 to A.6. In addition to the

performance impact, the evaluation must also check whether PPE shows the expected behaviour in the security-relevant scenarios.

12.2 Threat Model: Target Environment

The threat model in Chapter 9 yields several security-relevant scenarios that can be used to examine PPE's contribution. Table 12.6 assigns these scenarios to the expected PPE reactions and the remaining residual risks. The table thereby shows which attacks PPE directly addresses and where complementary protection mechanisms remain necessary.

Scenario	**Expected PPE behaviour**	**PPE behaviour**	**Mitigated STRIDE aspect**	**Residual risk**
Legitimate executable file is executed	PPE recognizes the hash value and permits program execution.	✓	No attack; functional capability of the control mechanism.	The trust model must classify the hash value correctly.
Unknown executable file is executed	PPE finds no valid hash value and blocks execution.	✓	Spoofing, elevation of privilege.	A permissive trust model can weaken the protection.
Legitimate executable file is replaced	PPE calculates a different hash value on the next execution and denies execution.	✓	Spoofing, tampering.	PPE does not recheck already running processes.
Process triggers permitted kernel operation	Tetragon permits the operation according to the policy.	✓	Controlled functional capability.	Policies that are too broad can permit unnecessary accesses.
Process triggers blocked kernel operation	Tetragon blocks the operation in enforcement mode.	✓	Tampering, information disclosure, elevation of privilege.	Granularity depends on the available eBPF selectors.
Runtime environment loads code dynamically	PPE authenticates the runtime environment, but not automatically every dynamically loaded code unit.	✓	Partial spoofing.	Runtime environments require additional specific controls.

Table 12.6: Validation of security assumptions

After the functional security validation, the next chapter interprets the performance measurements.

13 Performance Analysis

At first glance, the results of the stress-ng test in Chapter 12.1.1 appear severe. With PPE, performance with regard to the `execve` syscall, as shown in Figure 12.1b, reaches only about 5 % of the baseline. The decisive factor is that the hash calculation in the authentication mechanism is primarily responsible for the performance loss. If only the authorization mechanism with

Tetragon is active, by contrast, the performance loss is low at 2.3 %. As Figure 12.1a shows, the other stressors also indicate that PPE, with and without the authentication mechanism, has a considerable influence on performance. With PPE, the stressors reach only between 63 % and 77 % of baseline performance. The relative performance of the `open` and `chmod` stressors differs from the baseline by less than five percentage points. For `open`, performance is practically identical to the baseline. This suggests that, in the concrete test configuration, the selected stressor did not hit an additionally loaded policy-path combination or that the additional effort lay within the measurement noise of the individual measurement run. Although these performance losses are substantial, the tests intentionally stress precisely those syscalls and kernel paths that PPE and Tetragon use for authentication and authorization. In more realistic performance tests, the performance loss appears less dramatic.

The results for Redis and NGINX show a limited influence of PPE in non-overloaded ranges. For NGINX, Figure 12.2 shows that throughput and latencies up to 2600 parallel connections remain in a similar order of magnitude as the baseline. The measurement points at 2800 and 3000 parallel connections, by contrast, should be interpreted as an overload or timeout range and are not suitable for a normal overhead statement. Redis also shows no clearly negative PPE-specific effect in the measured values for GET (Figure 12.3) and SET (Figure 12.4). The likely reason is that the tools use the kernel paths protected by PPE and Tetragon much less frequently than stress-ng. This reduction means that the negative influence caused by PPE and Tetragon may still be present, but is less clearly visible because of the smaller number of such calls. In particular, the `execve` syscall, which is made much more expensive by PPE with the authentication mechanism, likely plays only a minor role during application runtime.

The performance evaluation of “Online Boutique” cannot be interpreted in the same way. From around 500 parallel users onward, the test system or the demo application is overloaded. The number of successful requests collapses, while failures increase sharply. In this range, the available metrics do not separate a PPE-specific effect from the general overload.

Table 13.1 summarizes the interpretation of the measured values. It distinguishes between effects with a close relation to syscalls and effects in more realistic workloads.

Test type	Observation	Likely cause	Practical relevance
`execve` stressor	Very large performance loss with full PPE.	Authentication calculates the hash value of the executable file during program execution.	Relevant for workloads that very frequently execute new programs.
Other stress-ng stressors	Partly significant overhead under continuous load with a close syscall relation.	PPE and Tetragon intervene precisely in the loaded syscall paths.	Shows an upper load boundary, but does not represent normal server operation.
PPE without authentication	For the `execve` stressor, the overhead remains low.	Without hash calculation, the most expensive part of process authentication is absent.	Separates the influence of authentication and authorization.
NGINX and Redis	In non-overloaded ranges, throughput and latency remain in a similar order of magnitude as the baseline.	The applications call the protected kernel paths less frequently relative to total execution.	Indicates limited overhead in the investigated server workloads.
Online Boutique	Overloaded from around 500 parallel users; the same basic behaviour occurs with and without PPE.	Error rate and successful requests primarily show the load limit of the test system; an isolated PPE effect is not visible.	Realistic example showing that PPE has no directly measurable additional influence in this measurement. However, the measurement does not allow a general statement about all microservice workloads.

Table 13.1: Interpretation of performance measurements

The evaluation thus indicates that PPE can visibly affect system performance when workloads call the protected security-relevant syscalls at high frequency. This applies in particular to the `execve` path controlled by PPE's authentication mechanism. Since many applications do not call the protected kernel paths nearly as frequently during runtime as stress-ng with its stressors, the influence on system performance may be lower in such workloads. In addition to this performance perspective, it remains to clarify how completely PPE achieves the goals formulated in Chapter 4 and which limitations the approach has.

14 Discussion

The example of the "Online Boutique" microservice application illustrates the use of PPE and introduces the subsequent discussion of the approach's strengths and weaknesses. Chapter 12.1.3 describes the application as an interaction of multiple components running in different containers. Although the components run in containers, the associated processes are visible from the perspective of the host's initial PID namespace. One of these components is called "frontend" and starts the process `/src/server` from the perspective of the container's mount namespace. To start this process under PPE, an administrator must first scan the file system for potentially executable files. The administrator must ensure that the executable files required later are available on the system during the scan. In the case of "Online Boutique", this means that the administrator must pull the required container images from the registry before the scan. After completing the

scan, the administrator should check whether SPIRE contains the expected hash values required for operation. If this is not the case, the administrator must investigate why the corresponding executable files do not match the trust model used. If all required executable files match the trust model used, the administrator can start PPE in monitoring mode and then start "Online Boutique". In monitoring mode, PPE records the security-relevant kernel operations relevant for authorization. After an observation period suitable for "Online Boutique", the administrator can stop PPE and then start it in enforcement mode. How long this observation period must be depends on the runtime behaviour of the respective application. If the administrator then attempts, for example, to start `/src/server` from an image updated after the scan whose hash value is not contained in the loaded authentication rules, authentication fails and PPE blocks the program start. For authorization, PPE permits through Tetragon those security-relevant kernel operations that the loaded policies allow. If `/src/server` triggered during the monitoring phase, for example, only file accesses that the policy generated from them permits, PPE can block an operation that occurs for the first time later, such as creating a symlink, if no policy allows this operation. The example thus shows both the general PPE workflow and central prerequisites of the approach, especially the completeness of rule generation and the dependence on the selected trust model. This chapter discusses how well the proposed solution from this example achieves the goals of the thesis, which limitations exist, and in which direction PPE could be further developed.

14.1 Goal Achievement

PPE checks the executable file used when loading a new program image against loaded authentication rules using its hash value. Only if this authentication succeeds and the configured trust model permits the executable file may the kernel load the new program image. PPE does not solve the problem of programs with a runtime environment such as Java or Python. To authenticate the main program started by Java or Python, PPE would have to understand how the respective runtime environment starts a program. PPE could in principle implement this for individual runtime environments. At this point, however, it is more promising to write specific Tetragon tracing policies for the respective runtime environments and, for authentication of Java classes for example, to rely on an approach such as that proposed by Sharma et al. in [19].

For authorization, PPE uses Tetragon, a tool that enforces authorization rules. Because of the limitations of eBPF [16], however, the possible granularity of authorization rules is limited. PPE can restrict the use of security-relevant syscalls, but reaches its limits when restricting the permitted arguments. Validation of particularly critical arguments would nevertheless be conceivable.

Since PPE currently limits itself to syscalls that Bernaschi et al. describe in [38] as especially worthy of protection and to the corresponding kernel paths, PPE does not protect all possible system resources. The architecture allows PPE to extend the kernel operations to be protected without difficulty. In any case, further investigation would have to examine to what extent PPE can or should also restrict the arguments.

PPE provides a way to create an initial rule set for a system for both authentication and authorization rules. PPE does not, however, claim that this rule set is perfect. It is rather a best-effort approach. Users would have to create a more tightly restricted and better-fitting rule set either manually or based on data from a build system.

The performance analysis suggests that the authorization rules have at most a limited influence on system performance in the investigated non-overloaded reference applications. The performance loss due to process authentication is more serious. Since the number of required process

authentications remains limited in many server workloads, this performance penalty may be acceptable.

The approach implemented in this thesis works only on Linux and cannot be transferred directly to other platforms. For other platforms, further investigation would have to examine how access to system resources works there and how possible accesses can be intercepted.

Table 14.1 summarizes goal achievement along the central research questions. It shows which parts PPE has implemented, which evidence supports the assessment, and which limitations remain.

Research question	**Implemented mechanism**	**Evidence**	**Limitation**
Authenticate executable files	Hash value of the executable file loaded in the `execve` path, comparison with authentication rules and SPIRE.	Architecture and rule generation in Chapter 11.4; security validation in Table 12.6.	Applies to directly loaded executable files; dynamically loaded code and later process state remain runtime-dependent.
Authorize processes	Tetragon policies for restricting security-relevant kernel operations.	Architectural decision in Chapter 11.2; syscall mapping in Table A.1.	eBPF and Tetragon limitations restrict argument checking.
Protect system resources	Focus on file system resources and resources with a close syscall relation.	Threat model in Chapter 9; evaluation of individual syscalls in Chapter 12.1.1.	PPE does not protect all possible resources comprehensively.
Report performance impact	Comparison of identical test configurations with and without PPE.	stress-ng, NGINX, Redis, and Online Boutique in Chapter 12; interpretation in Table 13.1.	The measured values are indicative.
Address availability and single points of failure	Separation of rule generation and enforcement, fail-closed operational goal, and failure-case analysis.	Availability analysis in Chapter 11.5; failure-case table 11.1.	This thesis analyses failure cases and operational goals; it does not implement or test concrete operational measures.
Create initial rule set	Static analysis of executable files and dynamic observation of syscalls.	Solution approach and architecture in Chapter 11.3 and Chapter 11.4.	The initial rule set remains a best-effort approach and requires refinement in production environments.

Table 14.1: Traceability of goal achievement

In summary, PPE answers the central research questions partly positively. PPE can authenticate on Linux systems the executable file loaded in the `execve` path using its hash value. PPE can authorize security-relevant kernel operations with the help of Tetragon, but remains limited in argument checks because of eBPF limitations. The approach is particularly suitable for heterogeneous Linux systems and resources with a close syscall relation. The performance effects are clearly visible in microbenchmarks with high syscall intensity. For more realistic server

workloads, NGINX and Redis show smaller effects in non-overloaded ranges; because of overload, the Online Boutique measurement is not robust enough for a strong statement.

14.2 Limitations

Although PPE largely achieves the defined goals, several limitations remain. This chapter examines and justifies them. Table 14.2 documents the not fully achieved goals as technical findings. To this end, the table distinguishes between the selected approach, the observation, the cause, and the consequence for PPE.

Goal	Approach	Observation	Cause	Consequence
Authenticate dynamically loaded code	Authentication at the `execve` syscall.	PPE authenticates the initially loaded executable file or runtime environment, but not automatically every dynamically loaded file.	Loading paths differ depending on the runtime environment.	Runtime environments require specific additional controls.
Protect Kubernetes workloads	Application of the PPE concept to container processes.	Static rules only fit dynamically scheduled containers to a limited extent.	Kubernetes decides at runtime where and how workloads start.	For Kubernetes, image and workload policies are often more suitable.
Use managed Kubernetes	Examination using Azure AKS.	PPE cannot use the required LSM attach points there.	Azure AKS does not allow free configuration of the relevant kernel options.	PPE requires self-managed nodes or less strongly abstracted platforms.
Fine-grained argument checking	Authorization through Tetragon policies and eBPF selectors.	PPE can restrict security-relevant kernel operations, but reaches limits with arguments.	eBPF and Tetragon restrict the available selectors.	Authorization remains limited for complex argument conditions.
Perfect initial rule set	Static file analysis and dynamic observation.	The generated rule set remains a best-effort result.	Complete legitimate runtime behaviour is not always observable.	Production rules require refinement by operators or build data.

Table 14.2: Not fully achieved goals

14.2.1 Runtime Environment

PPE can only authenticate the executable file loaded in the `execve` path. Such a process can also be a runtime environment that dynamically loads code in the form of libraries, scripts, or, in the case of Java, classes. To control this dynamic loading, PPE would have to know for each runtime environment in which form it loads code. If this were the case, PPE could intercept file accesses to corresponding files and authenticate the underlying file. However, because the diversity of possible runtime environments is unlimited, as anyone can theoretically implement their own runtime environment, implementing support for individual runtime environments makes only limited sense. More important is that PPE users permit runtime environments only cautiously or use control mechanisms of the respective runtime environment if available. PPE can control which runtime environments are themselves permitted. This limitation is therefore not merely an implementation gap, but a deliberate boundary of the selected mechanism. PPE controls program execution generically at the Linux level; runtime-environment-specific dynamic loading paths belong in complementary, specialized policies.

In addition to technical limits regarding runtime environments, the selected trust model also limits the significance of authentication.

14.2.2 Trust Model

PPE's trust model is based on a PPE user deciding which signatures in Rekor they want to trust. PPE cannot make this decision because it does not know the user's focus, security requirements, threat model, or risk tolerance. PPE therefore supports various trust models across a spectrum from manually managed hash values to a model in which all Rekor entries matching the trust model are acceptable. Even a time-based approach without using Rekor is possible. Which trust model is correct depends on the PPE user. PPE therefore deliberately does not enforce a universal trust model. This decision keeps PPE usable for different deployment environments, but shifts responsibility for the quality of the trust anchor to the operator.

A further limitation arises where an orchestration layer controls execution and host processes are not directly at the centre.

14.2.3 Kubernetes

PPE can theoretically be operated on a container orchestration platform. Kubernetes, however, starts workloads dynamically on the basis of container images. If the file specified in the container image is not part of the previously generated rules, PPE prevents the container from starting. To avoid this, PPE would have to be active in generation mode on the cluster before the containers start for the first time. Since standard Kubernetes does not guarantee on which system it starts a container, however, the correctness of the rules remains uncertain.

To use PPE's authentication mechanism on Kubernetes nevertheless, dynamic rules would be necessary. PPE would therefore have to check the trustworthiness of the file to be loaded during program execution. However, because this requires an external call, it can only be done in userspace. To avoid the TOCTOU-like problem, PPE would therefore have to block an initial start attempt and then perform the trustworthiness check, so that a second or third start attempt would succeed. In the Kubernetes environment this would in principle be conceivable, because Kubernetes by default attempts to restart a crashed container.

It is even more difficult to create meaningful dynamic authorization rules for containers managed by Kubernetes. These rules cannot be queried externally as in the case of authentication, because they depend on the container's runtime behaviour and static information cannot generate them.

PPE is therefore not necessarily the most suitable approach for Kubernetes. Since Kubernetes controls the processes that it starts based on images, other possibilities are available than for programs executed directly on the host. As a replacement for authentication, Kubernetes could, for example, use Sigstore's Kubernetes Policy Controller and SECCOMP for authorization [50], [51]. The Kubernetes Policy Controller checks whether an image that Kubernetes wants to start is signed. Authentication therefore takes place one level higher than with PPE. However, this layer exists in the case of PPE only for container workloads, which is not guaranteed in PPE's case. In Kubernetes, SECCOMP can be configured directly in the workload configuration. Since Kubernetes controls the container processes, Kubernetes can restrict the processes from the beginning and provide a SECCOMP profile.

PPE therefore addresses heterogeneous Linux systems with programs executed directly on the host. Kubernetes, by contrast, has more possibilities to control the managed workload and enforce security measures because of the additional orchestration layer. The Kubernetes limitation thus arises not primarily from a missing PPE function, but from a different abstraction level. For Kubernetes, it is more useful to control images and workload configurations before start, while PPE plays to its strengths for programs executed directly on the host.

14.2.4 Managed Kubernetes: Azure AKS

As Chapter 14.2.3 already shows, the use of PPE on a Kubernetes cluster is only conditionally useful. The thesis nevertheless examined this case using Microsoft's managed Kubernetes cluster solution. The advantage of clusters managed by Microsoft turns into a disadvantage for PPE: Azure AKS restricts the configuration options of the nodes belonging to the cluster. In particular, as can be seen from the Azure documentation, Azure AKS does not allow free configuration of kernel options. [52] As a result, it is not possible to include `bpf` in the kernel's active LSM list, which is visible via `/sys/kernel/security/lsm`. Without this option, PPE cannot use the required LSM attach points, which makes the use of PPE on Azure AKS impossible in the examined form. To use PPE on Azure, users would have to operate an unmanaged Kubernetes cluster based on virtual machines. This finding clearly delimits PPE's practical area of use. PPE requires control over the relevant kernel options and is therefore only conditionally suitable for heavily managed platforms that deliberately abstract this control.

In addition to these platform limits, the granularity of authorization also remains limited within supported Linux environments.

14.2.5 Argument Checking

PPE can in principle restrict security-relevant kernel operations. Particularly fine-grained authorization would, however, have to check not only the operation itself but also its arguments. This is possible for individual paths, as the protection of individual directories shows. PPE does not achieve generic, arbitrarily deep argument checking with Tetragon. The reason does not lie in the substantive objective, but in the available Tetragon selectors and the restrictions for eBPF programs. PPE deliberately documents this limit because it directly influences which types of policies can be formulated meaningfully in practice. For production use, this means that PPE is particularly well suited to kernel-adjacent and path-based restrictions, while complex semantic rules require additional mechanisms.

These limitations determine how the security gain of PPE should be classified.

14.3 Security Effect

From a security perspective, PPE offers added value because it can prevent the execution of unauthenticated files and restrict possible security-relevant kernel operations. This added value, however, stands in relation to the quality of the rules and the trust model applied. PPE therefore cannot provide absolute security guarantees.

From a security perspective, PPE's greatest weaknesses are that it checks the hash value of the initially loaded executable file only at program execution and that PPE cannot generically check code dynamically loaded by runtime environments. PPE therefore cannot react if a program becomes untrustworthy at a later point, for example because the executable file contains malicious code introduced through a supply-chain attack. Not checking this, however, is a deliberate design decision of PPE. PPE's approach is based on authentication in the path of the corresponding syscall. To reassess already running processes, PPE would have to periodically scan the running processes and check the hash values of the underlying executable files. This would be feasible, but is not the focus of the thesis.

Chapter 14.2.1 already discusses the problem of dynamically loaded code.

14.4 Outlook

PPE offers extension potential at several levels. Future work could expand the scope of protected kernel operations and discuss in greater depth which operations PPE should protect. In this context, it could extend the protection of these operations to checking the arguments. It would be particularly relevant to examine how such protection could be implemented despite the limitations present in eBPF and Tetragon. In addition, it would be interesting to show, using one runtime environment as an example, how dynamically loaded code could be authenticated.

With regard to storing local credentials, future work could extend PPE with a SPIRE-based Git proxy. Such a proxy would allow PPE to authenticate to Git via the proxy using a SPIRE workload identity in order to read and write authorization rules.

Finally, further work could expand the supported storage solutions for authorization rules or develop additional solutions for storing authentication rules.

The discussion has therefore described the goals achieved, the remaining limits, and possible further developments. The conclusion summarizes the thesis's contribution on this basis.

15 Conclusion

With the authentication of the executable file loaded in the `execve` path on the basis of its hash value and the enforcement of authorization rules, PPE shows that the zero-trust principle can be transferred to the process level. The security level depends on the rules and the trust model used. The rules that PPE can generate form a suitable starting point for this. The performance analysis indicates that PPE's process authentication has a measurable influence on performance during program execution. For NGINX and Redis, the influence remains limited in non-overloaded ranges; because of overload and incomplete metrics, the Online Boutique measurement is not robust enough for a strong overall statement.

In a broader context, PPE should be understood as a building block in combination with other security tools. PPE cannot, for example, replace a firewall, just as a firewall cannot fulfil PPE's functionality. Each of these elements is a building block that makes the overall system more secure when users deploy and configure it correctly.

References

[1] M. Lins, R. Mayrhofer, and M. Roland, „Unveiling the Critical Attack Path for Implanting Backdoors in Supply Chains: Practical Experience from XZ“, in *Cryptology and Network Security*, Y. Kim, A. Miyaji, and M. Tibouchi, Eds., Singapore: Springer Nature, 2026, pp. 521–541, ISBN: 9789819544349. DOI: 10.1007/978-981-95-4434-9_24

[2] „Usage statistics and market share of operating systems for websites“, Accessed: Jul. 31, 2026. [Online]. Available: https://w3techs.com/technologies/overview/operating_system

[3] J. Kindervag, „Build Security Into Your Network's DNA: The Zero Trust Network Architecture“, 2010.

[4] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, „Zero Trust Architecture“, National Institute of Standards and Technology, Aug. 11, 2020. DOI: 10.6028/NIST.SP.800-207 Accessed: Nov. 2, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf

[5] „Zero trust cybersecurity: Never trust, always verify“, Accessed: Jul. 31, 2026. [Online]. Available: https://www.nist.gov/blogs/taking-measure/zero-trust-cybersecurity-never-trust-always-verify

[6] M. Penelova, „Access control models“, *Cybernetics and Information Technologies*, vol. 21, no. 4, pp. 77–104, 2021.

[7] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: From I/O Ports to Process Management*. "O'Reilly Media, Inc.", Nov. 17, 2005, 944 pp., ISBN: 978-0-596-55491-0. Google Books: h0lltXyJ8aIC.

[8] „What is kernel in linux?“, Accessed: Jul. 31, 2026. [Online]. Available: https://monovm.com/blog/what-is-kernel-in-linux/

[9] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, „Linux security module framework“, in *Ottawa Linux Symposium*, vol. 8032, 2002, pp. 6–16.

[10] „LSM BPF Programs — The Linux Kernel documentation“, Accessed: Nov. 22, 2025. [Online]. Available: https://www.kernel.org/doc/html/v6.12/bpf/prog_lsm.html

[11] „What are the process states in unix/linux?“, Accessed: Jul. 31, 2026. [Online]. Available: https://medium.com/@marketing_864/what-are-the-process-states-in-unix-linux-a02830db95e7

[12] „Operating systems: Chapter 2“, Accessed: Jul. 31, 2026. [Online]. Available: https://cs.nyu.edu/~gottlieb/courses/2000s/2000-01-spring/os/chapters/chapter-2.html

[13] L. Rice, *Container Security, 2nd Edition*, ISBN: 979-8-3416-2769-7. Accessed: Jan. 4, 2026. [Online]. Available: https://learning.oreilly.com/library/view/container-security-2nd/9798341627697/

[14] „Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces | Toptal®“, Toptal Engineering Blog, Accessed: Sep. 2, 2026. [Online]. Available: https://www.toptal.com/developers/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[15] „Namespaces(7) - Linux manual page“, Accessed: Apr. 16, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html

[16] L. Rice, *Learning EBPF*. O'Reilly Media, 2023, ISBN: 978-1-0981-3509-6. [Online]. Available: https://books.google.ch/books?id=dW-yEAAAQBAJ

[17] B. Gregg. „eBPF“, Accessed: Jul. 31, 2026. [Online]. Available: https://www.brendangregg.com/eBPF.html

[18] H. M. Almohri, D. Yao, and D. Kafura, „Process Authentication for High System Assurance“, *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 2, pp. 168–180, Mar. 2014, ISSN: 1941-0018. DOI: 10.1109/TDSC.2013.29 Accessed: Oct. 26, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/6560050/

[19] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus. „SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java“. arXiv: 2407.00246 [cs], Accessed: Nov. 1, 2025. [Online]. Available: http://arxiv.org/abs/2407.00246 pre-published.

[20] C. Mazzocca, A. Garbugli, M. Armillotta, R. Montanari, and P. Bellavista, „Flexible and secure process confinement with eBPF“, in *Security and Trust Management*, F. Martinelli and R. Rios, Eds., Cham: Springer Nature Switzerland, 2025, pp. 97–109, ISBN: 978-3-031-76371-7.

[21] „Overview“, Tetragon - eBPF-based Security Observability and Runtime Enforcement, Accessed: Jun. 27, 2026. [Online]. Available: https://tetragon.io/docs/overview/

[22] „Seccomp BPF (SECure COMPuting with filters) — The Linux Kernel documentation“, Accessed: Jun. 20, 2026. [Online]. Available: https://docs.kernel.org/userspace-api/seccomp_filter.html

[23] „Systemd.exec(5) - Linux manual page“, Accessed: Jun. 20, 2026. [Online]. Available: https://man7.org/linux/man-pages/man5/systemd.exec.5.html

[24] „Seccomp security profiles for Docker“, Docker Documentation, Accessed: Jun. 20, 2026. [Online]. Available: https://docs.docker.com/engine/security/seccomp/

[25] Z. C. Schreuders, T. McGill, and C. Payne, „Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM“, *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, 19:1–19:28, Sep. 1, 2011, ISSN: 1094-9224. DOI: 10.1145/2019599.2019604 Accessed: Nov. 29, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2019599.2019604

[26] „49.3. Brief Background and History of SELinux | Deployment Guide | Red Hat Enterprise Linux | 5 | Red Hat Documentation“, Accessed: Nov. 29, 2025. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/deployment_guide/rhlcommon-appendix-0005

[27] „What is SELinux?“, Accessed: Nov. 29, 2025. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux

[28] *Sigstore/fulcio*, sigstore, Aug. 5, 2026. Accessed: Aug. 9, 2026. [Online]. Available: https://github.com/sigstore/fulcio

[29] „Overview“, Sigstore, Accessed: Aug. 9, 2026. [Online]. Available: https://docs.sigstore.dev/cosign/signing/overview/

[30] „Verifying Signatures“, Sigstore, Accessed: Aug. 9, 2026. [Online]. Available: https://docs.sigstore.dev/cosign/verifying/verify/

[31] „Threat Modeling Process | OWASP Foundation“, Accessed: Apr. 17, 2026. [Online]. Available: https://owasp.org/www-community/Threat_Modeling_Process

[32] R. Sobti and G. Ganesan, „Cryptographic Hash Functions: A Review“, *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, vol. Vol 9, pp. 461–479, Mar. 1, 2012.

[33] M. Bishop, M. Dilger, et al., „Checking for race conditions in file accesses“, *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.

[34] „Rekor“, Sigstore, Accessed: May 14, 2026. [Online]. Available: https://docs.sigstore.dev/logging/overview/

[35] „Policy Language | Open Policy Agent“, Accessed: Oct. 4, 2025. [Online]. Available: https://openpolicyagent.org/docs/policy-language

[36] J. Corbet, „Sleepable BPF programs“, *LWN.net*, Jul. 7, 2020. Accessed: Jul. 25, 2026. [Online]. Available: `https://lwn.net/Articles/825415/`

[37] „Selectors“, Tetragon - eBPF-based Security Observability and Runtime Enforcement, Accessed: Jul. 26, 2026. [Online]. Available: `https://tetragon.io/docs/concepts/tracing-policy/selectors/`

[38] M. Bernaschi, E. Gabrielli, and L. V. Mancini, „Enhancements to the linux kernel for blocking buffer overflow based attacks“, in *4th Annual Linux Showcase & Conference (ALS 2000)*, 2000.

[39] „SPIFFE | SPIRE Concepts“, Accessed: May 14, 2026. [Online]. Available: `https://spiffe.io/docs/latest/spire-about/spire-concepts/`

[40] „Elf(5) - Linux manual page“, Accessed: Jul. 28, 2026. [Online]. Available: `https://man7.org/linux/man-pages/man5/elf.5.html`

[41] „File(1) - Linux manual page“, Accessed: Jul. 28, 2026. [Online]. Available: `https://man7.org/linux/man-pages/man1/file.1.html`

[42] „Kernel/Reference/stress-ng - Ubuntu Wiki“, Accessed: Jul. 10, 2026. [Online]. Available: `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`

[43] Z. Zhou, Y. Bi, J. Wan, Y. Zhou, and Z. Li, „Userspace bypass: Accelerating syscall-intensive applications“, in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 33–49.

[44] T. Senart, *Tsenart/vegeta*, May 16, 2026. Accessed: May 16, 2026. [Online]. Available: `https://github.com/tsenart/vegeta`

[45] „Redis benchmark“, Docs, Accessed: May 16, 2026. [Online]. Available: `https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/`

[46] *GoogleCloudPlatform/microservices-demo*, Google Cloud Platform, Jul. 9, 2026. Accessed: Jul. 10, 2026. [Online]. Available: `https://github.com/GoogleCloudPlatform/microservices-demo`

[47] K. Penfound, Dagger, C. Nickerson, and Kubeshop. „Production-Grade Container Orchestration“, Kubernetes, Accessed: Jul. 10, 2026. [Online]. Available: `https://kubernetes.io/`

[48] „Kind“, Accessed: Jul. 10, 2026. [Online]. Available: `https://kind.sigs.k8s.io/`

[49] „Locust.io“, Accessed: Jul. 10, 2026. [Online]. Available: `https://locust.io/`

[50] „Kubernetes Policy Controller“, Sigstore, Accessed: Jul. 27, 2026. [Online]. Available: `https://docs.sigstore.dev/policy-controller/overview/`

[51] „Seccomp and Kubernetes“, Kubernetes, Accessed: Jun. 20, 2026. [Online]. Available: `https://kubernetes.io/docs/reference/node/seccomp/`

[52] schaffererin. „Customize the Node Configuration for Azure Kubernetes Service (AKS) Node Pools - Azure Kubernetes Service“, Accessed: Jan. 7, 2026. [Online]. Available: `https://learn.microsoft.com/en-us/azure/aks/custom-node-configuration`

[53] L. Torvalds, *Torvalds/linux*, Jun. 28, 2026. Accessed: Jun. 28, 2026. [Online]. Available: `https://github.com/torvalds/linux`

A Appendix

A.1 Rekor Entry

Listing 2: Rekor Entry

```
apiVersion: 0.0.1
kind: hashedrekord
spec:
  data:
    hash:
      algorithm: sha256
      value: 316
         d712726d857c20744c57cb36aa47cfc79bc0af7e82cebf3780244b654c073
  signature:
    content: >-
      MEUCIHDV/tGLCxxAtLQKC7Q6qNScjNrQdx+O7v2xVQcjsW68AiEAkWr5/
         VgvHVpbs7QFp+CqtP39HkOC2EzsUVMXPXYi0wQ=
    publicKey:
      content: |
        -----BEGIN CERTIFICATE-----
        MIIDEzCCApmgAwIBAgIUIDWk2VqYxtd+IuJxxiF5tflSqkgwCgYIKoZIzj0EAwMw
        NzEVMBMGA1UEChMMc2lnc3RvcmUuZGV2MR4wHAYDVQQDExVzaWdzdG9yZS1pbnRl
        cm1lZGlhdGUwHhcNMjYwNjEyMTEyNTQ4WhcNMjYwNjEyMTEzNTQ4WjAAMFkwEwYH
        KoZIzj0CAQYIKoZIzj0DAQcDQgAEyPZmUqZveCLJazj0rA9PABsygpMYfoM3mPGh
        rPc1KrrqfQXYZAv+TxIswaYVPEqG9uq1XfU+Hl05lX4fHB/bBqOCAbgwggG0MA4G
        A1UdDwEB/wQEAwIHgDATBgNVHSUEDDAKBggrBgEFBQcDAzAdBgNVHQ4EFgQUNfIB
        tmp0bo05YPpfWfzHJsqGqKIwHwYDVR0jBBgwFoAU39Ppz1YkEZb5qNjpKFWixi4Y
        ZD8wQgYDVR0RAQH/BDgwNoE0a3J1bC1zdGFnaW5nQGs4cy1yZWxlbmctcHJvZC5p
        YW0uZ3NlcnZpY2VhY2NvdW50LmNvbTApBgorBgEEAYO/MAEBBBtodHRwczovL2Fj
        Y291bnRzLmdvb2dsZS5jb20wKwYKKwYBBAGDvzABCAQdDBtodHRwczovL2FjY291
        bnRzLmdvb2dsZS5jb20wJQYKKwYBBAGDvzABGAQXDBUxMTE5MzY3MDc5NTcwODEz
        MjczMzgwgYkGCisGAQQB1nkCBAIEewR5AHcAdQDdPTBqxscRMmMZHhyZZzcCokpe
        uN48rf+HinKALynujgAAAZ67lL/9AAAEAwBGMEQCIFZLJNU/gfy8couEj103saDJ
        nDkZrlkgxc3IKOZ1amFoAiAlYADqCRPqLrEtH0bks8cXvJv+1U6ldDtK3tleFkP7
        tzAKBggqhkjOPQQDAwNoADBlAjAdSiEfZ7lr854toj8ooIB0YoMGCJv/sg0I+Pcb
        bRbNcKD7ecr7pkQnXhi/6IspA3UCMQC7AniP8N+S97/NHKDcd2JRvoaCZLVC6le9
        UDy+3qBmfFVnnQ4BFmtwjaKp0dAX8gI=
        -----END CERTIFICATE-----
```

Listing 3: Fulcio certificate of a keyless signature

```
data:
  Serial Number: '0x2035a4d95a98c6d77e22e271c62179b5f952aa48'
Signature:
  Issuer: O=sigstore.dev, CN=sigstore-intermediate
  Validity:
    Not Before: 2 months ago (2026-06-12T13:25:48+02:00)
    Not After: 2 months ago (2026-06-12T13:35:48+02:00)
  Algorithm:
    name: ECDSA
    namedCurve: P-256
  Subject:
    extraNames:
      items: {}
    asn: []
X509v3 extensions:
  Key Usage (critical):
  - Digital Signature
  Extended Key Usage:
  - Code Signing
  Subject Key Identifier:
  - 35:F2:01:B6:6A:74:6E:8D:39:60:FA:5F:59:FC:C7:26:CA:86:A8:A2
  Authority Key Identifier:
    keyid: DF:D3:E9:CF:56:24:11:96:F9:A8:D8:E9:28:55:A2:C6:2E:18:64:3F
  Subject Alternative Name (critical):
    email:
    - krel-staging@k8s-releng-prod.iam.gserviceaccount.com
  OIDC Issuer: https://accounts.google.com
  OIDC Issuer (v2): https://accounts.google.com
  1.3.6.1.4.1.57264.1.24: 0c
     :15:31:31:31:39:33:36:37:30:37:39:35:37:30:38:31:33:32:37:33:33:38
  1.3.6.1.4.1.11129.2.4.2: 04:79:00:77:00:75:00:dd:3d:30:6a:c6:c7
     :11:32:63:19:1e:1c:99:67:37:02:a2:4a:5e:b8:de:3c:ad:ff:87:8a
     :72:80:2f:29:ee:8e:00:00:01:9e:bb:94:bf:fd
     :00:00:04:03:00:46:30:44:02:20:56:4b:24:d5:3f:81:fc:bc:72:8b:84:8f
     :5d:37:b1:a0:c9:9c:39:19:ae:59:20:c5:cd:c8:28:e6:75:6a
     :61:68:02:20:25:60:00:ea:09:13:ea:2e:b1:2d:1f:46:e4:b3:c7:17:bc:9b:
     fe:d5:4e:a5:74:3b:4a:de:d9:5e:16:43:fb:b7
```

A.2 Tetragon Tracing Policy

Listing 4: Tetragon Tracing Policy Enforcement

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: ppe-path-symlink
spec:
  options:
    - name: "policy-mode"
      value: "monitor"
  lsmhooks:
    - hook: path_symlink
      args:
        - index: 0
          type: "path"
        - index: 1
          type: "dentry"
        - index: 2
          type: "string"
      selectors:
        - matchActions:
            - action: "Post"
```

Listing 5: Tetragon Tracing Policy Enforcement

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
    name: ppe-enforce-path-symlink
spec:
    lsmhooks:
        - hook: "path_symlink"
          message: "ppe-enforce-path-symlink"
          selectors:
              - matchBinaries:
                    - operator: "In"
                      values:
                          - "/home/ubuntu/build/ppe"
                          - "/usr/bin/dockerd"
                          - "/usr/bin/kubelet"
                          - "/usr/bin/runc"
                          - "/usr/lib/systemd/systemd"
                          - "/usr/local/bin/containerd"
                          - "/usr/local/bin/containerd-shim-runc-v2"
                          - "/usr/local/sbin/runc"
                matchActions:
                    - action: "NoPost"
              - matchParentBinaries:
                    - operator: "In"
                      values:
                          - "/home/ubuntu/build/ppe"
                matchBinaries:
                    - operator: "Prefix"
                      values:
                          - "/proc/self/fd/"
                matchActions:
                    - action: "NoPost"
              - matchActions:
                    - action: "Override"
                      argError: -13
```

A.3 Security-Relevant Syscalls

Syscall	Kernel function	eBPF attach point	Description
chmod fchmod	`security_path_chmod`	`bpf_lsm_path_chmod`	Changes file or directory permissions
chown fchown lchown	`security_path_chown`	`bpf_lsm_path_chown`	Changes ownership of files or directories
execve	`security_bprm_check_security`	`bpf_lsm_bprm_check_security`	Loads a new program image
mount	`security_sb_mount`	`bpf_lsm_sb_mount`	Mounts file systems
rename	`security_path_rename`	`bpf_lsm_path_rename`	Renames or moves files or directories
open	`security_file_{open, permission}`	`bpf_lsm_file_{open, permission}`	Opens and accesses files
link symlink unlink	`security_path_{link, unlink, symlink}` `security_inode_{link, unlink, symlink}`	`bpf_lsm_path_{link, unlink, symlink}` `bpf_lsm_inode_{link, unlink, symlink}`	Creates or removes references
setuid setresuid setfsuid setreuid	`security_task_fix_setuid`	`bpf_lsm_task_fix_setuid`	Changes the user ID of a process
setgroups setgid setfsgid setresgid setregid	`security_task_{fix_setgid, setgroups}`	`bpf_lsm_task_{fix_setgid, setgroups}`	Changes the group ID of a process
init_module finit_module	`security_kernel_{read_file, load_data}`	`bpf_lsm_kernel_{read_file, load_data}`	Loads kernel modules from a file or memory

Table A.1: Security-relevant syscalls and exemplary corresponding LSM hooks [38], [53]

A.4 Results

Stressor	Base (Bogo Ops)	PPE (Bogo Ops)	Relative performance	PPE without authentication (Bogo Ops)	Relative performance
open	2097136	2097135	100 %	2097136	100 %
rename	10895249	8287047	76.06 %	8384782	76.96 %
link	1479	1127	76.2 %	1125	76.06 %
symlink	650	542	83.38 %	492	75.69 %
chmod	30177	29216	96.82 %	29268	96.99 %
chown	3017167	1900112	62.98 %	1928617	63.92 %
fork	506339	324443	64.08 %	336337	66.43 %
exec	53883	3087	5.73 %	52655	97.72 %

Table A.2: Performance measurement with stress-ng

Connections	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
200	25277.26	12.03	15.97	24717.7	12.48	15.96
400	26210.22	21.52	28.68	25990.64	23.25	28.82
600	27055.47	31.14	41.02	26201.62	34.4	41.37
800	27553.98	41.11	54.33	26806.19	45.03	54.13
1000	27757.22	51.88	67.5	26850.54	55.47	66.72
1200	27847.04	59.82	76.3	27274.04	63.5	80.19
1400	27983.84	67.25	87.42	27574.88	69.01	89.01
1600	27893.77	75.98	99.46	27599.25	80.5	103.73
1800	27949.29	84.23	110.89	27747.54	85.95	112.1
2000	27889.24	92.08	120.03	27789.68	92.79	119.68
2200	28116.28	97.17	125.54	27691.15	102.65	132.72
2400	28052.24	104.36	133.21	27791.0	105.33	135.5
2600	27993.71	114.7	146.64	27642.08	112.68	144.46
2800	387.93	30002.84	30010.65	27777.23	120.41	153.06
3000	410.3	30002.27	30008.39	382.33	30003.44	30021.86

Table A.3: NGINX results with and without PPE (Source: own representation)

Connections	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
100	92833.27	0.66	0.71	88747.68	0.7	0.75
200	90749.87	1.24	1.28	88603.78	1.28	1.33
300	90566.59	1.8	1.85	88410.3	1.86	1.91
400	90283.67	2.37	2.42	87614.01	2.45	2.5
500	88431.41	2.98	3.04	86312.55	3.06	3.13
600	88167.88	3.58	3.63	84126.93	3.75	3.82
700	86923.27	4.23	4.29	84214.78	4.35	4.44
800	85766.23	4.86	4.93	82962.77	5.01	5.12
900	84902.62	5.5	5.58	84449.48	5.52	5.66
1000	78106.08	7.04	7.14	77451.53	6.8	6.91

Table A.4: Redis GET results with and without PPE (Source: own representation)

Connections	Base Rate	Base P95 [ms]	Base P99 [ms]	PPE Rate	PPE P95 [ms]	PPE P99 [ms]
100	90083.59	0.68	0.73	87673.92	0.71	0.76
200	87803.25	1.28	1.32	87494.42	1.3	1.34
300	90403.66	1.81	1.85	90491.18	1.82	1.87
400	87936.05	2.46	2.51	89577.64	2.41	2.46
500	86034.83	3.09	3.14	85994.13	3.09	3.16
600	84434.5	3.77	3.83	87251.66	3.62	3.7
700	82921.49	4.45	4.51	84457.32	4.34	4.44
800	86786.73	4.81	4.87	85211.54	4.88	5.0
900	85102.04	5.49	5.56	84124.1	5.54	5.7
1000	83043.38	6.23	6.33	81965.2	6.38	6.54

Table A.5: Redis SET results with and without PPE (Source: own representation)

Users	GET /						POST /cart/checkout					
	Base			PPE			Base			PPE		
	Req.	Successful	Failures	Req.	Successful	Failures	Req.	Successful	Failures	Req.	Successful	Failures
100	151	151	0	143	143	0	50	49	1	55	55	0
200	286	286	0	263	263	0	66	65	1	69	69	0
300	404	404	0	416	416	0	85	83	2	77	77	0
400	515	515	0	506	506	0	82	82	0	95	95	0
500	701	73	628	791	0	791	247	45	202	329	0	329
600	883	67	816	971	0	971	256	19	237	362	0	362
700	1027	42	985	1111	0	1111	334	4	330	428	0	428
800	1200	8	1192	1269	0	1269	407	0	407	453	0	453
900	1323	5	1318	1424	0	1424	420	0	420	537	0	537
1000	1511	5	1506	1623	0	1623	448	2	446	560	0	560

Table A.6: Online Boutique results for the request types `GET /` and `POST /cart/checkout` (Source: own representation)

A.5 Source Code

The source code for this project is available at:

`https://github.com/fhnw-imvs/fhnw-ppe`