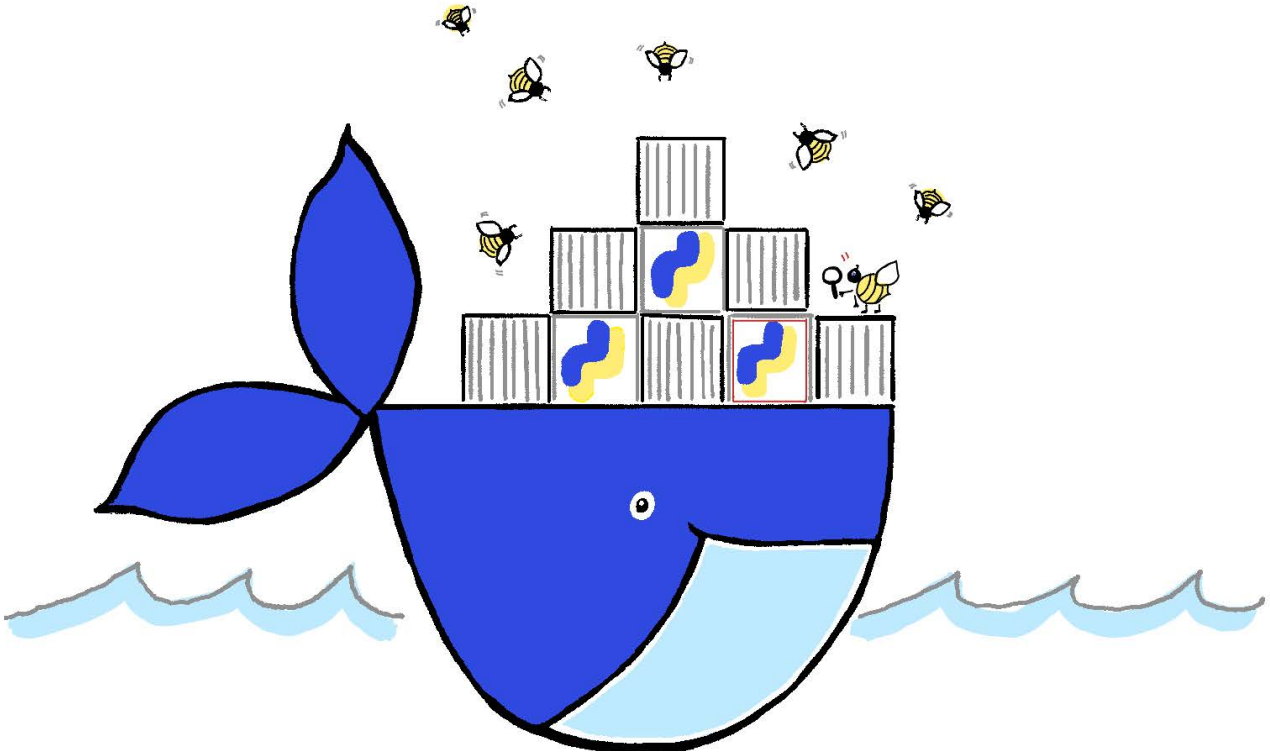
CHALMERS
UNIVERSITY OF TECHNOLOGY



SnakeBPF:
Runtime Python Package Detection

An eBPF-based approach for Vulnerability Prioritization in Containerized Environments

Master's thesis in Computer Systems and Networks

ANNA LITHELL

ALICE THORNELL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

SnakeBPF: Runtime Python Package Detection

An eBPF-based approach for Vulnerability Prioritization in Containerized Environments

ANNA LITHELL
ALICE THORNELL

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

SnakeBPF: Runtime Python Package Detection
An eBPF-based approach for Vulnerability
Prioritization in Containerized Environments
ANNA LITHELL
ALICE THORNELL

Supervisor: Benjamin Eriksson, Department of Computer Science and Engineering
Industrial supervisor: Sathya Prakash Kadhirvelan, Ericsson
Examiner: Andrei Sabelfeld, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Inspired by Docker, Python and eBPF logotypes.
Illustration made by Anna Lithell in reMarkable, edited in Paint.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Abstract

Maintaining awareness of software dependencies is essential for system security, as vulnerabilities in dependencies may introduce significant security risks. Static vulnerability scanning tools often identify large numbers of libraries and packages, making vulnerability prioritization challenging. To improve prioritization, it is valuable to determine which packages are actively used during runtime.

This thesis presents SnakeBPF, a runtime Python package detection approach based on eBPF tracing of interactions with the Linux kernel. Several data collection sources and strategies are evaluated, and the proposed approach primarily leverages `openat` system calls to identify Python packages used during program execution.

To establish an evaluation baseline, multiple alternatives are considered. Ultimately, results from the static analysis tool Trivy and Syft are used to evaluate the effectiveness of the proposed approach. The detection technique is further evaluated using multiple containerized web applications as well as a 5G packet core Kubernetes cluster to assess its applicability in real-world containerized deployment scenarios.

The results demonstrate that information obtained from the `openat` system call can be used to detect Python packages imported during runtime. However, the approach is sensitive to Python's in-memory caching mechanisms, which may result in false negatives when tracing is not initiated during application startup or deployment. With correct initialization, the proposed runtime approach SnakeBPF may complement static vulnerability scanning, by providing contextual information about actively used dependencies.

Keywords: Vulnerability Scanning, Library Detection, Dynamic Analysis, eBPF, Python Package Detection.

Acknowledgements

First, we would like to express our gratitude to the Brick team at Ericsson for the opportunity to conduct our Master's thesis at your Gothenburg site. This opportunity has provided us with insights on how cybersecurity practices works inside large organizations. Thanks to Tanya Balic for brainstorming early project ideas, and to David Carlström for assisting with the technical setup during the evaluation phase. We are especially grateful for our industrial supervisor Sathya Prakash Kadhirvelan, who has provided us with excellent guidance throughout the project. Always eager to help, you've supported us in numerous ways, from understanding the problem in depth to interpreting results while conducting experiments inside the corporate environment.

At Chalmers University of Technology, we would like to extend a heartfelt thank you to our supervisor Benjamin Eriksson from the Computer Science and Engineering department. You have provided invaluable feedback along the way. Thanks for the many brainstorming sessions, suggestions and possible research directions that have made this project truly exciting. Finally, thanks to our examiner Andrei Sabelfeld for his crucial insights and feedback that has helped us develop a well-rounded thesis. We are also grateful for the broader experiences and opportunities this project has provided, giving us a deeper understanding for the research community and the work being done in this field.

Anna Lithell and Alice Thornell, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis. Each acronym is listed in alphabetical order:

API	Application Programming Interface
CID	Container ID
CPE	Common Platform Enumeration
CRA	Cyber Resilience Act
CVE	Common Vulnerabilities and Exposures
DAST	Dynamic Application Security Testing
eBPF	Extended Berkeley Packet Filter
FN	False Negative
FP	False Positive
LCS	Longest Common Substring
OSV	Open-Source Vulnerability database
PID	Process ID
PPID	Parent Process ID
SAST	Static Application Security Testing
SCA	Software Composition Analysis
SBOM	Software Bill of Materials
TN	True Negative
TP	True Positive

Nomenclature

Below is the nomenclature that have been used throughout this thesis.

Sets

$A \cap B$	Intersection of sets A and B
$A \setminus B$	Elements in A but not in B

Contents

List of Figures

List of Tables

1

Introduction

In today's connected world, 5G mobile networks constitute a critical pillar of modern communication infrastructures. As their importance to infrastructure systems increases, these networks are becoming attractive targets for cyber attacks. Common goals of threat actors targeting telecoms networks include the theft of sensitive data, extortion of organizations reliant on uninterrupted connectivity, and the disruption or sabotage of services [1].

To counter these risks, it is essential that telecommunication networks are properly protected and secured. In 2023, new regulations were introduced under the Cyber Resilience Act (CRA) [2]. This EU initiative requires companies to remediate vulnerabilities without delay and prohibits the release of products containing known security flaws. Consequently, the demand for comprehensive security and vulnerability-management tools is stronger than ever.

The increased focus on comprehensive security assessments can also place a significant burden on developers and security professionals, who must analyze large volumes of security findings. This often leads to time-consuming manual analysis, alert fatigue and information overload. To address these challenges, effective prioritization mechanisms are needed to identify the most critical vulnerabilities. By enabling a more structured handling of security scanning results, such approaches can support faster and more efficient remediation of vulnerabilities [3].

1.1 Goal and Research Questions

The goal of this thesis is to develop a method capable of detecting Python packages used by an application at runtime, in order to assess whether any of those packages contain known vulnerabilities. This is achieved by leveraging eBPF, a technology used to observe system calls within a Linux kernel at runtime. As runtime application behavior is characterized by user inputs and actions, experiments are difficult to recreate in an exact manner. This relates to the challenge of establishing a baseline when evaluating the final proposed approach. If the thesis demonstrates that a method relying on dynamic analysis of Python applications can be effectively developed, it may provide a novel approach for securely maintaining software in dis-

tributed systems, thereby contributing to the broader goal of improving the security of critical infrastructure.

To achieve this aim, the following research questions are addressed:

- RQ1: Is it possible to detect vulnerable Python packages at runtime by analyzing dynamic behavior using eBPF?
- RQ2: How does such a runtime detection method handle multiple Python applications running simultaneously?
- RQ3: How does the proposed runtime detection method compare to an existing static analysis method?

1.2 Problem Description

Companies commonly use static vulnerability scanning tools for dependency management, as these tools tend to provide broad coverage and generates extensive lists of detected packages. However, static analysis alone does not detect which packages are actually utilized during runtime under specific application workloads. For vulnerability prioritization, this distinction is important, as vulnerabilities that occur frequently along common execution paths may require more urgent remediation than those that are unlikely to be triggered in practice. Understanding runtime behavior therefore helps developers obtain a more accurate overview of relevant vulnerabilities and prioritize mitigation efforts accordingly.

Additional challenges arise when static analysis is applied in cloud-native environments. In such environments, applications are typically deployed as containers, where each container runs a microservice on top of a base operating system. Static analysis tools also capture programs present at the operating system level, such as package managers like Pip. These programs are often used during the container build process but are not part of the actual application workload at runtime. Consequently, results from static scanning may include packages and vulnerabilities that are irrelevant during execution, thereby introducing noise into the analysis.

Figure 1.1 illustrates an example container that includes several installed packages as well as application code. The application contains two possible execution paths, depending on whether a conditional statement evaluates to true or false. Assume that the condition evaluates to true in 99% of executions. Applying static scanning to this container would then yield five detected packages, no matter which execution path is most likely to be chosen. Among these packages is the package manager Pip, which was used during container creation but is not required during runtime and is never executed.

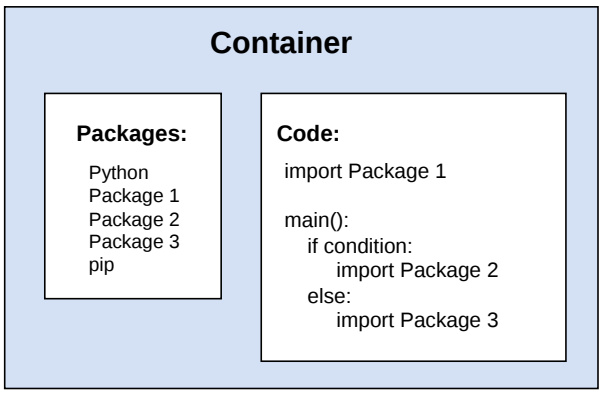


Figure 1.1: Example of a container with packages and executed code.

In contrast, a runtime detection tool produces more precise results. The runtime approach identifies only the packages that are actually executed during program execution. Considering the container example in Figure 1.1, this approach would detect *Python*, *Package 1*, and either *Package 2* or *Package 3* depending on the execution path taken. If *pip* is not used at runtime, it is not included in the results. In this example, vulnerabilities affecting *Package 2* may be more urgent to address, as the corresponding execution path occurs during the most common configuration. In large and complex codebases, it is often not feasible to determine precisely which execution paths are exercised under all workloads, nor which packages are actively used at runtime. Different versions of the same packages may also be used across different software components.

This makes dependency management difficult, especially in large corporate environments. As a real-world example of the potential consequences this might have, the UK National Cyber Security Centre (NCSC) conducted a security evaluation of Huawei equipment in 2020 [4]. Their report highlighted the risks of using outdated and poorly maintained software components. For example, they found an unmanageable number of OpenSSL distributions used in Huawei products, some with known vulnerabilities [4]. These findings indicated unsustainable lifecycle management of third-party software, which was announced as one of the main reasons why the UK Government will remove Huawei products completely from their 5G network by the end of 2027 [5].

Understanding which packages are actually utilized is therefore crucial, as removing unused packages improves security by reducing the attack surface. Some packages may expose the system to a large number of vulnerabilities despite never being used in practice. This includes packages only used in debug mode or during initial build and installation that might contain vulnerable code. What all of these challenges highlight, is the need for a runtime detection approach that identifies packages based on actual execution rather than static presence alone.

1.3 Related Work

A scoping review was conducted to explore whether existing vulnerability detection tools already covered the thesis aim presented in section 1.1. In addition, the review aimed to map the current landscape of existing tools and methods within the area of vulnerability detection. In more detail, the conducted scoping review aimed to answer the following questions:

- SRQ1: Is there an already existing tool or methodology that fulfills the identified research questions in Section 1.1?
- SRQ2: Does a tool or proposed methodology rely on the usage of eBPF to resolve the research aim defined in Section 1.1?
- SRQ3: Are there any existing runtime vulnerability detection tools that relies on the usage of static analysis?

In addition to the stated research questions, key concepts, their related keywords, and synonyms presented in Table 1.1 were used to define search queries for online databases and web-browsers.

Table 1.1: Key concepts, related keywords, and synonyms considered during the scoping review.

Key concept	**Keywords and synonyms**
Vulnerability detection	vulnerability scanning, cybersecurity analysis
Run-time scanning	run-time detection, run-time security, dynamic analysis, non-static analysis
Kernel-level monitoring	eBPF, strace
Microservice architecture	Kubernetes, Containerized applications, Docker

Vulnerable and outdated components is included as part of the OWASP Top 10 for software development [6]. This has drawn recent attention to developing methods for automatization of updating third-party dependencies in order to fix known security issues [7]. For example, Huang et al. utilizes dynamic analysis to detect runtime failures after drop-in replacements of outdated packages [8]. In contrast, Backes et al. has developed a static analysis method resilient to code obfuscation that is capable of detecting package vulnerabilities [9]. However, these efforts does not target the intermediate step of detecting packages with already known vulnerabilities.

In [10], the authors explain that microservice architectures are inherently more vulnerable to software vulnerabilities, as sharing the host's Linux kernel enlarges the overall attack surface. This has caused a surge in developing runtime vulnerability detection tools for containerized applications. The current landscape of exist-

ing vulnerability detection tools is challenging to navigate, as a lot of the tools are closed-source. In the cases of closed-source vulnerability detection tools, proprietary documentation is not available to non-paying costumers. Instead, advertisements, marketing announcements, blog posts, white papers, and release notes were reviewed to infer information relevant to our research questions.

In [11], Minna et. al. classify existing industrial tools used for network visualization and security monitoring of microservice architectures. A compiled list of detection systems were mapped to distinct attributes, such as whether they're open-source or utilize eBPF [11]. Their presented work provided information related to answering SRQ1 and SRQ2. In addition, the OWASP community lists known Dynamic Application Security Testing (DAST) in [12] and Static Application Security Testing (SAST) tools in [13]. These lists has helped further narrow down possible contenders solving the research aim defined in section 1.1. Based on the findings of the scoping review, related existing tools is presented in Table 1.2.

Table 1.2: Comparison of existing security scanning tools, their capabilities and supported features. **Static**: security analysis of source code and/or dependencies without execution. **Runtime**: security analysis consider runtime behavior during program execution. **eBPF**: used for kernel-level monitoring. **Multi-app**: capable of distinguishing multiple applications from one another when running simultaneously. **OSS**: Open Source Software. **Pkg. detection**: Capability to detect package usage per application during runtime.

Tool	Static	Runtime	eBPF	Multi-app	OSS	Pkg. detection
Sysdig[a]	✓	✓	✓	✓	✗	✓
Wazuh[b]	✓	✗	✗	✓	✓	✗
Snyk[c]	✓	✗	✗	✓	✗	✓
Falco[d]	✗	✓	✓	✓	✓	✗
Datadog[e]	✓	✓	✓	✓	✗	✗

[a] `https://www.sysdig.com/`
[b] `https://documentation.wazuh.com/current/user-manual/capabilities/vulnerability-detection/index.html`
[c] `https://snyk.io/product/open-source-security-management/`
[d] `https://falco.org/docs/`
[e] `https://docs.datadoghq.com/security/code_security/iast/`

Based on the reported findings in Table 1.2, the runtime vulnerability scanning tool Sysdig seems to be a promising contender solving our stated research questions (see Section 1.1). Sysdig relies on the open-source tool Falco, and utilizes eBPF to gather kernel space information [14]. However, there is no explicit mention in the Sysdig documentation that the tool is capable of finding packages used during runtime [15]. In August 2025, Sysdig and Snyk announced their partnership, where Sysdig advertised that they will utilize Snyk's runtime capabilities inside their own product [16, 17]. Snyk's runtime capabilities stems from using Software Composition Analysis (SCA), which is a form of static analysis [18]. Based on this information, we

conclude that the Sysdig tool does not provide runtime package detection capabilities without relying on static analysis from Snyk.

In conclusion, there are multiple existing tools that perform runtime vulnerability detection. However, a multipart of those support static analysis as well. Without source-code access we're unable to rule out the possibility of these tools leveraging static analysis to filter the results after running dynamic analysis. In addition, we've found existing tools that utilize eBPF to perform security monitoring in microservice architectures, but no evidence that eBPF is used primarily for the purpose of detecting packages during runtime. Therefore, no existing tools of our knowledge solely perform dynamic analysis to detect vulnerable packages in running microservice deployments.

1.4 Scope and Limitations

To limit the scope of the project, the proposed method is primarily implemented for Python packages. Python is selected because its packages are typically resolved dynamically at runtime, in contrast to statically linked languages such as C++ and Go. The main focus of this thesis is the mapping of kernel-level information (i.e., runtime-level system observations) to Python packages used at runtime. Other components of the system, such as data transfer mechanisms and the mapping between package versions and Common Vulnerabilities and Exposures (CVEs), are not the primary focus of this thesis and are assumed not to directly influence detection accuracy. However, they may impact overall system performance and completeness of results. Consequently, the proposed package detection method is not evaluated with respect to performance and is not subjected to capacity or scalability testing.

As this thesis is conducted in collaboration with an external company, it is also subject to certain limitations regarding reported results and methodology. No vulnerabilities detected within the 5G core environment using the approach proposed in this thesis will be disclosed. Neither the exact set of detected packages is made publicly available. The evaluation is further limited by the absence of a reliable ground-truth baseline. For experiments involving open-source applications, such a baseline could in principle be established through manual source code analysis. However, due to time constraints, this was not performed. This limitation may introduce bias and affect the measured rates of true positives, false positives, and false negatives.

1.5 Thesis Contributions

Although eBPF is utilized by existing vulnerability detection tools today, many does not disclose technical details related to how it has been integrated. This thesis presents a methodology for mapping kernel-level runtime evidence of Python package usage by leveraging traces from selected eBPF programs, and demonstrates how these results can be used to identify known software vulnerabilities. In addition,

this thesis provides a comprehensive evaluation of different eBPF implementations and hook points. The evaluation assesses their effectiveness for package detection under varying runtime scenarios. To the best of the authors' knowledge, such a systematic evaluation has not been previously reported.

2

Theory

This chapter introduces theoretical concepts as a technical foundation for the thesis. A brief overview of current vulnerability scanning techniques used today are provided, followed by methods for uniquely identifying software and correlating them with discovered vulnerabilities. This chapter then focus on the Linux operating system, showcasing common system calls and their capabilities. After introducing system calls and how they can be observed using kernel instrumentation mechanisms, eBPF is addressed as a promising technology to achieve runtime vulnerability scanning. Finally, how the Python programming language defines packages and handles module imports is uncovered as well as a short introduction to dynamic library loading in C.

2.1 Vulnerability Scanning Techniques

NIST define vulnerability scanning as techniques used to identify host(s) attributes and associated vulnerabilities [19]. When performing vulnerability scanning, different approaches can be applied. These approaches are typically categorized as static or dynamic analysis. Static analysis examine a program's source code without executing it [20]. This approach detect potential vulnerabilities by searching for libraries with known vulnerabilities or insecure coding constructs. Since the program is analyzed without execution, the results consider all possible execution paths. Static analysis can be further categorized into techniques such as Static Application Security Testing (SAST) and Software Composition Analysis (SCA) [21].

SAST perform static analysis to detect vulnerabilities present in the source code, using techniques like data flow and taint analysis [22, 13]. On the other hand, SCA analyzes a software product's dependency management. This analysis results in a Software Bill of Materials (SBOM), which is a list of all integrated software components, systems and packages [23]. Under the EU's Cyber Resilience Act (CRA) regulation, SBOMs are required to be included upon the release of any software product to facilitate more effective vulnerability detection and management [2].

In contrast to static analysis, dynamic analysis identifies properties of a program when it is executed and actively running. The purpose of dynamic analysis is to

correlate any input and output changes with the internal program behavior [24]. One such technique is called Dynamic Application Security Testing (DAST) [21]. DAST identifies both new and known security vulnerabilities in typically web-based applications through runtime penetration testing [12].

2.1.1 Software Composition Analysis Tools

As described in section 2.1, there are multiple static analysis techniques. Since this thesis project primarily focus on dependency management related to the usage of third-party Python packages, as well as comparing dynamic analysis with static analysis, this subsection presents two existing SCA tools in more detail.

Trivy[1] is a open source static security scanner from aqua security. It provides extensive support for multiple target types. Trivy categorizes targets into two categories, pre-build and post-build targets. Pre-build targets include code-repositories and for these target types Trivy will also analyze build files, e.g lock files and requirements files. For post-build targets, such as container images, the file system is inspected instead. For Python, Trivy supports multiple package managers and formats [25]. As an example, Trivy searches for the `site-packages` Python directory during the scanning process. Depending on the package format, Trivy access different metadata files in order to identify packages [26].

Syft[2] is an open-source tool used at Ericsson to generate SBOMs. It supports multiple software targets, such as container images, directories and simple files. In this project, it will be used for scanning container images. When scanning an image, Syft will identify a image registry and container platform specified in the image reference [27]. After scanning, Syft outputs a human-readable text file containing identified package names and versions [28].

2.1.2 ZAP Web Application Scanning

Traditionally, web scanning is a black box vulnerability detection technique for web applications. **ZAP**[3] (Zed Attack Proxy) by Checkmarx is one such web scanning tool. The tool is free, open source and claims to be the world's most widely used web app scanner [29]. The process of web app scanning can be divided into three steps: configuration, crawling and scanning [30].

The configuration step includes defining scanning parameters as well as the scanning entry point, i.e the URL of the web application to scan. The crawling stage is where the web application structure is mapped. The goal is to discover all pages of the application and is achieved by following all links descending from the entry point URL.

During the final scanning step, all pages discovered during the crawling phase are

[1] https://trivy.dev/
[2] https://github.com/anchore/syft
[3] https://www.zaproxy.org/

visited. The scanner performs actions on each pages, such as button clicks and sending forms to simulate a real browser user. The web scanning process is performed while the web application is running, and is therefore a type of dynamic analysis. The responses from the web application are then analyzed for vulnerabilities, according to the defined vulnerability policies [30].

2.1.3 Identifying Software Vulnerabilities

The **CVE** (Common Vulnerabilities and Exposures) program is a joint international effort to catalog and report vulnerabilities [31]. Many attack surface management platforms offer functionality to search for a CVE identifier associated with a CPE or PURL identifier [32]. The **CPE** (Common Platform Enumeration) naming scheme can be used to map a distinct identifier to any existing software component [33]. Similar to a social security number for a person, each software component (along with its version) is assigned a unique CPE identifier, which can be used to report detected vulnerabilities. **PURLs** (Persistent Uniform Resource Locators) are permanent identifiers for web applications. By using CPEs and PURLs, security teams can efficiently map identified CVEs with affected software, aiding the process of detecting and mitigating known vulnerabilities.

2.2 Linux

Monolithic operating systems (OS) like Linux provides a user with a wide range of general purpose functionalities. The privileged role of an OS kernel provides desirable observability capabilities, useful in security applications [34]. However, introducing new code inside the OS kernel while ensuring high standard in regards to both security and stability is not easy, due to the complex codebase consisting of tightly coupled components. Luckily, instrumenting the kernel to obtain runtime Python package usage can be done in other ways.

In this section, core concepts related to obtaining runtime information from the Linux kernel will be presented. First, we need to understand processes (Section 2.2.1), their structure and when they're created in order to later correlate which application has used a specific Python package. To detect packages at runtime, system calls (Section 2.2.2) will be important to evaluate. Finally, different kernel instrumentation (Section 2.2.3) mechanisms will be introduced.

2.2.1 Processes

In Linux, a process is an instance of a running computer program [35]. It is a distinct entity the kernel is capable of allocating requested system resources for, such as memory usage and CPU time. Each process is represented by a `task_struct` data structure, which keeps track of the CPU specific context of the process, for example its level of priority and whether it's currently running on a CPU [35]. It is also the `task_struct` that keeps track of stack pointers, registers, and pointers to files used by the process [36].

Process Types

Processes are said to run in the foreground or the background. Simply explained, processes that require user interaction typically run in the foreground, while background processes run independently of a user [37]. A special type of background processes are daemons. These processes run unattended and are available at all times until they're stopped by a root user.

Process Identifiers

The kernel assigns a unique identifier to a process when it's created, known as the process ID (PID) [36]. The PID of the first process created during system boot is 1, and from there the PIDs assigned typically increase sequentially by one. A PID is unique throughout a process' lifetime within a certain namespace, which is an abstraction layer created by the kernel to isolate related processes and their system resources in use [38]. In practice, this means that processes are only aware of other processes residing in the same namespace. Once a process has terminated, its PID can be reused and assoicated with a new process.

In order to determine whether a process has *read, write* or *execute* rights to a file, the `task_struct` contain a `group` vector which determine the access rights of the process. These rights depend on the user identifier (UID) and group identifier (GID) of the user that has invoked the process [36].

Process Creation

All processes running on a system are clones of another process, known as the parent process [36]. However, there are two exceptions to this rule. The two processes with PIDs 0 and 1 are created directly by the kernel [35]. Process 1 is known as the `init` process, which is the process all other processes are related to. Because of this, the relationships between processes are ordered in a hierarchical structure, where each process keeps a pointer to its parent. A process also keeps track of its parent's process ID, known as the PPID. Similarly, the parent process keeps pointers to all of its children processes, since it might clone multiple times.

The /proc Folder

In Linux environments, the `/proc` file system provides an interface to obtain information about the system at runtime. This folder contains general information as well as process-specific directories. There is also the *self* folder, which is the process specific folder for the process reading the file system [39].

For this thesis project, the subdirectory *root* and the files *maps* and *status*, located in each process specific directory, will be especially interesting and are depicted in Table 2.1. The *root* subdirectory contains the root directory of the process. The *maps* file contain information about mapped memory for executables and library files. It also keep track of the object's address and associated filename. Finally,

the *status* file contains information about the process' status, e.g process id, process name, parent id etc [39].

Table 2.1: Subdirectories and files under process specific directories in the /proc filesystem.

Subdirectory/file	Description
root	The root directory of the process.
maps	Executables and libraries mapped in memory of the process.
status	Status of the process

2.2.2 System Calls

Whenever a process would like to request a resource from the kernel, it must first invoke a system call (syscall). Syscalls are the fundamental interface between user-space applications and the Linux kernel, through which the kernel expose specific services to running processes [40]. Under the hood, syscalls are assembly instructions, tasked to identify the type of action requested, trigger a kernel mode switch, and retrieve the results after completing the requested action [41]. Because of this, syscalls are generally invoked via wrapper functions defined in system libraries (e.g. libc) [41]. In Table 2.2, descriptions of common Linux syscalls relevant to this thesis are presented.

Whenever a user-space process wants to access a file, an operation that is often required in order to use a Python package, it will likely invoke the `openat` and `read` system calls. If repeatedly used, this may introduce overhead, as a context switch must occur in order to copy data from the kernel to user address space [42]. To solve this problem, files that are frequently used by many processes can be mapped into the process address space using the `mmap` system call [42]. This limits the need for repeatedly requesting file access through system calls by user-level processes, as mappings can be passed down to other processes whenever the `fork` system call is invoked.

2.2.3 Kernel Instrumentation

There are a various instrumentation mechanisms one can use to observe the Linux kernel. Before understanding how these mechanisms operate, common tracing terminology used to describe these mechanisms are defined in Table 2.3.

Static Tracepoints

Static tracepoints are predefined instrumentation points situated at specific code locations inside the kernel [45]. They provide the capability of recording data at a specific kernel site for later retrieval, and are optimized to have minimal performance impact [45, 44]. The sites have been specifically chosen by kernel developers to enable meaningful insights from important and common kernel functions, such as task scheduling and memory allocation [45]. Since tracepoints are predefined by

Table 2.2: Common Linux syscalls and their purpose.

Syscall	Description
execve	Given a program referred to by *path*, execve replaces the current running program with the new program.[1]
openat	Opens a file specified by *path* and return a file descriptor *fd*.[2]
read	Reads from file descriptor *fd* into the buffer starting at *buf*, until a limit of *bytes* is reached.[3]
mmap	Maps a file, referred to by its file descriptor *fd*, into the virtual address space of the calling process.[4]
clone	Create a new child process, and invoke a separate namespace for the child.[5]
fork	The calling process duplicates itself to create a new process, including a replica of its entire virtual address space.[6]
vfork	A special case of the syscall clone, it creates a new process without copying the parent page tables.[7]
fstat & newfstatat	Retrieve information about a file.[8]

[1] `https://man7.org/linux/man-pages/man2/execve.2.html`
[2] `https://man7.org/linux/man-pages/man2/openat2.2.html`
[3] `https://man7.org/linux/man-pages/man2/read.2.html`
[4] `https://man7.org/linux/man-pages/man2/mmap.2.html`
[5] `https://man7.org/linux/man-pages/man2/clone.2.html`
[6] `https://man7.org/linux/man-pages/man2/fork.2.html`
[7] `https://man7.org/linux/man-pages/man2/vfork.2.html`
[8] `https://man7.org/linux/man-pages/man2/stat.2.html`

Table 2.3: Definition of Linux tracing terms.

Tracing term	Description
Hook	A location in the kernel that act as a break point to pause or redirect the kernel execution flow in order to run custom code [43].
Tracepoint	A specific location in application code that provides a hook to invoke a probe [44].
Probe	Function that is hooked to a specific tracepoint. If the tracepoint is enabled, the probe is called whenever the tracepoint is encountered during runtime [44].
Event	Occur when a tracepoint is encountered during runtime. Depending on the tracing mechanisms, events can represent a specific location in the code, or have a logical meaning such as a context switch [44].

kernel developers, they tend to remain stable across different kernel versions [46]. However, because their placement is limited to specific locations, not all kernel functionalities can be traced using them.

Kprobes

A kernel probe (kprobe) is a mechanism used to dynamically instrument and observe the behavior of the Linux kernel [47]. They can be deployed at any location, inside the Linux kernel or within application code. To do so, kprobes dynamically attach hooks inside the kernel code at runtime by using a trap-based approach [47]. After a kprobe has been registered, it creates a copy of the function to be probed and replace the first bytes of the probed function with a breakpoint instruction. Once the breakpoint instruction is hit, a trap occurs, saving the CPU's registers. The kprobe is invoked and pass the address of the kprobe struct, along with saved registers, to a dedicated handler.

This handler is a user-specified instrumentation function, which observe the kernel's behavior. Once the handler is active, the kprobe note the machine's state for each executed instruction. Since this interrupts the usual execution flow of the kernel, kprobes must retain the stack frame, register set and instruction pointers from before the trap occurred. Once the tracing is terminated, the kernel's execution path will continue as before.

Uprobes

User-space probes (uprobes) can be used to probe user-space applications. Uprobes does not operate within the kernel, but instead trap into the kernel using the `int3` instruction [48]. Two context switches are required because of this, one into and one from the kernel, which introduce additional overhead compared to kprobes. Still, uprobes are useful for non-intrusive application tracing, as they do not require recompilation of user-space application code in order to probe a specific event [48].

2.3 Containers

A container is at its core one or more processes isolated from the rest of the system [49]. In order to achieve this, containers have a separate namespace and cgroup. The namespace allows containers to virtualize system resources, such as the filesystem, while the cgroup is used by the kernel to limit system resource usage, such as CPU and memory [50]. Containers are commonly used to overcome obstacles related to deploying applications on different systems and configurations, especially in cloud native environments [49].

The first step of deploying a container is to build a container image [51]. This image includes all files necessary to run the container, including dependency and environment metadata. In order to run the container, a high-level container platform can be used to download the image and unpack the files into a runtime file system bundle. Finally, a designated runtime process run the bundle and deploy the container.

2.3.1 Container Lifecycles

On a low-level, there are multiple processes involved to run a container. The `runc` process is called the container runtime, contributed to the Open Container Initiative (OCI) as a reference runtime specification [51]. `runc` can also be used as a stand-alone command-line tool, providing a mechanism to run containers. However, it does not handle downloading or unpacking container images.

These are the responsibilities of a container lifecycle manager, such as `containerd`. `containerd` is a long-lived daemon process responsible for image transfer and starting container runtimes [52]. Between `containerd` and `runc` resides a `containerd-shim`, which aides communication between the two parts and allows runtimes to survive if the manager needs to restart.

2.3.2 Container Platforms

Instead of communicating directly with low-level processes, container platforms such as Docker and Podman can be used. They provide user-facing interfaces for building and running containers. Docker expose this functionality through a second daemon, known as `dockerd`. `dockerd` provide useful workflow commands such as `docker build` for building container images, and `docker run` for running them. Under the hood, `dockerd` delegates a user's commands to `containerd`, responsible for actually managing the containers [52].

Podman on the other hand does not rely on daemons for container management. Instead, it provides a library called `libpod`, used to manage container images and lifecycles. The advantage with this approach is that while background processes like daemons need root permissions to run, Podman utilizes Linux user namespaces to obtain rootless, unprivileged containers [52]. This is an important distinction from a security standpoint, as privileged containers pose a notable risk. If an attacker manage to exploit a vulnerability in a workflow running on a privileged container, it can then escape the context of the container and gain root access on the host system [53]. With Podman, this type of attack is far less feasible.

2.3.3 Container Orchestration with Kubernetes

Kubernetes is a container orchestration platform used to deploy and manage distributed systems in a resilient manner [54]. A Kubernetes cluster consists of a control plane and a set of worker nodes [55]. The worker nodes are physical or virtual machines that host pods and are managed by the control plane. Pods are the smallest units in Kubernetes and contain one or more containers that execute parts of a distributed application [56]. The pods, and consequently the containers, interact with the worker node kernel through a container runtime.

One of the main advantages with Kubernetes is that it automatically handles failures within the cluster. If a node dies or a pod fails, Kubernetes reschedules the task to another available node [57]. This way, the system will maintain availability

with minimal downtime. Figure 2.1 illustrates a very simplified Kubernetes cluster architecture. For this thesis, the primary interest lies in the fact that containers located on the same worker node share the same kernel. Furthermore, an application may be distributed across multiple physical machines, and consequently across multiple kernels.

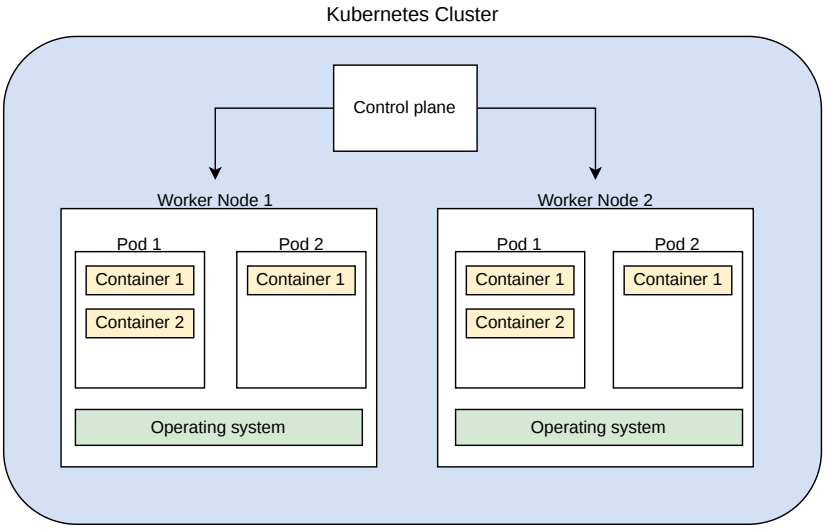


Figure 2.1: Very simplified view of the architecture of a Kubernetes cluster.

2.4 eBPF

eBPF (Extended Berkeley Packet Filter) is an event-driven programming technology that leverage the usage of hook points inside the Linux kernel [34]. It is a technology used to extend existing OS kernel capabilities in a safe and efficient manner. By attaching eBPF programs to kernel hook points, eBPF allows a user to load programs dynamically into specific places inside the kernel [58]. When the kernel or an application pass a specified hook point, the eBPF program is executed while respecting the core security and stability requirements of an OS [58]. Effectively, eBPF functions as a programmable Virtual Machine (VM) hosted by a kernel runtime, which means that eBPF programs are isolated from the kernel itself but can run in a privileged context [34]. This way, runtime security programs can be developed and enforce policies without interrupting the normal execution flow of the kernel, a crucial feature when dynamically analyzing program behavior [59].

2.4.1 Program Development

Developing eBPF functionality typically stems from crafting two distinct programs [58]. One program resides in the user space. This program can be written in multiple languages, such as Python, Go, and C++, acting as a frontend by interacting with the second eBPF program, situated inside the kernel space [60]. The second program is typically written in C and is used as a instrumental interface with the kernel

[60]. Usually, eBPF programs are written indirectly by using a userspace loader. The userspace loader provides a high-level abstraction interface to write the eBPF program [34]. In this thesis project, the userspace loader BCC is used for eBPF program development.

2.4.2 Program Execution

The eBPF program is loaded into the kernel as bytecode by using the system call BPF_PROG_LOAD, invoked by the userspace loader [34]. The bytecode is inspected by an eBPF verifier, ensuring the program is safe to run, that the running process has privileged capabilities, and that it does no harm to the system [34]. After compiling the eBPF program into bytecode, the Just-In-Time (JIT) compiler is invoked to translate generic bytecode into native machine instructions [34]. This optimizes the execution time and performance of the eBPF program [58]. Afterwards, the program is attached to the specified hooks.

2.5 Python

The following section introduce relevant Python terminology, such as what the difference between a module and a package is and how the Python import mechanism works. The section also contains a description of the Python/C API. For all technical details of Python discussed in this thesis, CPython is referred to as it is the original reference implementation of Python.

2.5.1 Modules and Packages

To differentiate between Python modules and packages, modules can be compared to the files of directories, where the directory itself is similar to a Python package [61]. Since Python only provides one distinct object type to represent a module, packages are used to represent module hierarchies. Technically, a Python package is just a module with a defined `__path__` attribute. Because of this attribute, a package is capable of knowing where associated submodules, subpackages and other items defined in the package are stored.

There are two main Python package categories. These are regular and namespace packages. Regular packages contain an `__init__.py` file that is not present in namespace packages. On the other hand, namespace packages serves mainly as a container for subpackages, and thereby lack a physical representation [62].

2.5.2 The Import System

When Python code inside a module is made available to another module, a Python import process has been completed [61]. There are multiple functions available to invoke this process, but the most common one is the `import` statement, which combines two processes into one [61]. The first process involves searching for the module

to import by calling the `__import__()` function. The second process binds the results of the search to a name, defined within the local scope of the process invoking the import. If a subpackage is requested, the parent package is imported before the subpackage [63]. The first time a module is imported, Python first checks if the module has been previously imported. If that is the case, an entry corresponding to the package will be stored in `sys.modules` object, serving as a cache [61]. If no such entry can be found, the search continues using finders and loaders [61].

Finding modules

The responsibility of the finders is to determine which loader should be used to execute the module. There are two different types of finders, meta path finders and path entry finders. In the import mechanism, meta path finders are the first finders to be called [64]. The meta path finders are stored in the `sys.meta_path` list, and Python iterates over them until the query is successful. These finders must implement the `find_spec()` method, which takes three arguments, a name, an import path, and an optional target module. By default, `sys.meta_path` contains three meta path finders. Together they're capable of finding built-in and frozen modules, as well as how to import modules from an import path.

Loading modules

By querying the finders present in `sys.meta_path`, Python checks whether it can handle the named module or not. If a meta path finder knows how to handle the module Python is searching for, it will return a `module spec` object [65]. The module spec object contains all information necessary for a loader to execute the module, including the appropriate loader itself. The module spec also updates the `sys.modules` cache. Once a module is executed by a loader, the module's namespace is populated [61]. If successful, a new module object is created by Python. The module object is then added to the `sys.modules` cache by the loader.

The module search paths used by Python during the import mechanism are stored in the `sys.path` list. The list is initialized during Python start and users can add their own paths using the `PYTHONPATH` environment variable. The list contains the directory of the project/input script as well as the prefix and exec_prefix directories which contain platform independent Python modules and extension modules respectively [66]. Extension modules are modules written in C or C++ using the Python/C API, described in subsection 2.5.3. The `sys.path` list also contains the path to the site-packages directory which contains third party modules [67].

2.5.3 Python/C API

CPython offers a API that exposes the Python interpreter to C and C++ programs, i.e the Python/C API. There are two main purposes of this API, to enable custom extension modules and to enable embedding Python in applications. The custom extensions are written in C and can interact with the Python Interpreter through the API functions. These functions are often called with arguments of type PyObject.

A PyObject is a pointer to the C representation of an arbitrary Python object. All PyObjects contain a type, e.g a list or an integer, as well as a reference count for memory allocation purposes [68].

2.6 Library Loading in C

The C programming language supports both static and dynamic linking of libraries. With static linking, the code and data of the library is added to the source program at compile time. With dynamic linking, multiple program can share the same libraries. These shared libraries are referred to as shared objects and are linked and loaded at runtime before execution of the first program statement by a dynamic linker-loader [69].

The C programming language also supports loading of additional libraries during runtime through a dynamic-linker API. One of the functions exposed by this API is dlopen(), which is used to load a new shared library [69]. If the shared object has dependencies, these are also loaded automatically [70]. Loading the shared object includes opening files and mapping them into memory [71].

3

Methodology

The proposed approach SnakeBPF is a runtime Python package detection method that combines tracing of operating system and Python events with data post-processing to identify packages and associated vulnerabilities. Note that the method is therefore not intended to use for discovering new vulnerabilities, but instead relies on prior knowledge of vulnerabilities linked to the detected packages. SnakeBPF consists of three main components illustrated in Figure 3.1.

Figure 3.1 shows how the different components interact and the information flow from operating system and Python event data collection to vulnerability identification. First, an information-gathering component collects data about currently running Python programs and their associated libraries. This is done either by observing interactions with the Linux Kernel or monitoring Python events.

Once raw data has been collected over a period of time and under varying traffic loads, the second component identifies which packages have been used during runtime. The final component matches the detected packages with known Common Vulnerabilities and Exposures (CVE) entries. This final component primarily enables comparison with existing vulnerability detection tools, whereas the main research focus of the thesis lies in the first two components.

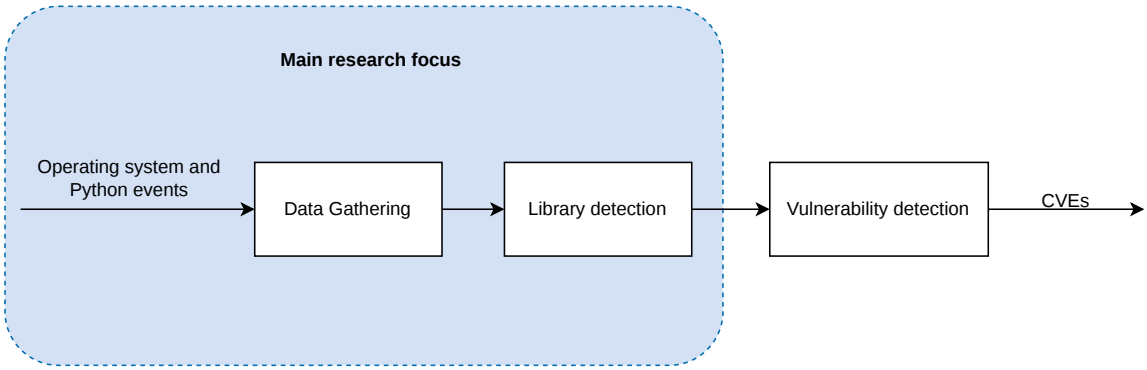


Figure 3.1: Runtime vulnerability detection method consisting of three components

3.1 Data Gathering

The data gathering component is responsible for tracing events of the monitored host. For Python package detection, several approaches can be used. In the following subsections, the different approaches and their use cases are discussed in detail. The first two subsections leverages eBPF for information gathering, which can be applied either to user space or kernel space. Another possibility to gather runtime package data is to use custom-made wrappers and override Python's import mechanism. Different approaches related to this technique will be presented in more depth in the third subsection. Finally, some information not directly related to Python packages must also be collected. This includes information such as existing processes on the machine, Python module search paths as well as package versions obtained from Metadata files.

3.1.1 User-space Tracing

The first type of data collection explored in this thesis is user-space tracing using eBPF. Two potential hook points relevant for Python package detection are investigated: the Python/C API and the standard C library function `dlopen`.

Python C API

By utilizing eBPF uprobes, a tracing function can be attached to the Python `PyImport_ImportModuleLevelObject` function. As described in Section 2.5.3, `PyImport_ImportModuleLevelObject` is part of the Python/C API and is invoked when a module is imported. The function accepts multiple parameters, where the first parameter is a `PyObject` representing the name of the module to be imported.

As described in Section 2.5.3, a `PyObject` is the C representation of a Python object. Since Python objects are not directly accessible from eBPF programs written in C, the `PyObject` is represented by a memory address. To access the desired name field of the `PyObject`, the correct memory offset must be known. A primary limitation of this approach is that these offsets are unstable across Python versions. Furthermore, the availability of hook points in the Python/C API, as well as the internal usage patterns of these functions, are version dependent.

When performing package detection on containerized applications, the uprobe must be attached to the container's `libpython` binary rather than the host machine's `libpython`. The path to the relevant `libpython` binary can be identified through the `/proc/{pid}/maps` file of the target process. To reduce implementation complexity, uprobes are attached to all matching `libpython` binaries using the regular expression `.libpython..so.*`. Since new containers may start during runtime, the search for matching `libpython` paths must be performed periodically.

dlopen

eBPF uprobes are also used to trace the `dlopen` function, described in Section 2.6. The uprobe is attached to the C standard library, and a trace event is generated each time the function is used to load a shared library. Similar to the `PyImport_ImportModuleLevelObject` uprobe described in Section 3.1.1, tracing containerized applications requires resolving the correct library path within the container environment rather than using the host system's library path.

3.1.2 Kernel-space Tracing

eBPF can also be used for kernel-space tracing. This is performed for two reasons, to retrieve package usage information and information about running processes.

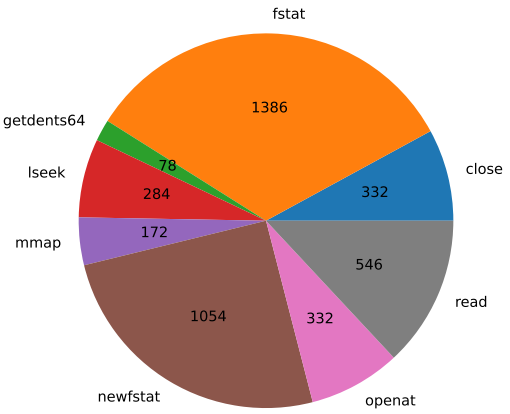
Information about Packages

To determine which system calls are most relevant for Python package detection, the tool strace[1] was used. Additionally, a Python test program containing the single line `import pandas` was constructed.

The results of running strace on this program are shown in Figure 3.2a and Figure 3.2b. Figure 3.2b contains the subset of system calls in Figure 3.2a whose arguments reference the imported library `pandas`. This figure therefore provides an indication of which system calls are most relevant for Python package detection.

% time	seconds	usecs/call	calls	errors	syscall
32.91	0.008834	6	1263		read
17.48	0.004693	2	2197	260	newfstatat
14.48	0.003886	5	777	14	openat
9.01	0.002419	4	549		mmap
4.94	0.001326	1	765		close
4.55	0.001222	11	111		brk
3.78	0.001015	1	771		fstat
3.03	0.000813	1	656	3	lseek
2.96	0.000795	5	146		getdents64
2.21	0.000592	13	43		munmap
1.91	0.000513	4	108		mprotect
1.55	0.000417	417	1		execve
0.43	0.000116	3	35		futex
0.15	0.000040	40	1		open
0.09	0.000024	2	11		clone3
0.07	0.000019	0	27		rt_sigprocmask
0.06	0.000016	4	4		pread64
0.06	0.000016	5	3		getrandom
0.04	0.000012	1	12		mbind
0.04	0.000010	10	1	1	access
0.04	0.000010	5	2		prlimit64
0.03	0.000008	0	11		madvise
0.03	0.000008	8	1	1	statfs
0.03	0.000007	3	2		getcwd
0.03	0.000007	3	2		sched_getaffinity
0.02	0.000005	2	2		gettid
0.02	0.000005	5	1		rseq
0.01	0.000004	4	1		arch_prctl
0.01	0.000004	4	1		set_tid_address
0.01	0.000004	4	1		set_robust_list
0.00	0.000001	0	67		rt_sigaction
0.00	0.000001	1	1		epoll_createl
0.00	0.000000	0	8	1	ioctl
0.00	0.000000	0	3		uname
0.00	0.000000	0	6		fcntl
0.00	0.000000	0	9	4	readlink
100.00	0.026842	3	7599	284	total

(a) Running the Python test program with strace -c.



(b) Summary of system calls detected with strace -yy whose arguments contains the word pandas.

Figure 3.2: System calls detected by strace on a Python program containing the line " import pandas ".

Based on the above results, the most frequently called system calls are `fstat`,

[1] https://strace.io/

`newfstatat`, `openat`, `close`, `read`, and `mmap`. As described in Section 2.2.2, `newfstatat` and `fstat` are used to obtain information about file descriptors associated with open files. Therefore, these system calls were deprioritized, since file descriptors must first be generated through another system call such as `openat`. The `close` system call was also deprioritized due to its close relationship with the `openat` system call. In summary, the system calls `openat`, `read`, and `mmap` were selected as potential candidates for providing useful insights regarding Python package usage.

Tracing the `openat` system call returns information about the opened file. The data relevant for Python package detection in this thesis consists of the file path of the opened file, as well as the PID of the process requesting the file. The file path is later used to determine the accessed package, while the PID is used to identify which application accessed the package. In this implementation, the collected results are stored in a CSV file before being processed by the second component.

For the `openat` system call, the file path is provided directly as one of the function arguments and is therefore directly accessible. In contrast, the `read` and `mmap` system calls operate on file descriptors instead of file paths. Since the file path is required to determine package usage, the file descriptor must first be translated into its corresponding file path.

A successful `openat` call returns a file descriptor associated with the opened file. Therefore, the `openat` eBPF program stores the filename together with the generated file descriptor in an eBPF map that can be accessed by other eBPF programs. When tracing the `read` and `mmap` system calls, the corresponding file paths can then be retrieved from the eBPF map using the file descriptors as lookup keys. Figure 3.3 illustrates this process for the `read` system call.

Information about Processes

In addition to tracing the `read`, `openat`, and `mmap` system calls described previously, tracing of the `execve`, `clone`, `fork`, and `vfork` system calls is also performed. The primary purpose of tracing these system calls is to collect process-related data. This data consists of process IDs and parent process IDs (PIDs and PPIDs), which are used to determine which application accessed a particular package (see Section 3.2.2). Tracing of the `execve` system call was implemented using a simplified version of the existing BCC tool execsnoop[2].

3.1.3 Overriding the Python Import Mechanism

One alternative approach to data gathering using eBPF is to to override the Python import mechanism. All imports pass through the import process, either partly or in full, depending on if the library is cached or not. Collecting information by tracing the import mechanism, that can later be processed to detect library usage, is done through wrappers or adding additional finders. These are activated by importing

[2]https://github.com/iovisor/bcc/blob/master/tools/execsnoop.py

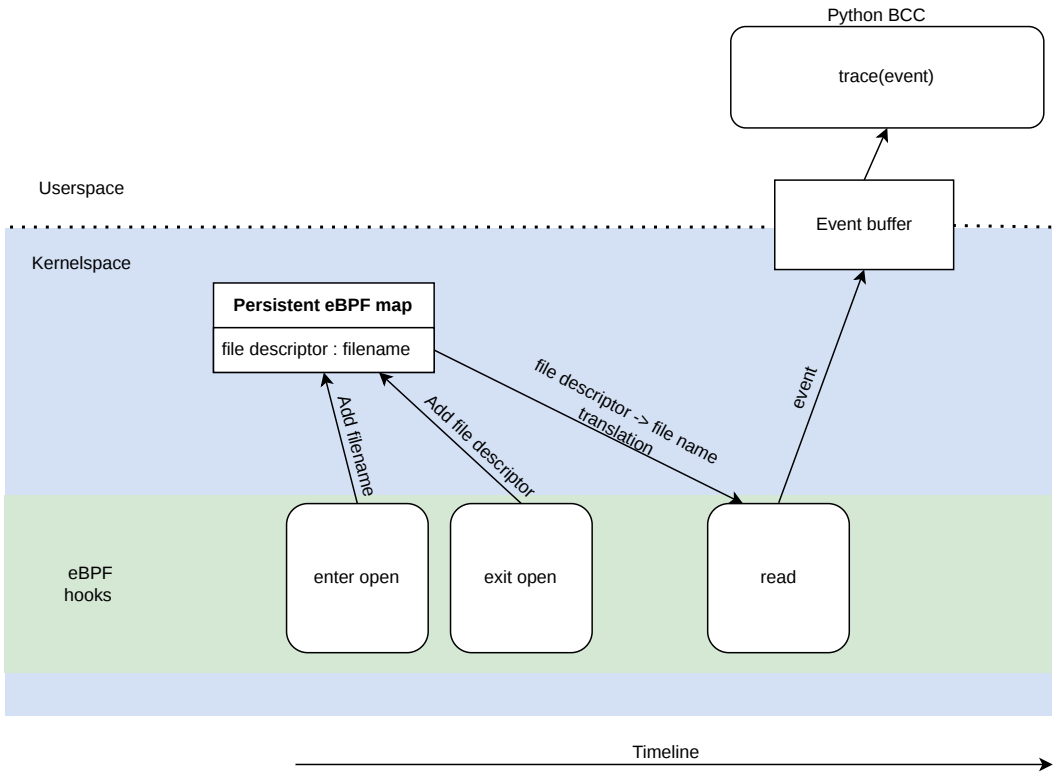


Figure 3.3: Tracing the `read` system call using eBPF.

them at the top of the import statements of the main program to be monitored. Therefore the source code of the traced application must be modified to support this data collection method, which can be difficult in production environments.

Wrapping the Import Statement

As described in Section 2.5.2, the import mechanism is most commonly initialized with the `__import__` statement. A wrapper for this statement can be created by replacing the `builtins.__import__` method with another function. The wrapper does some custom functionality, in this case tracing, and then calls the original `__import__` statement with the original parameters. The custom tracing function will therefore be called every time the import statement is invoked. The following code snippet depicts an overview of how the import statement is wrapped.

```
1  import builtins
2
3  _real_import = builtins.__import__
4
5  def custom_wrapper(name, globals=None, locals=None, fromlist=(),
       level=0):
6
7      do_custom_tracing()
8
9      return _real_import(name, globals, locals, fromlist, level)
10
```

```
builtins.__import__ = custom_wrapper
```

Listing 3.1: Wrapping of the Python import statement

The `do_custom_tracing` function writes the name of the imported module to a log file, which is processed later in the pipeline by the second component.

Wrapping the find_load() Method

Another approach is to wrap the `find_and_load()` function called when a module is located and loaded in the import process. Similarly to Listing 3.1, the following listing depicts how such an override can be implemented.

```
import importlib._bootstrap

_real_find_and_load = importlib._bootstrap._find_and_load

def custom_wrapper(name, import_):

    do_custom_tracing()

    return _real_find_and_load(name, import_)

importlib._bootstrap._find_and_load = custom_wrapper
```

Listing 3.2: Wrapping of the Python import `find_and_load()`function

Additional Custom Meta_path Finder

A third approach is to add a custom finder to the meta path finder list. As described in Section 2.5.2, the import mechanism will iterate over the meta path finder list until a finder returns a module spec. The list is iterated over in order, and therefore a custom finder can be placed first in the list as long as it does not return a module spec. In this scenario, once the code of the custom finder has been executed, the import mechanism will continue to the next meta path finder as usual. The functionality of the finders is therefore not disturbed.

The following listing shows how a custom meta path finder can be implemented. The custom finder must inherit the abstract `MetaPathFinder` class and implement the `find_spec()` method. Additionally, for the iteration to continue over the default path finders the custom finder must return None. Finally, the custom tracer is inserted at the beginning of the meta path finder list. This is depicted in the following listing.

```
class CustomTracer(importlib.abc.MetaPathFinder):
    def find_spec(self, fullname, path, target=None):

        do_custom_tracing()

        return None

```

```
8 sys.meta_path.insert(0, CustomTracer())
```

Listing 3.3: Implementation of a custom meta path finder for tracing

3.1.4 Container-based Tracing

SnakeBPF leverages static information to obtain package versions and Python `sys.path` variables, necessary to perform full library detection. This static information must be obtained from within a container, as package versions and file system hierarchies are application specific.

Package Versions

Gathering Python packages versions can be done in multiple ways. One method is to iterate over a container's internal Python site-packages directory and accesses metadata files to extract version numbers. For directories without explicit metadata files, the keyword `__version__` can be used.

However, as the package manage Pip is used by all open-source applications in this thesis, the command presented in Listing 3.4 will be used to gather package versions. Since Pip is capable of listing all packages and their corresponding versions installed within a Python environment, this command is executed within the inspected container. The output from each container is then stored in a CSV file, used for later package version matching.

```
1 pip list | tr -s '[:space:]'| sed 's/ /,/g' | sed '2d'
```

Listing 3.4: List Python packages and their corresponding versions installed by Pip

Python Module Search Paths

Section 3.2.1 discusses how Python module paths can be used to determine the relevance of detected Python packages. As described in Section 2.5.2, the Python `sys.path` variable contains a list of paths were Python modules may be stored. The value of the `sys.path` variable can be printed directly to a test file for later processing.

3.1.5 System Snapshot

All of the data collection approaches described above are runtime-based. This means that they only capture events occurring after the monitoring has started. This introduces a limitation, since data collection must begin at system startup to ensure complete coverage of all events. To address this limitation, a system snapshot is introduced to capture the existing state of the machine before tracing is initialized. This supports data collection from containers and processes that were already running before launching SnakeBPF.

To obtain a system snapshot, the first step is to identify all running containers on the system. This is achieved by leveraging either Docker or `crictl`, depending on the runtime environment, to retrieve a list of active containers. For each container, the container ID (CID) and the associated process ID (PID) are extracted. For each PID, standard Linux utilities such as `ps` are used to recursively enumerate all child processes and their descendants. Listing 3.5 shows an example of a system snapshot, where each CID is associated with a root PID and its child processes. The root PID refers to the main process of the container, similar to the init process in traditional Linux systems.

```
{
  "06414eaf48a5bd3e39d2671c4ad1c977fb67a627296f5524e98396a5dd5b94ef
    ": {
    "root_pid": 11506,
    "children": [11591, 11592]
  }
}
```

Listing 3.5: Example of captured system snapshot where one container is running. Two processes with PIDs 11591 and 11592 are running inside the container.

3.2 Data Processing

After data gathering has been performed by the first component, the second component performs data processing to identify detected libraries. This processing component applies regular expressions and filtering logic to traces obtained from the data gathering component, and results in a CSV file containing packages and package versions used during the tracing period.

3.2.1 Package Identification

For data collection with eBPF kernel space tracing, the collected data generated by the first component consists of PIDs and a file path. To identify packages, the processing component iterates over all data rows and evaluates whether the file path contains one of the specified module search paths for Python, e.g. the `site-packages` directory. If the path contains one of the module search paths, the package name is extracted by taking the first directory of the file path directly following the detected module search path.

Listing 3.6 is an example of a file open trace. The process `python3` with PID 218680 has requested to open a compiled Python file. For this specific entry, the processing component will detect that the file path does in fact contain a module search path, in this case `/.venv/lib/python3.14/site-packages/`. The processing component then extracts the package name `pandas`, as this is the first directory specified after the module search path. As described in Section 2.5.1, Python considers both packages and subpackages. However, this script will only consider packages, as software vulnerabilities and their associated CVEs are typically reported per package basis.

```
PID,process,filename
218680,python3,/.venv/lib/python3.14/site-packages/pandas/
    __pycache__/__init__.cpython-314.pyc
```

Listing 3.6: Example trace of the openat syscall

After the package name has been determined, the processing script will determine the library version by evaluating if any similar library exist in the library versions data, retrieved through container-based tracing as described in Section 3.1.4. If no exact package match exist, the processing component will make a guess by looking for the longest common substring (LCS) for each package without a specific version. This is done by iterating over all known packages with a version, and for each package without a version, calculate the LCS. If the final LCS has a length over 4, the version of the package which yielded the longest LCS is applied as a guess. In the case where a package name without a version has a length of 3, the LCS must instead be of length 3 to be a valid guess.

Listing 3.7 depicts a line of the resulting CSV file when the processing component has made a guess. The line contains the detected library, the version as well as the assumed version in parenthesis. The approximation is made because the file path and the official package name are not always consistent. In the following case, the package `wagtail_font_awesome_svg` is sometimes accessed by the name `wagtailfontawesomesvg`.

```
package,version
wagtailfontawesomesvg,1.2.1(wagtail_font_awesome_svg?)
```

Listing 3.7: Example of a Python package with a guessed version number

For data collected with eBPF user-space tracing or Python import mechanism wrapping, less processing is needed as the data consists of package names and not file paths. However, the same version mapping is performed regardless of scenario.

3.2.2 Handling Multiple Containerized Applications

Each detected package is associated with the PID of the process using the Python package. However, multiple processes on the system may use multiple different Python packages during the same tracing window. To determine which application has used a particular package, all PIDs must be grouped according to which application they originate from. As described in Section 3.1.2, data collected from the `execve`, `clone`, `fork`, and `vfork` system calls can be used for this purposes. Using traces from these system calls together with the system snapshot described in Section 3.1.5, process relationships can be reconstructed in order to associate processes with their corresponding application.

Identify Target Processes

The first step is to identify PID targets, which are processes we know belong to a targeted application, that might have invoked the `openat` system call to open a Python

package. Typically, these processes are the parent processes of containerized applications. For this purpose, the system snapshot and trace of system call `execve` is used. The system snapshot utilizes either the `docker inspect` or `crictl inspect` command to list all known containers and their children processes. This way, containers and any associated processes invoked before tracing system calls will be known.

From Section 2.3.1, we know that container initialization is handled by a `containerd-shim` process. To identify target PIDs from the `execve` trace, we utilize this knowledge by looking for new namespace creations invoked by `containerd-shim` processes. Since a new namespace is created specifically for a distinct container, the container ID (CID) of the future container is passed as an argument whenever a `containerd-shim` process is invoked. For each `containerd-shim` entry, this CID is retrieved using regex along with the PID from the `containerd-shim` entry.

In settings where Kubernetes is utilized for container orchestration and deployment, identifying target PIDs is extended to handle pod rescheduling as well (see Section 2.3.3). During testing, this event occurs whenever a running pod is killed deliberately and Kubernetes automatically creates a new pod instance. When this happens, the `containerd-shim` asks a `runc` process to stop the pod and invoke a new running instance of the pod (Section 2.3.1). In this case, the CID of the new container can be retrieved by pattern-matching the arguments passed to the `runc` process when executing the `start` container command, which is present in the `execve` trace. However, the root PID of the container is still set to the PID of the `containerd-shim` process responsible for rescheduling the pod.

Listing 3.8 shows an example entry from the `execve` trace, picturing how a Docker container namespace is initialized by a `containerd-shim` process with PID 17753. The container ID starting with **06414ea...** and PID 17753 on line 2 is retrieved and saved as a targeted process for the remaining steps of post-processing.

```
1 TIME(s),PCOMM,PID,PPID,RET,ARGS
2 164.712160,containerd-shim,17753,1006,0,"/usr/bin/containerd-shim-
    runc-v2" "-namespace" "moby" "-address" "/run/containerd/
    containerd.sock" "-publish-binary" "/usr/bin/containerd" "-id" "
    06414eaf48a5bd3e39d2671c4ad1c977fb67a627296f5524e98396a5dd5b94ef
    " "-bundle" "/run/containerd/io.containerd.runtime.v2.task/moby
    /06414
    eaf48a5bd3e39d2671c4ad1c977fb67a627296f5524e98396a5dd5b94ef" "
    delete"
```

Listing 3.8: Execve trace of a containerd-shim process

Process Grouping

After identifying target PIDs, the next step is to find their children processes. The snapshot (see Section 3.1.5) provide this information for containerized applications invoked before tracing, by iteratively searching for processes created by a container. For applications deployed during tracing, the traces obtained from both `execve` and `fork` are inspected. Since both traces keep track of parent process IDs, known as

PPIDs, a breadth-first search can be performed whenever a target PID is listed as a process' parent. After successful completion, this stage results in a best-effort process grouping for each containerized application. When combined with the package detection methodology described in Section 3.2.1, the final output maps which container ID has utilized a specific Python package and its version. These results are later used to retrieve CVEs associated with each detected package.

3.3 Vulnerability Detection

It is possible to obtain a list of CVEs associated with a specific library and version by querying the distributed Open Source Vulnerability (OSV) database [72]. To query OSV, a library together with a specific version along with which ecosystem it resides in (e.g. PyPI for Python) is provided [72]. The API response contain a summary of reported vulnerabilities associated with the library and version, along with their specific CVEs, and affected library versions [72]. By utilizing the API, the intermediate step of translating libraries and their versions to CPE numbers is not needed. Further implementation specific details are specified in the API documentation [3].

[3] https://google.github.io/osv.dev/api/

4

Evaluation

This chapter introduce the methodology used to evaluate the proposed detection approach SnakeBPF. To evaluate SnakeBPF, testing programs used to perform package detection are needed. In this chapter these programs are introduced, as well as how they will be interacted with to trigger package use. In addition to these programs, a baseline must be established. This baseline represents the correct results, which is needed to determine how well SnakeBPF perform. Towards the end of the chapter, the test scenarios are presented as well as a short discussion related to which evaluation metrics are used.

4.1 Experimental Configuration

Evaluation of SnakeBPF is performed in two different settings. One relies on using open-source web applications. The results from these tests will be fully disclosed for the public. The second setting includes testing SnakeBPF in a a 5G packet core Kubernetes cluster running on Ericsson test infrastructure. Because of the corporate setting, the testing environment will only be described in short. Further, the findings of the package detection approach applied to the Kubernetes cluster will be masked. Only the number of detected packages will be disclosed.

4.1.1 Open-source Web Applications

Five popular Python open-source web applications with docker setup have been used as test programs for evaluating the package detection approaches. In this subsection, each of these applications are briefly introduced.

Wagtail's Bakery Demo

The Wagtail Bakery Demo[1] is a web application used to demonstrate the Wagtail content management system and its capabilities. The application handles API calls from a website user and provides an admin interface among other features. In this thesis, the web application has been deployed using Docker. Figure 4.1 showcase the user interface of the Bakery Demo home page.

[1] https://github.com/wagtail/bakerydemo

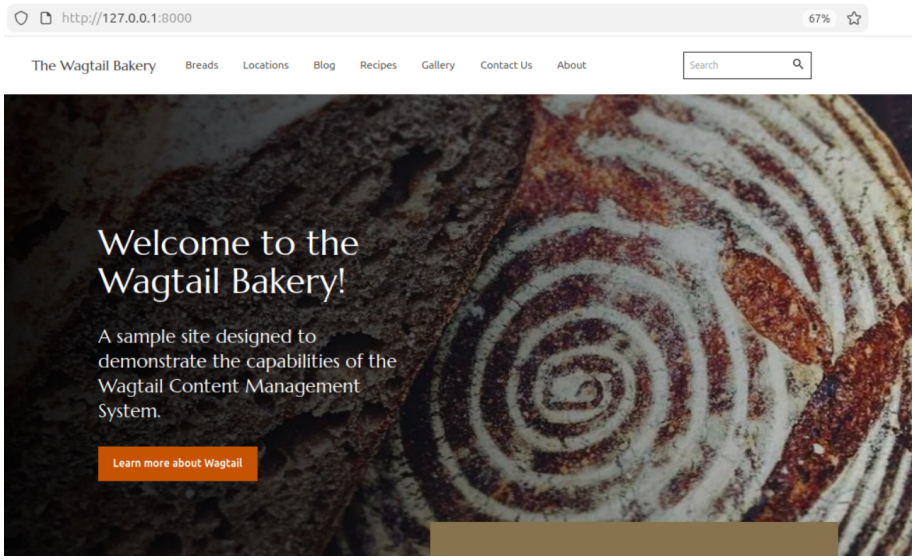


Figure 4.1: The Wagtail Bakery Demo home page

Docker-Django

Docker-Django[2] is a simple web application deployed as a Docker container. Its capabilities are limited and mainly provide URLs that redirect users away from the website. Figure 4.2 showcase the Docker-Django web interface.

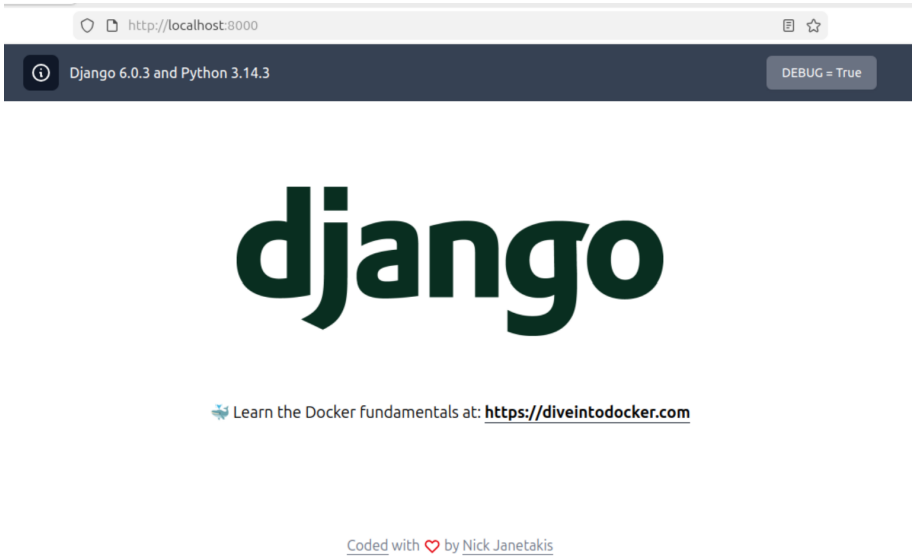


Figure 4.2: The Docker-Django web interface

Moments

Moments[3] is a social networking web application built using the Flask framework. It supports creation of user accounts and image uploading. Figure 4.3 shows the login home page.

[2]https://github.com/nickjj/docker-django-example
[3]https://github.com/greyli/moments

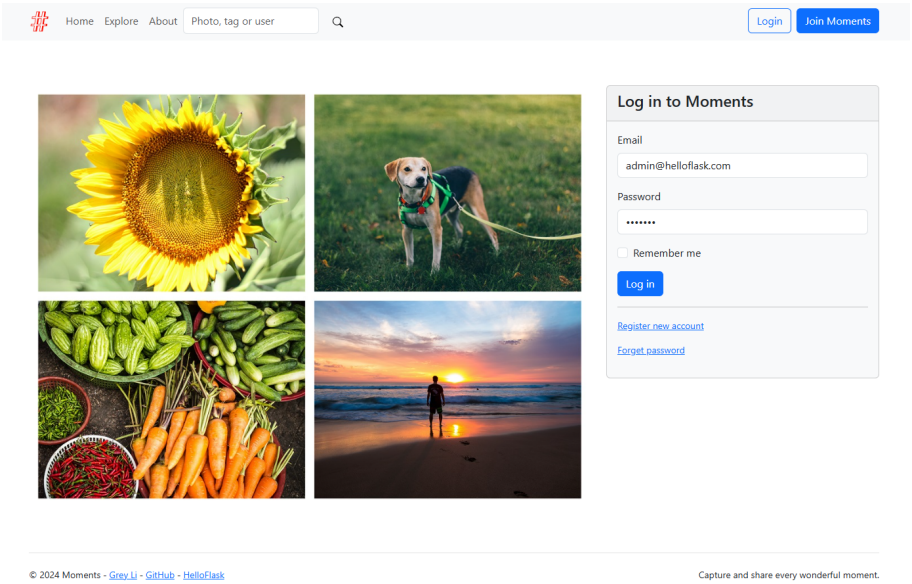


Figure 4.3: The home page of Moments

Todoism

Todoism[4] is a web application built using Flask. After creating an account, a user can create tasks to be accomplished. These tasks can then be marked as completed by checking the tick box displayed next to the task description. In total there are three separate lists displaying ”All”, ”Active” and completed ”Done” tasks. In addition, the app provides a ”Clear” functionality, removing all completed tasks. Figure 4.4 presents the web app home page.

Plone

Plone[5] is a content management system built with Python and React [73]. Upon deployment, the administrator can create a new Plone web site and add content such as events, images and text posts. Figure 4.5 showcase a newly created Plone site using the default arguments.

[4]https://github.com/greyli/todoism
[5]https://github.com/plone/plone.docker

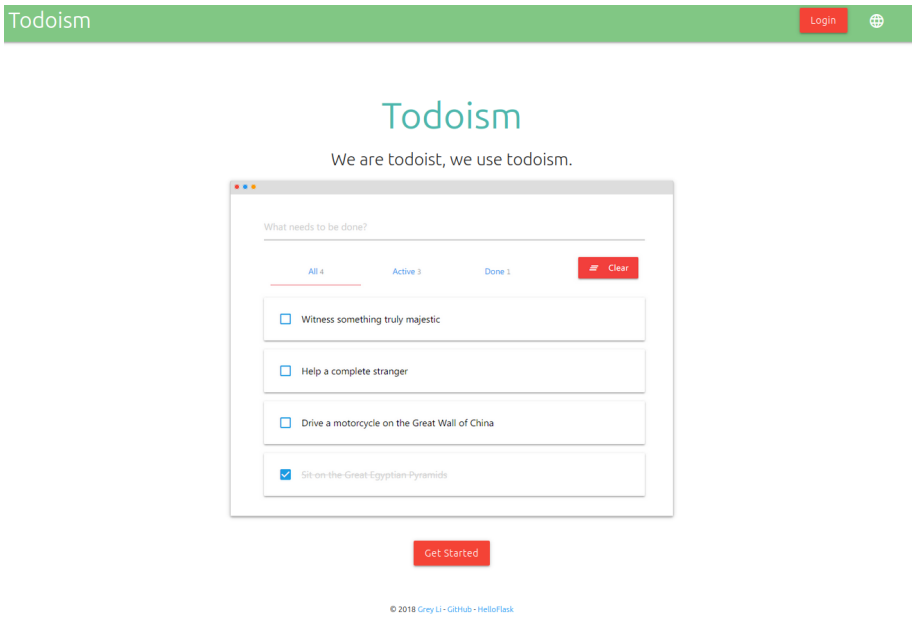


Figure 4.4: The home page of Todoism

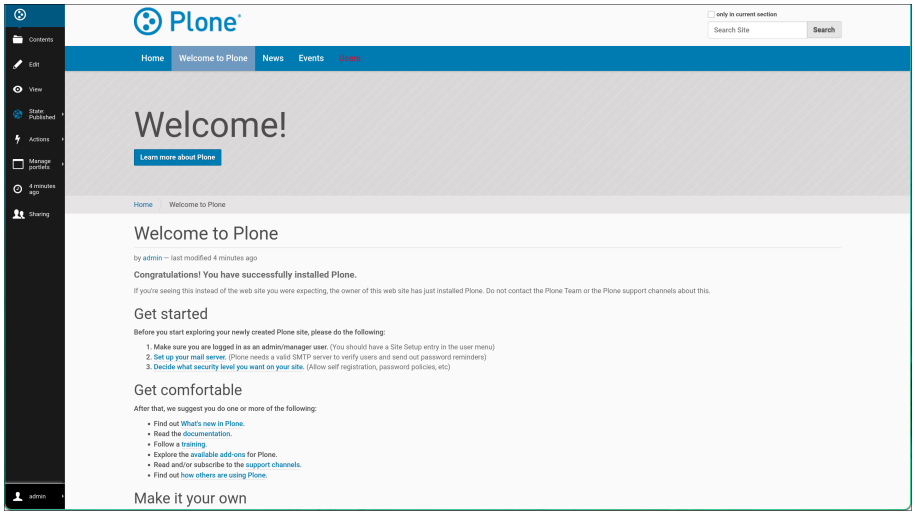


Figure 4.5: The home page of Plone

4.1.2 5G Core Kubernetes Cluster

The purpose of testing SnakeBPF inside a live Kubernetes cluster at Ericsson, is related to verifying that SnakeBPF can be applied in a real-life scenario motivated by the problem statement presented in section 1.2. If the solution manages to handle multiple containers in a truly distributed environment, its an indication that the methodology scales well in an increasingly complex setting.

In this thesis project, a 5G Packet Core Kubernetes cluster will be used for testing purposes. The cluster consist of one control plane and four workers. Each worker runs on a virtual machine and is responsible for managing multiple pods running simultaneously. For this thesis project, normal 5G network traffic is simulated by

default on the cluster during testing.

4.2 Application Loads

To trigger Python package usage among the testing environments described previously, they must be interacted with. This section describes the interactions performed during the evaluation. Since the microservices of the 5G core Kubernetes cluster cannot be scanned using a web scanner or interacted with through a browser, dynamic application behavior will be triggered differently and is described in a separate subsection.

4.2.1 Web Scanning using ZAP

To trigger as many functionalities in the web applications as possible, the web application scanner ZAP is used (see subsection 2.1.2). The expectation of triggering all functionalities in the application, is that all Python packages used somewhere in the code will be used. The results of this load should then in theory be useful to determine if the detection approach has systematic blind spots, i.e. if it is not able to detect certain packages in all scenarios.

The results of the web scanning load can also be viewed as the number of packages used by a container under maximum load. This in turn provides an upper bound threshold for other evaluation loads, as the targeted evaluation is expected to trigger less packages than ZAP. When using ZAP to subject an application to a significant load, the traces obtained are referred to as `load_trace`, e.g. when comparing different tracing strategies (see subsection 4.4.3).

4.2.2 Targeted Functionality Testing

When targeting a specific functionality of a web application to trace it's dynamic behavior, the goal is to trace a subset of the packages used by the web application. Because of this, each test must be constructed with respect to what functionalities each web application provide. This section presents how targeted functionality tests have been carried out for each open-source application.

Wagtail Bakery Demo

The Wagtail web application is deployed in a container, according to the instructions provided in its open-source repository[6]. The data gathering component is started and a browser is used to see the web interface. The header section "contact us" is opened and the web page form is completed with default values. A confirmation message is received as response if the query was successful. After the response is received, the data gathering component is terminated, and the results are processed by the processing component.

[6] `https://github.com/wagtail/bakerydemo?tab=readme-ov-file#setup-with-docker`

Docker-Django

The web application is deployed in a Docker container. Then, the data gathering component is initialized. The three hyperlinks presented as ”http://diveintodocker.com”, ”coded”, and ”Nick Janetakis” are clicked. Afterwards, the data gathering component is terminated and the data is processed.

Moments

The application is deployed in a Docker container. After deployment, the data gathering component is started. Then, the website is signed in to and a image of choice is published. After the image is successfully published, terminate the data gathering component.

Todoism

Docker is used to deploy the web application. After deployment, the data gathering component is started. The ”Login” button is clicked and ”Get a test account” is selected. The generated test account credentials are used to log in. After successful log in, all visible default tasks are marked as completed. Next, the ”clear” button is clicked to remove all tasks. Afterwards, the data gathering component is terminated.

Plone

The evaluation procedure consisted of deploying the Plone application, logging in through the web interface using the default credentials, creating a new Plone site with the default template, and finally creating a new event.

4.2.3 5G Core Kubernetes Cluster Interaction

As mentioned in subsection 4.1.2, normal network traffic is configured to run on the Kubernetes cluster by default. Therefore Python package usage when ordinary traffic is running can be captured by tracing on all workers without targeting a specific pod or microservice. These traces are used as a reference during later evaluations, to determine which detected packages are a result of dynamic behavior triggered by custom interaction.

These custom interactions consists of two parts, rescheduling a pod and later sending curl requests to the targeted pod. A pod is selected as target based on the size of its Python codebase and how many third-party Python packages are utilized internally. The selected pod used for these experiments run two containers. One container is only used during pod initialization, while the other remain active throughout a pod’s life cycle. After initiating tracing on all workers, the targeted pod is killed deliberately to trigger automatic pod rescheduling. This scenario is expected to be handled accordingly to the post-processing methodology described in subsection 3.2.2.

Since the targeted pod is replicated, a second instance of the pod is invoked during

rescheduling. Once rescheduling is completed, sending curl requests to the new pod is predicted to trigger Python package usage for the first time. However, since tracing is initialized before the pod is rescheduled and redeployed, this means that we might also capture packages used only during container initialization, and not as a direct results of handling the curl requests. The number of traced packages will then correspond to the union of `run trace` and `load trace`, presented later in subsection 4.4.3.

4.3 Baseline Candidates

To evaluate SnakeBPF, a baseline must be established to determine the expected packages to detect. However, since the results of dynamic analysis are highly dependent on user behavior, defining an accurate baseline is inherently non-trivial. Consequently, multiple alternative approaches are investigated in this section.

4.3.1 User-space Probing

As described in section 2.4, eBPF enables userspace tracing through uprobes. In subsection 3.1.1, userspace probes were attached to the Python/C API function `PyImport_ImportModuleLevelObject`. However, implementing this approach and attaching custom tracing functions is complex and highly dependent on specific Python versions. Due to these version-specific constraints and implementation challenges, the approach was deemed unsuitable for production use.

Despite these limitations, the method provides a dynamic tracing mechanism in which each import statement should, in principle, trigger the probe. Therefore, it is considered a viable baseline for evaluation purposes. The approach is evaluated using the Wagtail open-source application, where all libraries are expected to be triggered through interaction with the ZAP web scanning tool (see subsection 2.1.2). The resulting set of detected imports is then compared against the baselines defined in the following subsections.

4.3.2 Overriding the Python Import Mechanism

Tracing libraries through the Python import mechanism is introduced in subsection 3.1.3. While this method could theoretically be used for data collection in the final approach, it requires modifications to the CPython implementation, which may introduce stability and maintainability issues in production environments. Additionally, the source code of the analyzed application must be modified, and the approach is not generalizable to other programming languages. For these reasons, it was deemed unsuitable as part of the final system.

Nevertheless, due to its ability to capture import events dynamically, this method is retained as an alternative baseline for evaluation. The data collection process mirrors that of the uprobe-based approach, and the resulting detected libraries are used for comparative analysis.

4.3.3 Static Analysis

The final approach considered for establishing a baseline are the static analysis tools Trivy and Syft, described in subsection 2.1.1. Trivy will be used for evaluating open-source applications, while Syft is only used for test applications at Ericsson. Both tools produce analysis results that contain a complete list of packages and their corresponding versions, which are subsequently transformed into the same format as the outputs produced by the dynamic methods described in this report. The analysis compared with other baseline candidates is performed on the Docker image of the monitored applications.

Using static analysis as a baseline for evaluating dynamic approaches presents certain challenges. In particular, dynamic analysis is expected to identify only the subset of dependencies that are actually exercised during execution, whereas static analysis captures all declared dependencies. As a result, discrepancies between the two approaches must be interpreted with care.

4.4 Evaluation Scenarios

This section will describe the experimental evaluation scenarios briefly.

4.4.1 Baseline Selection

To evaluate which of the baseline candidates that provide the most accurate dynamic package detection baseline, experiments on the Wagtail Bakery Demo are performed. Desired traits of the baseline candidates is that when the application is subjected to ZAP web scanning and all packages are expected to be in use, the baseline should also find all packages detected by the static candidate Trivy. Trivy is chosen as the reference tool because it is widely used and curated to display only relevant packages. If the candidates pass this test, the candidates will then be evaluated when running applications under a targeted load. In this case, less packages than Trivy should be detected by an accurate dynamic baseline candidate.

4.4.2 Data Gathering Approach Selection

To compare the different data collection approaches described in section 3.1, they are evaluated using all open-source web applications. The applications are subjected to both web scanning with ZAP, as well as targeted loads.

4.4.3 Tracing Strategies

Since container deployment affects the output from system call tracing, careful consideration needs to be taken in terms of when we should start tracing and what behavior we would like to capture. Docker provides different techniques one can use in order to build and run a container. The `docker build` command creates a

static image, while the `docker run` command is responsible for creating and running a container based on said image. For this reason, we would like to trace each operation separately, represented as `build trace` and `run trace` in Figure 4.6.

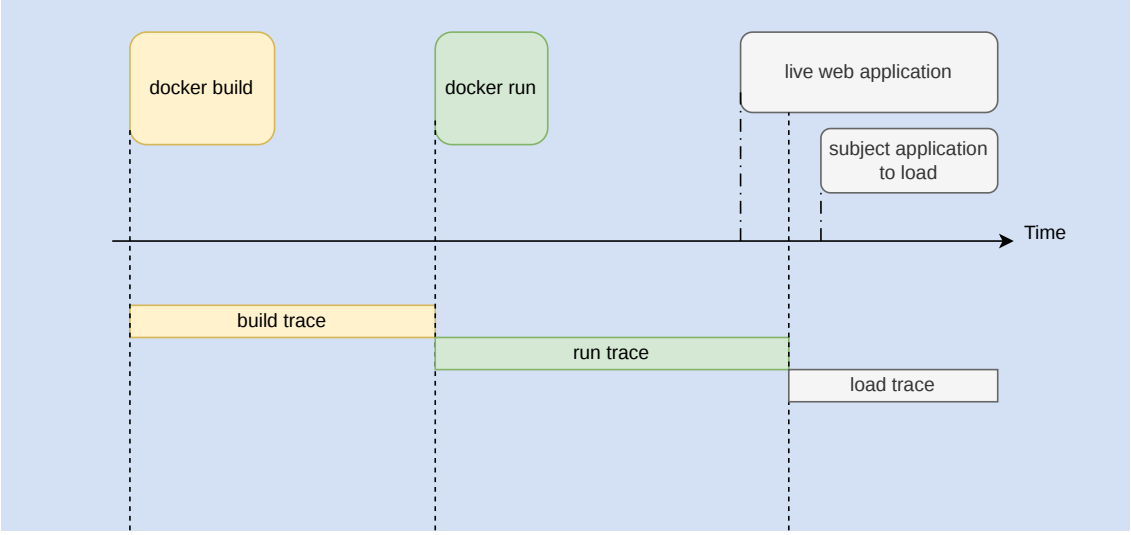


Figure 4.6: System call tracing during different stages of a containerized application's life cycle.

This way, we'll be able to determine which packages are used in build context only, and decide whether these should be excluded when reporting packages used during runtime or not. Such exclusion is motivated by the fact that vulnerabilities linked to build dependencies may be less relevant when prioritizing vulnerabilities actively exposed in a running application. The third tracing stage `load trace` in Figure 4.6 is performed after a containerized web application is live and can be accessed through a web browser's interface. The purpose of this trace is to capture dynamic application behavior when subjected to a load, such as a user's actions on a website.

4.4.4 Proposed Approach

Once the most suitable data collection approach has been determined, SnakeBPF will be evaluated when using this method only for data collection. SnakeBPF will again be applied to all web applications that are subjected to targeted loads and web scanning loads. The results from the proposed package detection approach is then translated to potential CVEs. These are compared with CVEs from the determined baseline. SnakeBPF will also be evaluated inside a Kubernetes 5G packet core environment, detecting package usage for a specific worker.

4.4.5 Multi-container Environment

To determine if SnakeBPF can be applied to containerized environments, two web applications (Wagtail's Bakery Demo and Moments) are deployed simultaneously on a local machine. SnakeBPF is applied to the machine, and ZAP scanning is then performed on both applications. The main goal of the application is to evaluate

whether the detection approach manages to map the packages to the correct container and application, a crucial feature when later evaluating SnakeBPF inside a Kubernetes cluster.

4.4.6 Python's In-memory Cache Impact

To determine the impact of pythons in memory cache, the detection technique is also applied on the Wagtail Bakery Demo during a prolonged time. The detection approach is applied before the container image is built, and is not terminated until 9 hours have passed. During this time, the application is subjected to load from ZAP twice, once directly after application deployment and once after 9 hours. After the ZAP scan has finished, SnakeBPF tracing is terminated and results are processed.

4.5 Evaluation Metrics

In binary classification, an event can be assigned one of two possible labels. Python package detection is an example of such a classification task, as a Python package can either be used or not used. Considering the results produced by a binary classification scheme, there are typically four different outcomes to consider when evaluating the results. In the context of Python package detection, the possible outcomes are defined in Table 4.1.

Table 4.1: Confusion matrix for Python package detection, showcasing possible classification outcomes.

		Detection	
		Used	**Unused**
Actual	**Used**	Correctly detecting used package (TP)	Not detecting used package (FN)
	Unused	Wrongly detecting unused package (FP)	Correctly ignoring unused package (TN)

According to the table above, each package can be mapped to one of four categories. However, the categories can be combined and weighted differently to generate new metrics. One possible metric is recall. A perfect recall is achieved when the package detection approach detects all Python packages in the baseline. However, a package detection approach that simply reports a very large number of packages could still achieve perfect recall despite producing many false positives. Therefore, false positives and false negatives must also be considered when evaluating performance.

To provide a comparable performance measure, the two metrics Youden's index[7] and F-score were considered. Youden's index has been used in the OWASP Benchmark project and combines the true positive rate with the false positive rate to a single metric, reflecting the overall ability of the tool. However, in the evaluation of a package detection system, it is difficult to define a true negative, as it represents the absence of a package that is not present in the tested code. Since it is not feasible

[7] https://owasp.org/www-project-benchmark/

to estimate the number of packages absent from a given codebase, the true negative value remains unknown. Therefore, this thesis prioritizes transparent reporting of the raw classification outcomes rather than aggregating performance into a single composite metric.

Instead, the evaluation primarily focuses on the classification outcomes defined in Table 4.1. As described previously, the package detection system returns both detected packages and their corresponding CVEs. Consequently, the evaluation can be performed either at the package level or at the CVE level. Since the primary focus of this thesis is the package detection component itself, and not the package-to-CVE translation, the system is mainly evaluated using detected packages rather than detected CVEs. Comparing packages directly evaluates the ability of the system to extract kernel-level information and accurately map it to packages. However, this approach treats all detected packages equally, regardless of the number of vulnerabilities associated with each package.

Therefore, the evaluation will also include a limited comparison based on the Common Vulnerabilities and Exposures (CVE) entries associated with the detected packages. A single package may be associated with multiple CVE entries. Consequently, when comparing results at the CVE level, the absence of a package linked to many vulnerabilities has a greater impact on the evaluation outcome than the absence of a package with no known vulnerabilities. This weighting effect may be considered desirable from a security perspective, since identifying packages associated with many vulnerabilities is generally more important than detecting packages with no known vulnerabilities.

5

Results

This chapter is structured into seven main sections. The first sections presents a comparison between different baseline candidates. The second section compares data gathering approaches using eBPF. Then, results from different tracing strategies is presented. The fourth section will present results obtained when applying SnakeBPF on five different web applications. The next section will present results of the proposed library detection message in a multi container environment. The sixth section presents results from a Kubernetes cluster followed by results from running the detection method for an extended period of time in order to evaluate the impact of Python's in-memory cache.

This chapter frequently use Venn diagrams to present selected results. The numbers in the Venn diagrams represent the number of packages detected uniquely or jointly by the different methods. For completeness and disclosure, all results and raw data used for each figure are available in our public Github repository[1].

5.1 Baseline Candidates

This section presents the detected packages for the evaluated baseline candidates. For all results in this section, packages are detected while the open-source application Wagtail Bakery Demo is subjected to web scanning using ZAP.

5.1.1 Overriding the Python Import Mechanism

The packages detected by the three proposed Python-based approaches for obtaining a baseline are shown in Figure 5.1. The figure shows that all three methods share a common subset of detected packages. Furthermore, all packages detected by `find_load` tracing are also detected by the meta path finder approach. These results show that `find_load()` does not contribute with any unique packages. Both `__import__` statement overriding and meta path finder tracing identify additional unique packages.

Figure 5.2 shows the comparison between packages detected by Trivy and two dy-

[1] `https://github.com/alicethornell/SnakeBPF`

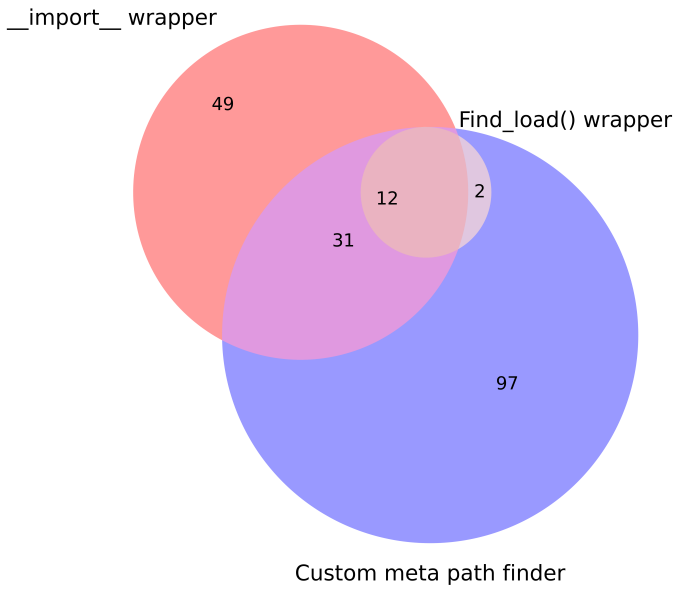


Figure 5.1: A Venn diagram showing which packages were identified by `__import__` statement overriding, `find_load` overriding, and the addition of a custom meta path finder.

namic baseline candidates, i.e., wrapping the `__import__` statement and adding a custom meta path finder. The figure indicates that the dynamic baseline candidates identify a larger set of packages compared to the static approach. A subset of packages is shared across all methods, and both dynamic approaches detect packages not present in the static analysis results. This figure is surprising, as many packages detected by the static analysis tool Trivy are not found by either dynamic baseline candidates. These findings are unexpected since the `__import__` wrapper and custom meta path finder are predicted to be triggered each time a package is imported. We discuss potential reasons for this in Section 6.1.

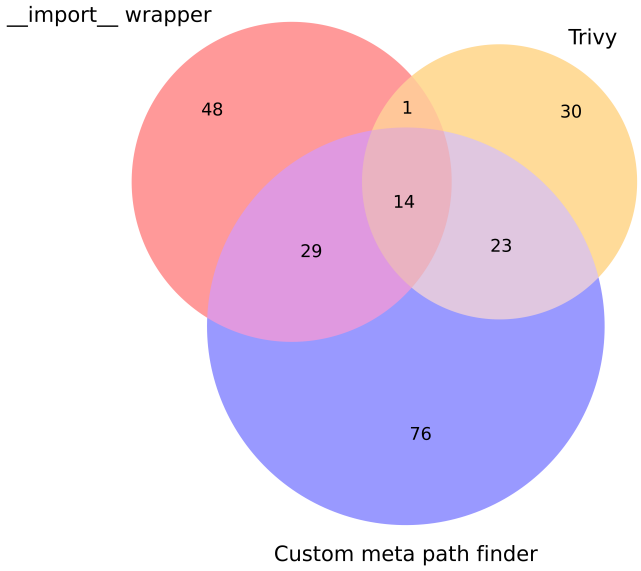


Figure 5.2: A Venn diagram showing which packages were identified by `__import__` statement overriding and a custom meta path finder in relation to Trivy static analysis results.

5.1.2 User-space Probing

Figure 5.3 illustrates the overlap between packages detected using Python/C API tracing and static analysis results from Trivy. The results show that Python/C API tracing and static analysis exhibit only partial overlap in the detected packages. A substantial number of packages identified by Python/C API tracing are not detected by static analysis. A majority of these packages are categorized as standard Python libraries, and are therefore of little interest in regards to vulnerability prioritization as discussed in Section 6.1).

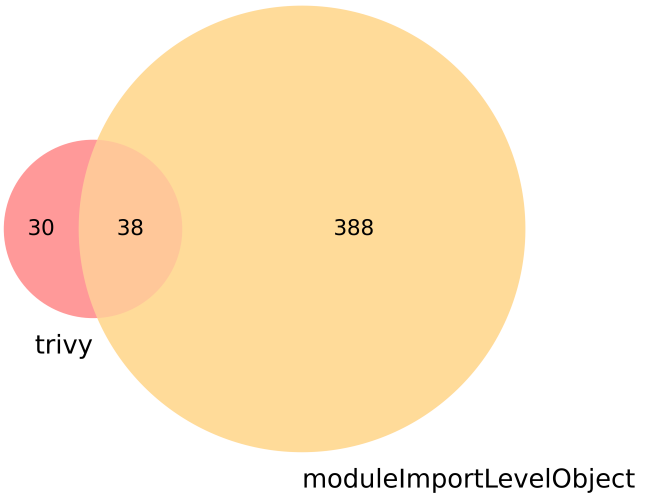


Figure 5.3: A Venn diagram showing which packages were identified by Python/C API tracing and Trivy.

5.2 Data Gathering Approaches

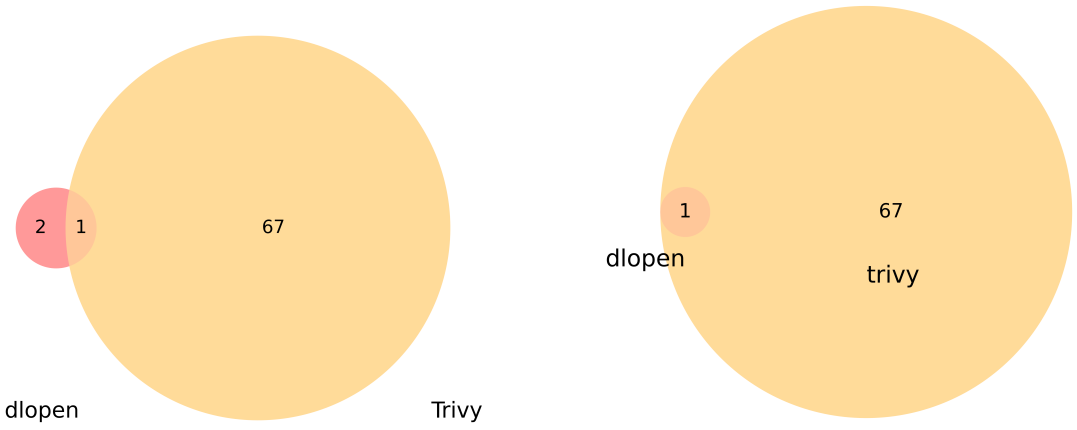
This section presents the results obtained using user- and kernel-space probes described in Section 3.1, retrieved from conducted experiments using the Wagtail Bakery Demo web application. To evaluate the effectiveness of the different data gathering approaches two sets of experiments were conducted to simulate different workloads. In the first experiment, the Wagtail Demo application was subjected to web scanning using ZAP to trigger as many packages as possible. During the second test, the application was triggered manually to exercise only a subset of its functionality.

5.2.1 User-space Tracing

In Section 3.1.1, two userspace tracing methods are presented: tracing of the `dlopen` function and tracing of the Python/C API. As previously discussed, tracing of the Python/C API was considered too volatile for production use. Consequently, this section presents only the results obtained using `dlopen` tracing.

ZAP evaluation

The packages detected using `dlopen` function tracing are presented in Figure 5.4. In this experiment, the Wagtail application was subjected to ZAP web scanning to maximize package utilization within the application. The results show that `dlopen` tracing detects only one of the packages identified by Trivy. In addition, `dlopen` tracing identified two extra packages that were not detected through static analysis. However, these packages were removed after applying the `site-packages` filter in the processing component, as shown in Figure 5.4b. Overall, the dlopen tracing method failed to detect 67 of the 68 packages identified through static analysis.



(a) The number of detected Python packages when considering all Python system paths. Red represents packages detected by `dlopen` and yellow by `Trivy`.

(b) The number of detected Python packages from the `site-packages` directory. Red represents packages detected by `dlopen` and yellow by `Trivy`.

Figure 5.4: The number of detected packages during ZAP web scanning load. `dlopen` userspace tracing is compared with the static analysis tool Trivy. Overlapping regions indicate packages detected by both methods.

Targeted Functionality Evaluation

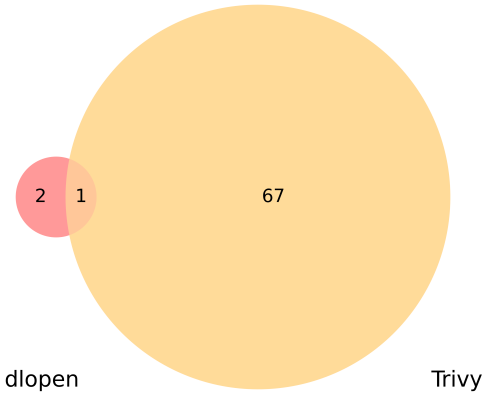
The packages detected by `dlopen` tracing when applied on the Wagtail Bakery Demo subjected to targeted functionality testing are depicted in Figure 5.5. Comparing these results with those obtained during web scanning shows that the detected package sets are identical. This result is notable, since web scanning was expected to exercise a larger portion of the application and therefore trigger the loading of additional packages compared to the targeted functionality workload.

5.2.2 Kernel-space Tracing

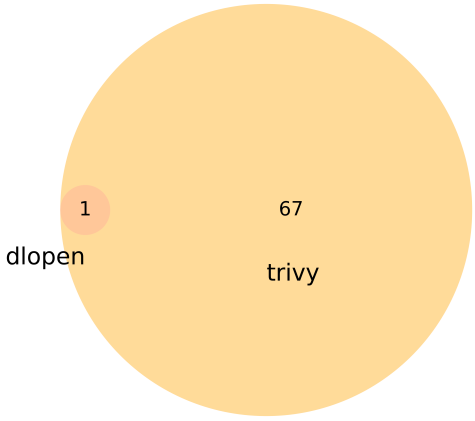
This section presents the results obtained for kernel-space tracing of different system calls, following the methodology described in Section 3.1.2.

ZAP Evaluation

Figure 5.6 shows the number of detected packages for the three evaluated data collection methods: `open`, `read`, and `mmap` system call tracing, as well as their overlap. The Wagtail Bakery Demo was traced in all three cases before container deployment, and terminated after ZAP web scanning. Figure 5.6a shows the results when all Python system paths are included in the post process filtering, while Figure 5.6b shows the results when only packages from the `site-packages` directory are considered. A complete list of the packages in Figure 5.6b is provided in Table B.1.

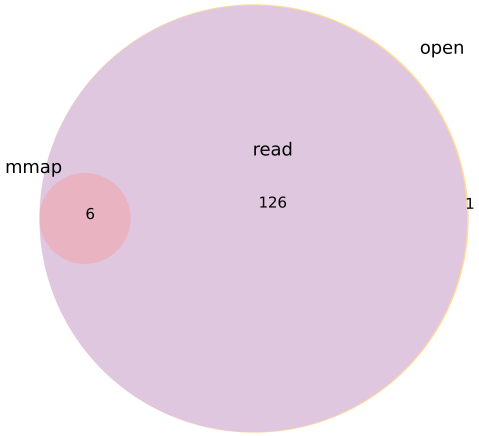


(a) The number of detected Python packages when considering all Python system paths. Pink represents packages detected by `dlopen` and yellow by `Trivy`.

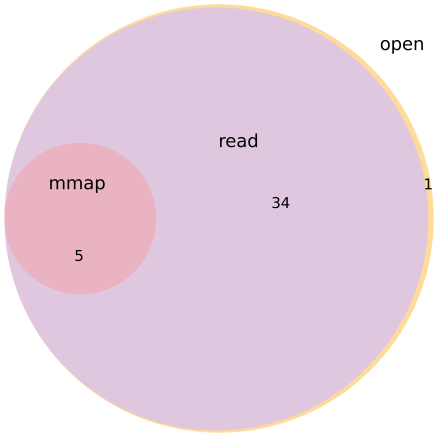


(b) The number of detected Python packages from the `site-packages` directory. Pink represents packages detected by `dlopen` and yellow by `Trivy`.

Figure 5.5: The number of detected packages during targeted functionality load by using `dlopen` userspace tracing. These results are compared with the static analysis tool Trivy. Overlapping regions indicate packages detected by both methods.



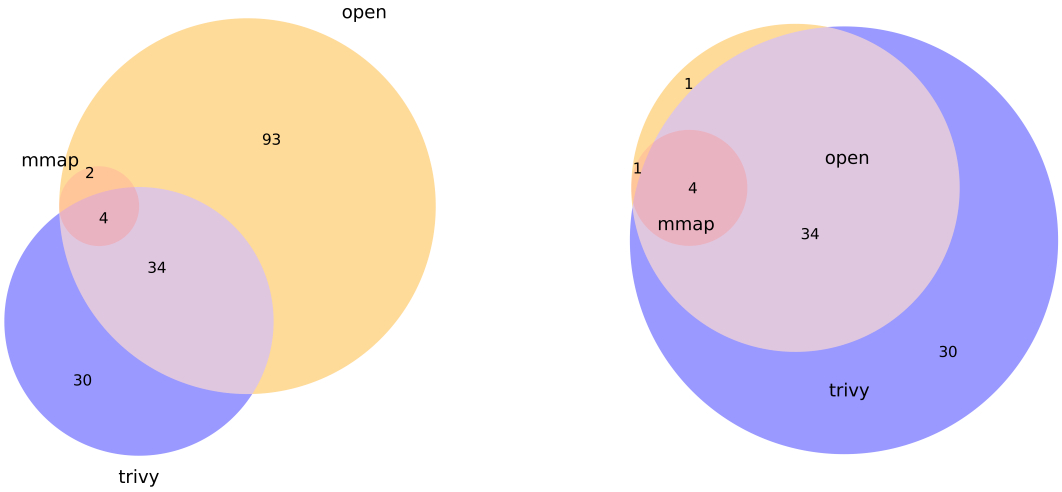
(a) The number of detected Python packages when considering all Python system paths. Pink represents packages detected by `mmap`, purple by `read`, and yellow by `openat`.



(b) The number of detected Python packages from the `site-packages` directory. Pink represents packages detected by `mmap`, purple by `read`, and yellow by `openat`.

Figure 5.6: The number of detected packages when tracing system calls `mmap`, `openat`, and `read`. Each region indicates how many packages were uniquely or jointly detected by the traced system calls.

The results in Figure 5.6 show that `openat` and `read` system call tracing yield close to identical results, as `openat` detects one additional package. The results also indicate that all packages detected by `mmap` were also detected by `read` and `openat`. Restricting the analysis to the `site-packages` directory reduces the total number of detected packages in both subsets. Figure 5.7 shows the overlap between packages detected by `openat`, `read`, and the Trivy static analysis tool. As seen in Figure 5.6, `openat` and `read` trace almost the same number of packages. Because of this, the package subset represented by `openat` was included in Figure 5.7, as the `openat` trace included one additional package.



(a) The number of detected Python packages when considering all Python system paths. Pink represents packages detected by `mmap`, blue by `Trivy`, and yellow by `openat`.

(b) The number of detected Python packages from the `site-packages` directory. Pink represents packages detected by `mmap`, blue by `Trivy`, and yellow by `openat`.

Figure 5.7: The number of detected packages when tracing system calls `mmap` and `openat` during automated testing of Wagtail's Bakery Demo using ZAP, compared with packages detected by Trivy. Each region indicates how many packages were uniquely or jointly detected by the different methods.

The results in Figure 5.7 show that all packages detected by Trivy and `mmap` are also detected by `openat`. When restricting the analysis to packages in the `site-packages` directory, the number of packages detected by `mmap` and `read` that are not identified by Trivy is reduced substantially. To determine the composition of the detected packages when using all search module paths in comparison to only packages in the `site-packages` directory, all standard libraries were removed. The results after this filtering are presented in Figure 5.8.

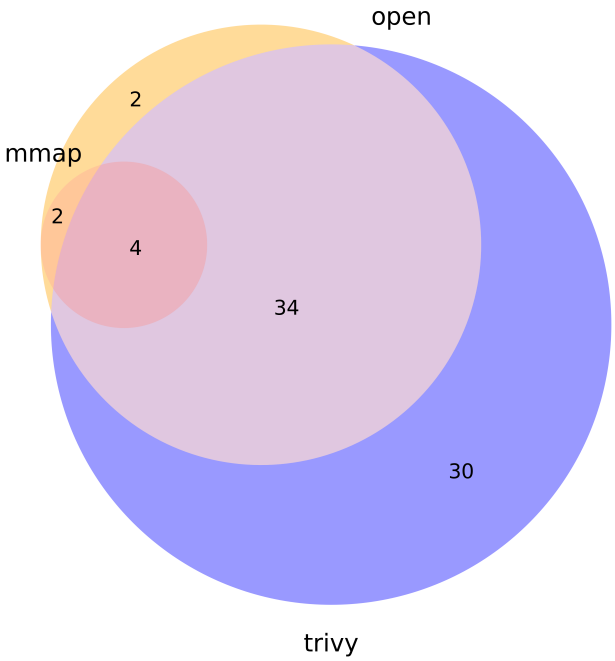


Figure 5.8: The number of detected Python packages when considering all Python system paths and removing standard libraries. Pink represents packages detected by `mmap`, blue by `Trivy`, and yellow by `openat`.

Targeted Functionality Evaluation

Following the approach described in Section 4.2.2, a targeted functionality test was conducted to determine which kernel-space tracing method generates the largest subset of packages that were also identified by Trivy for the Wagtail application. Based on the full trace revealing the exact packages identified by Trivy and the `openat` and `mmap` system calls, the Venn diagram in Figure 5.9 showcase the overlap between the different methods. A complete list of the packages in Figure 5.9 is provided in Table B.2.

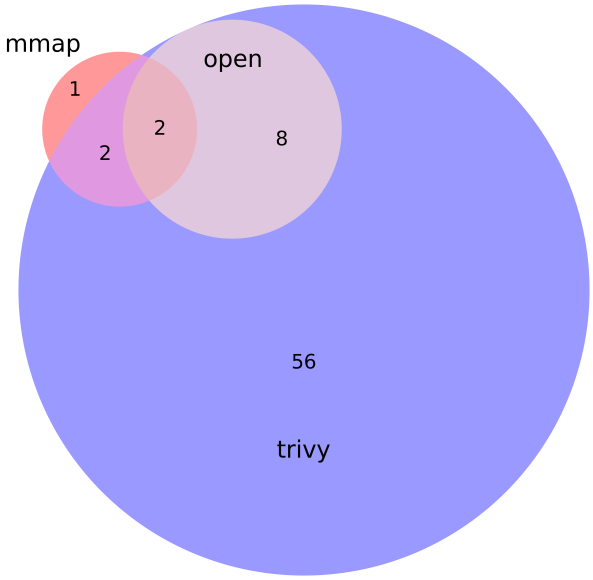


Figure 5.9: Packages identified after tracing the `mmap` (red) and `openat` (dusty pink) system call during targeted testing of Wagtail Bakery Demo. Each region pictures how many packages were uniquely identified by each method and how they differ compared to static Trivy results (blue).

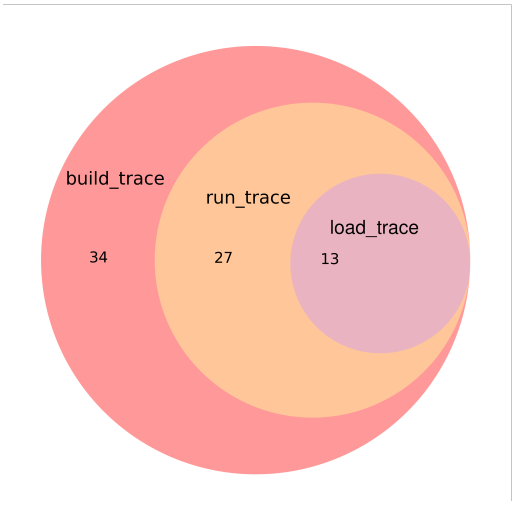
5.3 Tracing Strategy Selection

This section presents the results obtained by tracing system calls during different life cycle stages of a containerized web application, as described in Section 4.4.3. All results in this section were obtained by tracing web applications under the web scanning workload described in Section 4.2.1. The Plone web application is excluded from some experiments because the container image was already prebuilt and used without modification. Consequently, the image build phase could not be traced. All detected packages depicted in this section, i.e. in Figure 5.10, Figure 5.11 and Figure 5.12 are found in Appendix C.

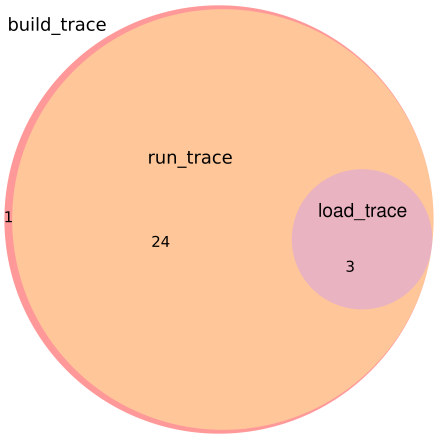
Figure 5.10 presents Venn diagrams illustrating the libraries detected for each web application using the three tracing strategies. For the Bakery Demo, Docker-Django, and Moments applications, the detected packages follow the same overall pattern: tracing during the build phase detects the largest number of packages, while tracing during deployment detects a subset of those packages. Finally, tracing during the ZAP web scanning phase detects the fewest packages, all of which are already detected during deployment tracing and image build tracing.

These results suggest that tracing during the build phase alone may be sufficient for package detection, since continuous tracing during later life cycle phases does not contribute additional packages. However, the Todoism application does not follow

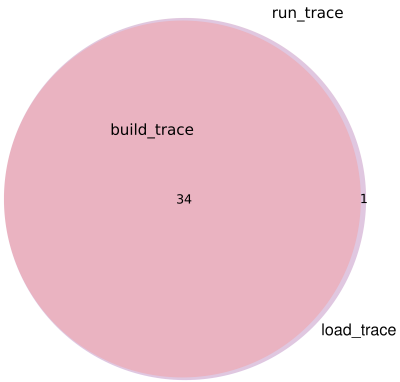
the same pattern. For this application, tracing during deployment and web scanning identifies one additional package. Overall, the different tracing phases produce similar results for Todoism, in contrast to the other applications where restricting tracing to later phases substantially reduces the number of detected packages.



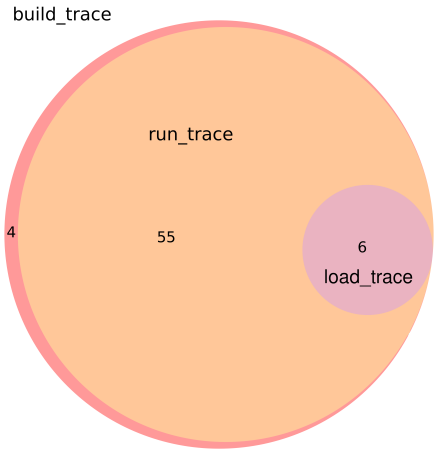
(a) Detected Python libraries for Wagtail's Bakery Demo.



(b) Detected Python libraries for Docker-Django.



(c) Detected Python libraries for Todoism. `load_trace` and `run_trace` contains one library not detected in `build_trace`.

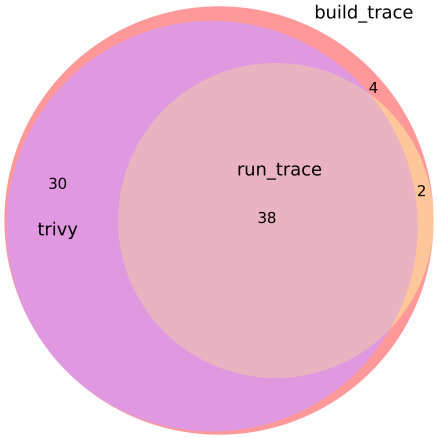


(d) Detected Python libraries for Moments.

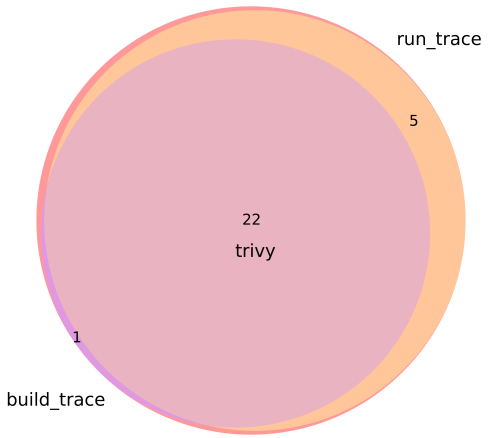
Figure 5.10: Venn diagrams that showcase detected Python libraries during building the container image, running the container, and web scanning of each application.

Figure 5.11 presents Venn diagrams comparing packages detected during the build and deployment phases of the container life cycle with packages identified through static analysis using Trivy. For all applications, the packages detected during the build phase fully cover those identified by Trivy. Although some additional packages are detected, the amount of noise remains limited. For all applications except the Wagtail Bakery Demo, packages detected during the deployment phase also fully cover the Trivy baseline. For the Wagtail Bakery Demo, 30 packages identified by

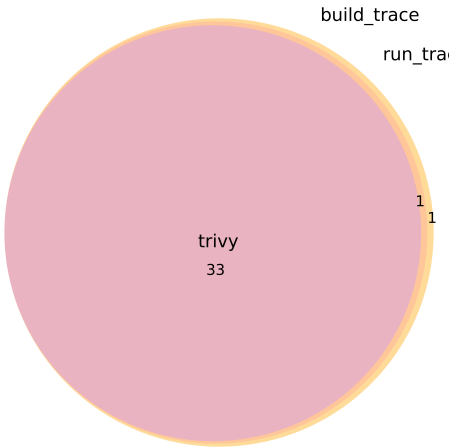
Trivy are not detected during deployment tracing. These packages were also absent from the baseline candidate set, as depicted in Section 5.1 and Figure 5.2.



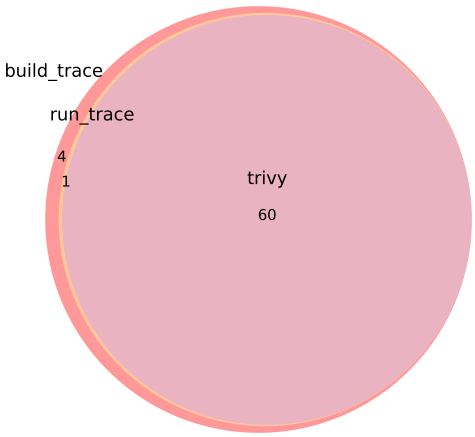
(a) Detected Python libraries for Wagtail's Bakery Demo. Python packages detected by Trivy are purple, `run_trace` is yellow and `build_trace` is red.



(b) Detected Python libraries for Docker Django app. Python packages detected by Trivy are purple, `run_trace` is yellow and `build_trace` is red.



(c) Detected Python libraries for Todoism app. Subsets overlap almost entirely. Python packages detected by Trivy are purple, `build_trace` is orange and `run_trace` is yellow.



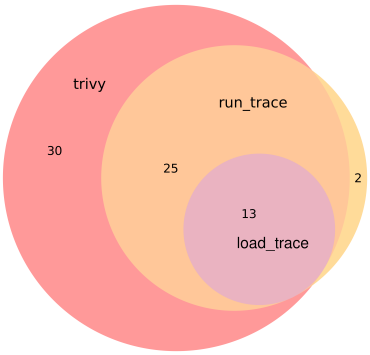
(d) Detected Python libraries for Moments app. Python packages detected by Trivy are purple, `run_trace` is yellow and `build_trace` is red.

Figure 5.11: Venn diagrams showcase detected Python libraries during building the container image and running the container. These two trace strategies are compared with results from Trivy static analysis.

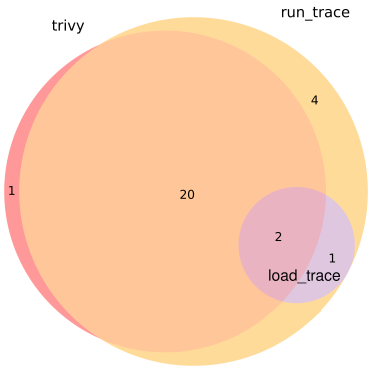
In Figure 5.12, packages detected during deployment and web scanning are compared directly against the Trivy results. The additional comparison introduced in these

diagrams concerns packages detected exclusively during the web scanning phase relative to the static analysis baseline.

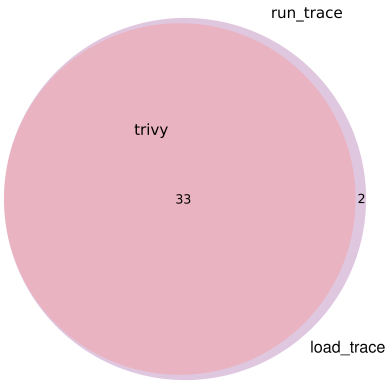
The results show that for the Bakery Demo, Docker-Django, and Moments applications, tracing during the web scanning workload identifies substantially fewer packages than static analysis. Once again Todoism introduce an exception, where applying SnakeBPF during deployment and web scanning yields more detected packages than Trivy. For Plone, one package identified by Trivy is not detected when applying SnakeBPF during the web scanning phase. Since Trivy identifies substantially fewer packages for Plone overall, the relative proportion of noise becomes significantly larger compared to the other applications.



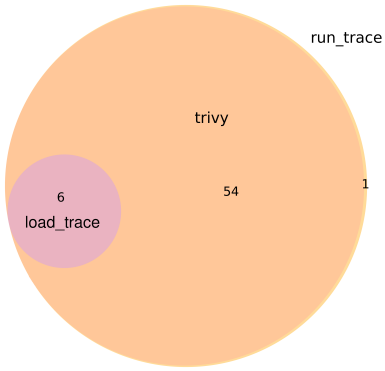
(a) Detected Python libraries for Bakery Demo.



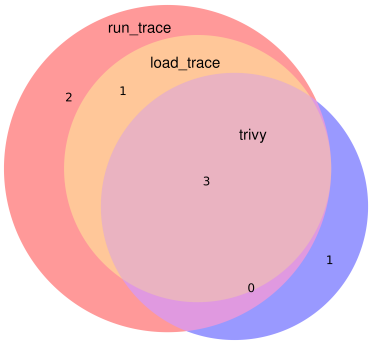
(b) Detected Python libraries for Docker Django.



(c) Detected Python libraries for Todoism. `load_trace` and `run_trace` contain two packages not found by Trivy.



(d) Detected Python libraries for Moments.



(e) Detected Python libraries for Plone. The red circle are packages contained in run_trace and blue are packages detected by Trivy.

Figure 5.12: Venn diagrams showcasing the number of detected Python libraries when running containerized web applications and subjecting web scanning using ZAP. The results are compared to results from the static analysis tool Trivy.

5.4 Web Application Evaluation

This section presents the results obtained when evaluating the proposed approach, SnakeBPF, on the open-source web applications introduced in Section 4.1. Two experiments were conducted: the applications were subjected either to automated web scanning using ZAP or to targeted functionality workloads, as described in Section 4.2. All packages presented in this section are found in Appendix D.

5.4.1 ZAP Evaluation

This section presents the packages detected by SnakeBPF when applied to five web applications subjected to automated web scanning. The purpose of the web scanning workload is to maximize package utilization within each application.

Table 5.1 summarizes the number of detected packages for each application and compares the results against the static-analysis baseline provided by Trivy. In this experiment, SnakeBPF was applied after the container image had been built but before the application container was deployed. Consequently, the tracing covers the deployment and load phases described in Section 4.4.3. In Table 5.1, the first column shows packages uniquely detected by SnakeBPF. The second column contains packages detected by both SnakeBPF and Trivy, while the third column contains packages detected exclusively by Trivy.

Since the applications were subjected to extensive web scanning intended to maximize package utilization, most packages are expected to be exercised and therefore detectable. Consequently, strong performance from SnakeBPF should result in a large overlap with Trivy (second column) and relatively few packages detected exclusively by Trivy (third column). The first column may represent either packages actively used at runtime but missed by Trivy, or noise introduced by SnakeBPF, and should therefore remain as small as possible.

The results in Table 5.1 show that, for all applications except Wagtail's Bakery Demo, SnakeBPF detects either all or the vast majority of packages identified by Trivy, indicating strong overall performance. The amount of noise, represented by packages uniquely detected by SnakeBPF, remains relatively small, although still non-negligible.

5.4.2 Targeted Functionality Evaluation

This section presents the detected packages when the five web applications are subjected to targeted functionality workloads, as described in Section 4.2. As in the previous experiment, only the deployment and workload execution phases were monitored. The results are presented in Table 5.2 using the same format as in the previous section. However, the interpretation of the columns differs in this experiment because not all packages are expected to be exercised during targeted functionality testing.

Table 5.1: Coverage comparison of detected Python packages between SnakeBPF (A) and Trivy (B) when scanning selected web applications using ZAP. The cells contain three numbers: Unique packages detected by SnakeBPF ($A \setminus B$), packages detected by both ($A \cap B$), and packages detected by Trivy ($B \setminus A$).

Web app.	$A \setminus B$	$A \cap B$	$B \setminus A$
Wagtail Bakery Demo	2	38	30
Docker-Django	4	22	1
Moments	1	60	0
Todoism	2	33	0
Plone	3	3	1

In this setup, a high number of packages in the second column ($A \cap B$) does not necessarily indicate strong performance. Since only a subset of application functionality is intentionally triggered, fewer packages are expected to be detected. This distinction is important for the intended purpose of SnakeBPF, namely vulnerability prioritization among Python packages. Detecting an excessively large number of packages could indicate that the approach identifies packages that are installed but not actively used at runtime.

The results in Table 5.2 show that, for most applications, SnakeBPF detects only a subset of the packages identified by Trivy. As discussed above, this outcome is expected because the targeted workloads are not intended to exercise all application functionality. However, for Todoism and Plone, SnakeBPF still detects either all or the vast majority of packages identified by Trivy.

Table 5.2: Coverage comparison of detected Python packages between SnakeBPF (A) and Trivy (B) when evaluating a targeted web application's functionality. The cells contain three numbers: Unique packages detected by SnakeBPF ($A \setminus B$), packages detected by both ($A \cap B$), and packages detected by Trivy ($B \setminus A$).

Web app.	$A \setminus B$	$A \cap B$	$B \setminus A$
Wagtail Bakery Demo	0	10	58
Docker-Django	1	2	21
Moments	0	6	54
Todoism	2	33	0
Plone	1	3	1

5.4.3 Vulnerability Detection

The package detection results from Section 5.4.1 were further translated into potential CVEs, as described in Section 3.3. Table 5.3 compares the vulnerabilities identified using SnakeBPF under targeted functionality workloads with those identified by the static analysis baseline Trivy.

As described in Section 4.5, translating package detections into CVEs serves multiple purposes. First, it demonstrates how runtime-detected packages can be mapped to

known vulnerabilities. Second, it introduces a weighting effect, where packages associated with vulnerabilities become more significant than packages without known CVEs. Consequently, the key observation in Table 5.3 is whether the relationship between detections from SnakeBPF and Trivy changes when evaluated at the vulnerability level rather than the package level.

The results in Table 5.3 show that, for Wagtail's Bakery Demo and Todoism, SnakeBPF identifies three vulnerabilities not detected by Trivy. In contrast, for Moments, and particularly for Plone, Trivy identifies a substantial number of vulnerabilities that are not detected by SnakeBPF.

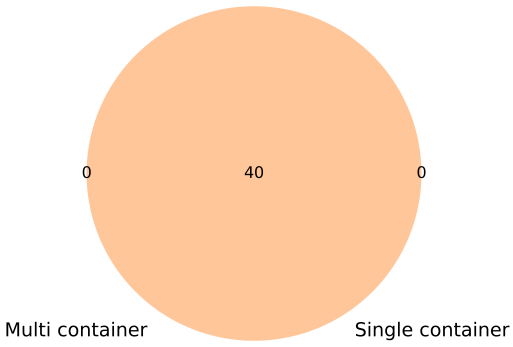
Table 5.3: Coverage comparison of detected CVEs between SnakeBPF (A) and Trivy (B) when evaluating a targeted web application's functionality. The cells contain three numbers: Unique CVEs detected by SnakeBPF ($A \setminus B$), CVEs detected by both ($A \cap B$), and CVEs detected by Trivy ($B \setminus A$).

Web app.	$A \setminus B$	$A \cap B$	$B \setminus A$
Wagtail Bakery Demo	3	5	5
Docker-Django	0	3	8
Moments	0	2	27
Todoism	3	52	1
Plone	0	7	78

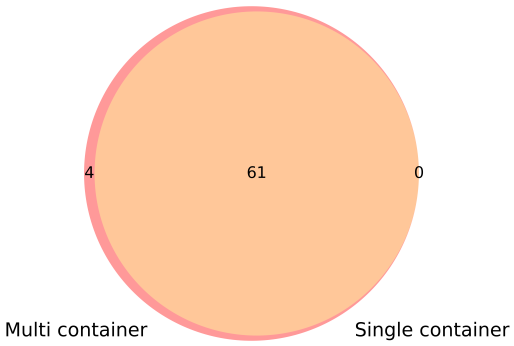
5.5 Multi Container Support

To evaluate whether the data-processing component presented in Section 3.2.2 can correctly map Python packages to their corresponding containers, two applications were executed simultaneously, as described in Section 4.4.5. The results are presented in Figure 5.13. The purpose of this experiment is to determine whether the packages detected by SnakeBPF are affected by execution in a multi-container environment. Good performance in this context means that the packages detected for a the application in a multi-container environment should match those detected when the application is executed in isolation.

For the Bakery Demo application, the proposed detection strategy identifies exactly the same packages regardless of whether the application is executed in isolation or in a multi-container environment. For the Moments application, four additional Python packages are detected when tracing is performed in the multi-container environment. These packages are listed in Table 5.4. The packages detected in the single container scenario are found in Appendix C, as data gathering during the deployment and ZAP phase are combined.



(a) Python packages detected for the Bakery Demo subjected to ZAP.



(b) Python packages detected for the application Moments subjected to ZAP.

Figure 5.13: Venn diagram displaying the number of detected packages for the applications Moments and Bakery Demo when subjected to ZAP. The multi container label indicates that the detection method was applied when the two applications were run simultaneously in two containers. The single container label indicates that the application was the only container on the system during library detection.

Table 5.4: Python packages detected in a multi container environment that were not detected when Moments is the only running container.

Package:
attr
pkg_resources
websocket
yaml

5.6 5G Core Kubernetes Cluster Evaluation

This section present the results obtained when tracing package usage inside a Kubernetes cluster running 5G packet core software. The results after performing the experiment (see Section 4.2.3) are presented in Table 5.5, where custom interactions and Syft subsets are visualized in Figure 5.14. The packages identified by SnakeBPF, targeting a pod, is compared with a container SBOM generated by Syft.

Table 5.5: Number of detected packages after tracing Python package usage by 5G packet core services. **Default traffic** corresponds to the number of detected packages used by a targeted pod during normal network traffic. **Custom interactions** capture the number of detected packages used by the targeted pod when rescheduled and receiving curl requests. **Curl** showcase the number of packages used by the pod when receiving curl requests. **Syft** corresponds to the number of detected packages after generating a SBOM for a container image.

	Default traffic	**Custom Interactions**	**Curl**	**Syft**
Detected Packages	3	67	2	55

Surprisingly, it seems like SnakeBPF manage to detect more packages during custom interactions than Syft. This goes against the assumption that static analysis yields more results than dynamic analysis. After inspecting the 12 packages uniquely detected by SnakeBPF, 5 packages are categorized as internal Ericsson packages and 3 packages are used implicitly whenever an API call is made using a package identified by both methods. Finally, 2 packages are likely detected subpackages and the last 2 are considered to be noise, as they were located inside the `/site-packages` directory but are neither listed as internal nor found at an official repository for third-party Python packages. Considering these findings, the two methods yield almost identical package detection coverage in this scenario. However, SnakeBPF introduce a small but non-negligible additional noise.

To isolate the curl events from pod rescheduling, the traces were cut at a predefined checkpoint present in both `execve` and `openat` outputs. These results were obtained from the same trace session as the pod rescheduling event. The resulting curl traces are compared with Syft in Figure 5.15. In this case, SnakeBPF identifies only two packages used by the pod, whereas one package is considered to be noise. The SnakeBPF package coverage is evidently much smaller than Syft in this case,

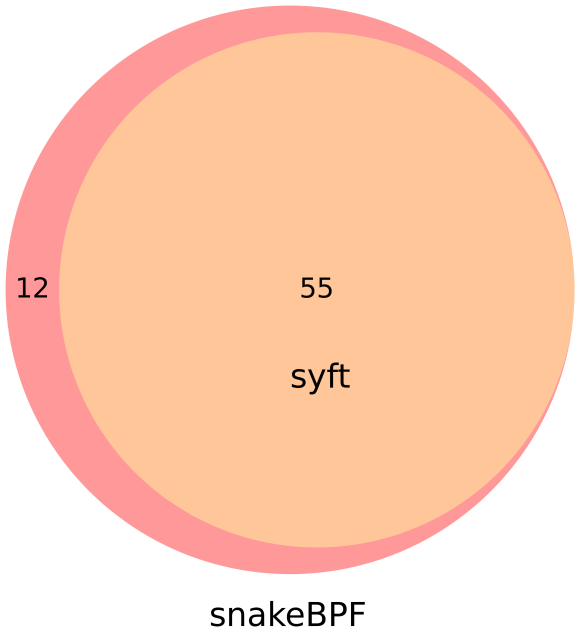


Figure 5.14: Uniquely and mutually identified packages by SnakeBPF and Syft, after tracing a targeted pod inside a Kubernetes cluster.

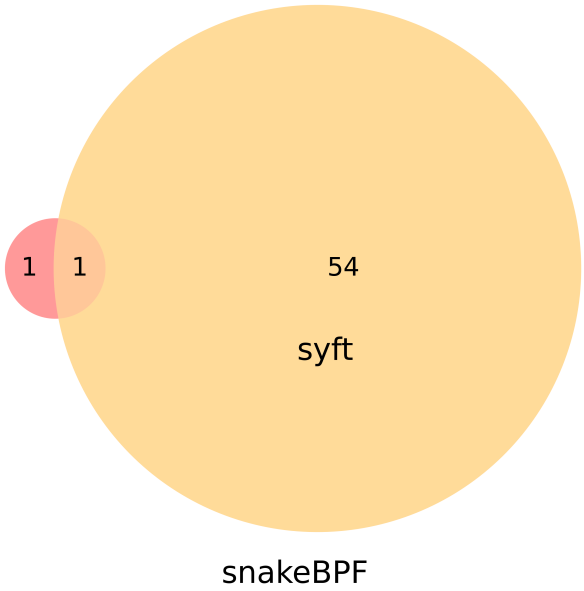


Figure 5.15: Uniquely and mutually identified packages by SnakeBPF and Syft, after tracing a targeted pod handling curl requests.

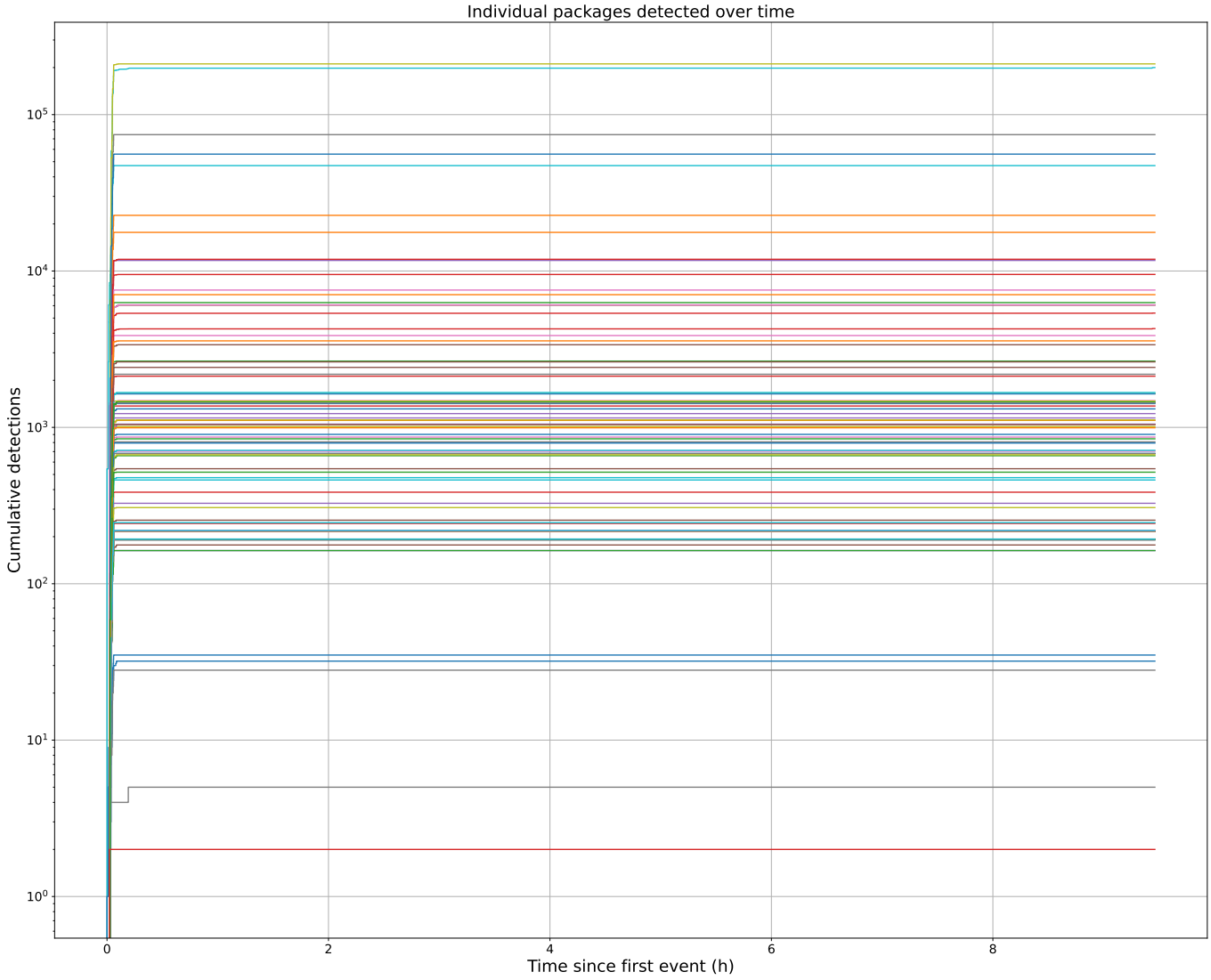
indicating that the number of packages detected by SnakeBPF is highly correlated with dynamic application behavior.

5.7 In-memory Cache Impact

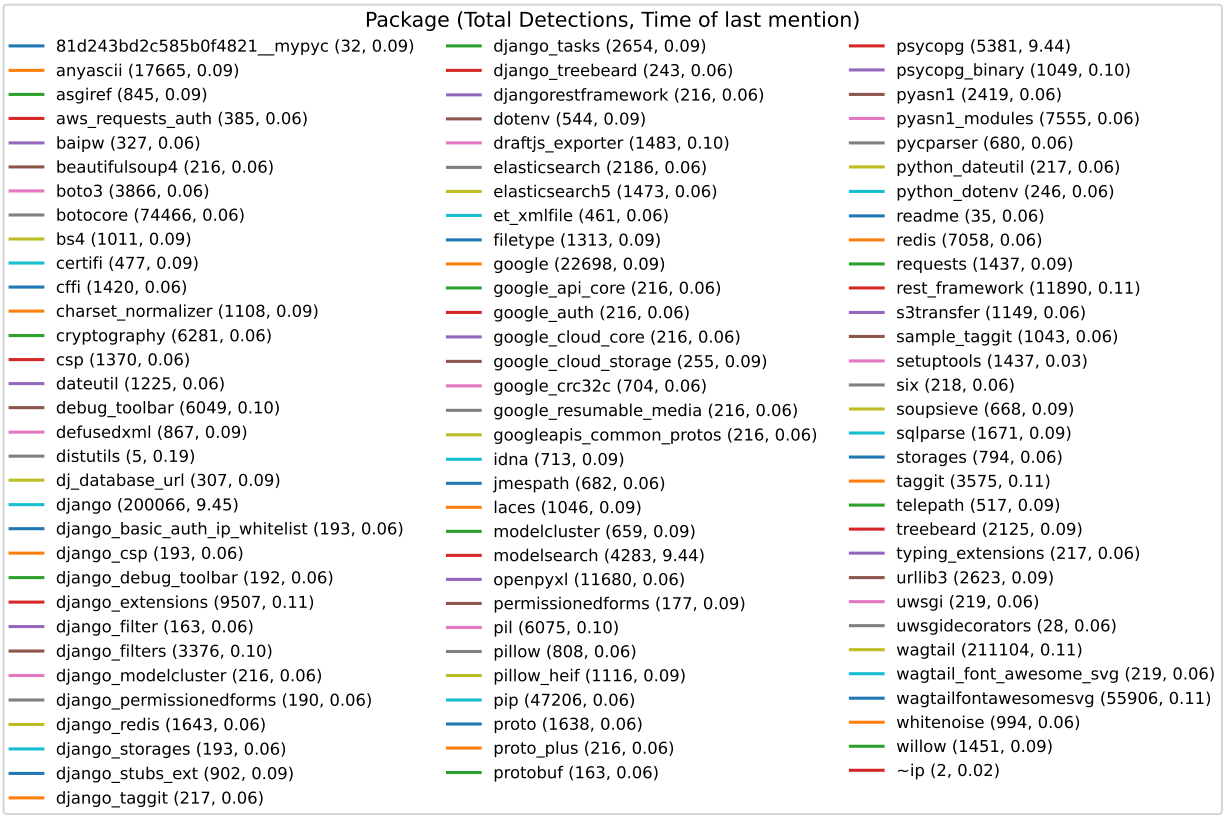
To evaluate the impact of Python's in-memory cache, web scanning was repeatedly applied as described in Section 4.4.6. The results are presented in Figure 5.16, which illustrates the number of detections for each package individually.

The number of detections increases sharply during the initial phase of execution, when the container image is built, deployed, and subjected to web scanning. The figure legend includes the package names, the total number of detections, and the timestamp of the final detection for each package. As shown in the legend, the final detection for most packages occur within the first hour of execution, with a few exceptions of packages being detected around the time of the second web scan.

In addition to the proposed method SnakeBPF, web scanning was also repeated while tracing the `read` system call. The corresponding results are presented in Figure 5.17. Similar to the previous experiment, most packages are last detected within the first hour of execution. However, a small number of packages continue to be detected during the second round of web scanning.

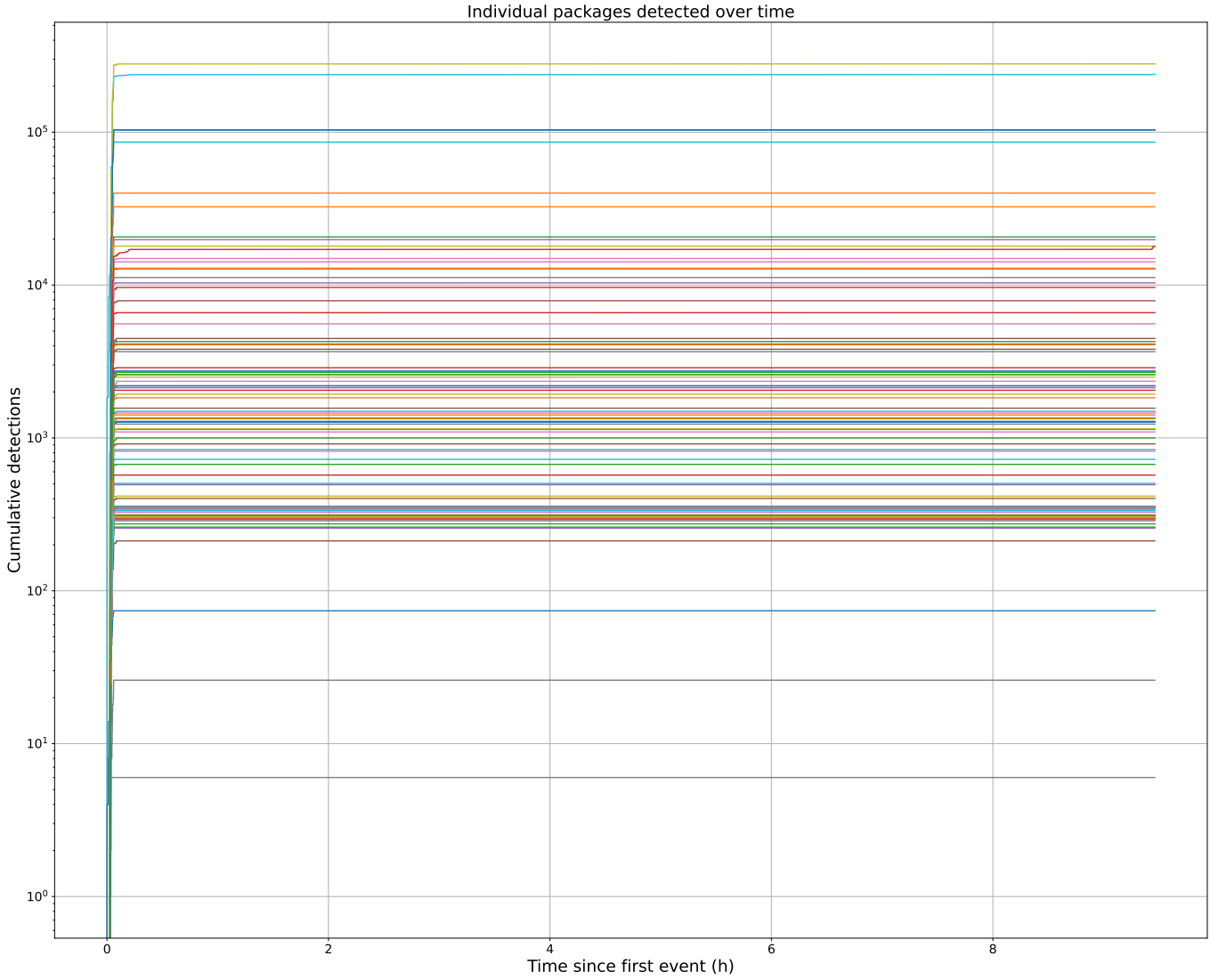


(a) Number of detections per detected Python package over time with `openat` system call tracing.

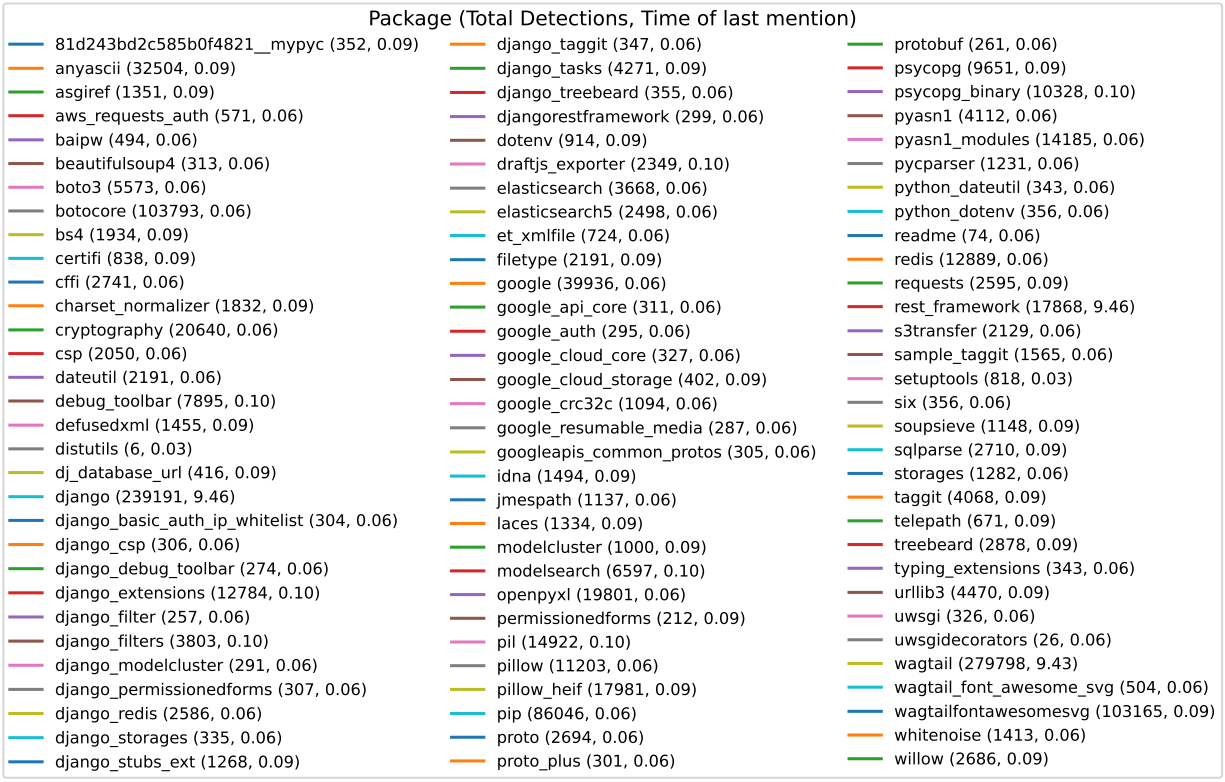


(b) Legend to a), including total number of detections and time of last detection (hours since start)

Figure 5.16: Individual packages detected with `openat` system call tracing when running the Wagtail Bakery Demo for 9 hours. The application was subjected to web scanning, once after container build and once after approximately 9 hours.



(a) Number of detections per detected Python package over time detected with `read` system call tracing.



(b) Legend to a), including total number of detections and time of last detection

Figure 5.17: Individual packages detected with read system call tracing when running the Wagtail Bakery Demo for 9 hours. The application was subjected to web scanning, once after container build and once after approximately 9 hours.

6

Discussion

This chapters examine the results presented in Chapter 5 and how these relate to the stated research questions. In some cases where there are know limitations with SnakeBPF, this chapter explain how these affect the reported results and what the future steps are to build a more robust methodology. The chapter begins with a discussion on how a reliable baseline for method evaluation can be found. Then, different candidate methods are compared for the final approach. The chapter also contains discussion around the final approach.

6.1 Baseline Selection

Figure 5.1 shows the relationship of detected packages for the three Python-based baseline contenders. All packages detected by `find_load` were also detected by the custom meta path finder, which identifies substantially fewer packages than the other two methods. As the number of packages is far from what we expect to be the baseline, we can conclude that `find_load` wrapping is most likely not a great baseline candidate.

The overlap between the packages detected by ___import___ statement wrapping, Trivy and meta path finder tracing is relatively small. In Figure 5.2, these two dynamic methods are compared with Trivy. It is noteworthy that none of the Python import methods capture all packages detected by Trivy. Instead, 30 packages are detected exclusively by Trivy. This suggests that there might exist an entry point to the import mechanisms that none of the evaluated methods capture.

Another alternative explanation is that those 30 packages are not used at all when the application is running. One inherent characteristic of import mechanism wrapping is that tracing does not begin until the main Python program has applied the tracing wrappers. The packages that are not detected could be used only during container build, and it is therefore natural that the candidates for a dynamic baseline do not cover them.

Table A.1 lists the packages detected by Trivy, that the wrapper baseline alternatives do not detect. From this table it is clear that a majority of the packages are not

mentioned in the source code for the application. Without performing a manual analysis of the source code (see project limitations in Section 1.4) we cannot state whether or not these packages are dependencies of other used packages, or falsely detected by Trivy. Other options include that the the packages might falsely be listed in the requirements file or otherwise are included during the build of the container without actually being used. From the table, it is also clear that some packages that Trivy detects, such as *elasticsearch* and *django_csp* are never expected to run for our evaluations, as their use is disabled by default. Because of these uncertainties, we can not state that wrapping the import statement or adding a custom meta path finder would provide a reliable dynamic baseline.

Figure 5.3 illustrates that Python/C API tracing using `PyImport_ModuleImportLevelObject` does not detect all packages identified by Trivy. Upon inspection, it is clear the it is the exact same 30 packages as in the previous figure. This supports the notion that the undetected packages might not be used by a running application, and if they're used then perhaps only during the build stage. The alternative explanation, that those packages pass through another part of the import mechanism still remains. Upon inspection of the C Python source code, it suggests that the API function `PyImport_ModuleImportLevelObject` that is traced uses the normal importlib package. Therefore it is reasonable that tracing `PyImport_ModuleImportLevelObject` does not provide much further coverage of the import mechanism itself. However, it is clear that this type of user-space probing does catch libraries that are present in the `sys.modules` cache, which includes libraries that do not need to be loaded again. This could be a possible explanation to why this tracing method detects vastly more packages.

Since none of the suggested dynamic baselines capture all Trivy packages, they cannot be considered stable baselines. Nevertheless, having a dynamic baseline remains useful for targeted evaluation where only a subset of packages are used. Without the dynamic baseline, it is not possible to conclude that SnakeBPF detects all running packages. As for the current evaluation methodology, Trivy is considered the most reliable and will be used as the baseline for further experiments. If we assume that static analysis results obtained from Trivy contain all packages of an application, we can use it as a ground truth for experiments where all packages are expected to be running. If SnakeBPF does not find all packages detected by static analysis in such experiments, it suggests that SnakeBPF does not have the ability to detect all libraries, therefore indicating a blind spot.

6.2 Approach Selection

This section discuss how the results indicate that kernel-space tracing of the `openat` system call provides coverage of all packages detected by Trivy during ZAP evaluation. Furthermore, the results show indications of including non-site-package directory packages in the processing filtering stage introduces unnecessary noise, which is why filtering the obtained traces is preferable.

6.2.1 Tracing in Kernel or User Space

Based on the results obtained after tracing `dlopen` in user space, it is evident from Figure 5.4 that this method detects substantially fewer packages than Trivy. Since ZAP web scanning was applied during tracing, it is assumed that a maximum package usage should have occurred. Therefore, the findings suggest that if only a single package was detected, this method has limited suitability for obtaining a comprehensive package coverage.

One possible explanation behind the results is that `dlopen` is only invoked when explicitly called in the code, and never when packages are loaded through other mechanisms (see Section 2.6). Furthermore, in C Python, source code is interpreted rather than compiled into C prior to execution. This may have reduced the number of `dlopen` calls made by the user-space application, which in turn yields less tracing opportunities. In other environments such as C Python, this behavior may differ from what this tracing experiment indicates, as code is translated into C before execution. Moving forward, user-space probing will not be considered further as part of the final proposed approach.

6.2.2 Selecting a Kernel-space Probe

As observed in Figure 5.6, `openat` and `read` system call tracing yields close to identical results, with a slight advantage for `openat` in the number of packages detected. Furthermore, all libraries detected by `mmap` were detected by `read` and `openat` system call tracing. As this experiment subject the Wagtail Bakery Demo application to maximum load through ZAP web scanning, it is desirable to see a subset of detected packages as similar to the Trivy results as possible. Since the web scanner is expected to exercise all application functionality, all running packages should be detected. Methods that achieve results similar to Trivy are therefore indicative of comprehensive package coverage. With this in mind, the results presented in Figure 5.7 indicate that `mmap` tracing yields a lower package detection coverage compared to `openat` and `read` system call tracing.

Based solely on these figures, it is not possible to determine a preferred method. However, `openat` system call tracing is slightly easier to implement than `read` tracing. This has to do with the fact that in order to read a file, a file descriptor must be retrieved, which is what the `openat` system call returns after successful completion. Consequently, the `read` system call cannot return successfully without first obtaining a file descriptor from the `openat` system call. Because of this dependency, SnakeBPF considers only tracing `openat` as sufficient.

Figure 5.9 presents results after tracing `openat` and `mmap` during targeted functionality evaluation of Wagtail's Bakery Demo (see Section 4.2.2). The experiment aims to trace only a subset of packages used after triggering a specific application functionality in order to simulate a different application load. In this scenario, all packages detected by `openat` are also detected by Trivy. Like before, tracing `mmap` yields fewer packages detected in total. However, `mmap` is no longer a perfect subset

of `openat`. Instead, `mmap` identifies two additional packages also found by Trivy, and manage to uniquely identify one additional package that was neither traced by `openat` or Trivy. In this case, the uniquely identified package was in fact not a real package and can be excluded as noise, see Table B.2. Table 6.1 depicts the set differences in detected packages.

Table 6.1: Unique packages detected by the `mmap` and `openat` kernel-space probes.

(a) Packages detected by mmap but not openat

Package
charset_normalizer
pillow_heif

(b) Packages detected by openat but not mmap

Package
django
django_debug_toolbar
django_extensions
django_filter
djangorestframework
draftjs_exporter
modelsearch
wagtail

As shown in Table 6.1, all identified packages correspond to legitimate packages. However, it is difficult to determine why some packages are identified by `openat`, but not `mmap` and vice versa based solely on this information. Recall the results presented in Figure 5.6 and Figure 5.7, were packages identified by `mmap` were also identified by `openat`. On closer inspection of the traces (see Table B.1), we can verify that `openat` managed to identify all packages presented in Table 6.1a when tracing an application subjected to ZAP web scanning.

One possible explanation to why `openat` does not find these packages during targeted functionality evaluation, might be related to when the tracing is actually initiated. We know that in order to create a new mapping, `mmap` requires a file descriptor as an argument (see Table 2.2), which is likely obtained through calling `openat`. If the file was opened before tracing, the resulting file descriptor is likely cached and reused when `mmap` is invoked. Consequently, the `mmap` syscall is observed within the tracing window but the `openat` call is not recorded. This explains why `openat` does not detect the packages in Table 6.1a, but manage to identify them in earlier tests using automated ZAP web scanning. These findings indicate that the set of detected packages is strongly influenced by the timing and duration of the tracing stage.

6.2.3 Disregarding Standard Libraries

After restricting the analysis to only consider packages found inside the `site-packages` directory, the number of detected packages by `openat` were significantly reduced as seen in Figure 5.6b and Figure 5.7b. To determine whether the removed libraries were of any significant importance, the packages detected in Figure 5.6a and Figure 5.7a were filtered to disregard standard libraries. The results after performing

this operation is presented in Figure 5.8, where the number of detected packages by tracing `openat` decreased with 68.4% from before. These results were almost identical to the results obtained when considering only `site-packages`. This suggests that the majority of packages removed when restricting the analysis to `site-packages` were mainly standard libraries, which have limited relevance in a vulnerability detection context later on. Moving forward, the proposed approach SnakeBPF will therefore by default consider only `site-packages`.

6.3 Tracing Strategy Selection

To evaluate SnakeBPF, tracing was performed during the three container stages described in Section 4.4.3. These stages correspond to building a container, container deployment, and scanning a container. This section compares the different tracing strategies and contrasts their results with those obtained using the static analysis tool Trivy.

6.3.1 Build, Run and ZAP Tracing Comparison

Figure 5.10 presents Python package detection results for four web applications. Package detection was performed using data collected during either the container build phase, the container deployment phase, or while the container was subjected to web scanning with ZAP. For the applications Bakery Demo, Docker-Django, and Moments, all packages detected during the web scanning phase were also detected during container deployment. Similarly, all packages detected during deployment were also detected during the build phase.

At first glance, these results could suggest that tracing during the build phase alone is sufficient to capture all packages used by the system, rendering deployment and scan tracing unnecessary. However, such a conclusion requires careful consideration of what SnakeBPF actually measures. The approach relies on tracing the `openat` system call and therefore assumes that opening a file belonging to a Python package implies package usage. This assumption is weaker during the build phase, since containers do not share a file system with the host. As a result, Python package files may be opened simply because they are copied into the container image rather than because they are executed or imported.

This limitation is evident from the inclusion of packages known to be disabled. Table A.1 lists some Python packages detected in the Bakery Demo application. The packages *elasticsearch* and *django_csp* are present in the source code but disabled by default. Consequently, they are never used during the testing scenarios and are therefore not expected to appear in the tracing results. However, both packages are detected during build tracing, while neither appears in the deployment or ZAP tracing results. This suggests that the package files are opened for purposes other than runtime usage, such as transfer into the container image.

An additional observation is that the build and deployment phases do not use the

same Python interpreter instance. Once the container image has been built, all Python processes terminate, and new processes are started when the container is deployed. This affects the behavior of Python's in-memory cache, which is discussed further in Section 6.7. In summary, the cache is cleared between the build and deployment phases, but not between deployment and ZAP tracing. As discussed in Section 6.7, this reduces the likelihood of false negatives for build and deployment tracing, whereas ZAP tracing is more susceptible to false negatives.

6.3.2 Static Results Comparison

Figure 5.11 compares the results obtained using Trivy, deployment tracing, and build tracing. The Venn diagrams show that tracing during the image build phase detects all packages identified by Trivy. For all applications except Bakery Demo, the Trivy results are also a subset of the deployment tracing results. The exception observed for Bakery Demo is possibly explained by packages that are present in the source code but never used at runtime, such as *elasticsearch* and *django_csp*.

Although build tracing identifies the same packages as Trivy, this behavior is not necessarily desirable if the detected packages are not actually executed. Since the purpose of file openings during the build phase cannot always be determined, build tracing introduces a risk of false positives, i.e., detection of packages that are not used at runtime. Therefore, SnakeBPF initiates data gathering after the container image has been built but before the application is deployed.

6.4 Web Application Evaluation

This section discusses the results from applying SnakeBPF on different web applications. To summarize the discussion of previous sections, SnakeBPF determine Python package usage by tracing the `openat` system call. In the post processing steps, only packages from the `site-packages` directory is kept and tracing is initiated before the application is deployed.

6.4.1 ZAP Evaluation

Table 5.1 depicts a comparison between packages detected by SnakeBPF and packages detected by Trivy when the web applications have been subjected to web scanning with ZAP. As stated in Section 4.2.1, web scanning is used to trigger as many packages in the applications as possible. Since most Python packages in the application are expected to be running at some point during the detection period, the packages detected by SnakeBPF should be similar to packages detected by Trivy.

As seen in Table 5.1, for all applications except Bakery Demo, SnakeBPF finds close to all packages detected by Trivy. For Plone, SnakeBPF does not find the package *zc_buildout*. However, the package *zc* is detected by SnakeBPF, but not by Trivy. One possible explanation is that they refer to the same package. For Docker-Django, the package that is found by Trivy but not by SnakeBPF is Pip. Since Pip is used

to install python packages, it is likely that this package is not actually used in the deployment phase. One possible explanation is that it is detected by Trivy because it is used during the container image build phase, which is an event that is not traced by SnakeBPF.

It is not certain that all packages detected by Trivy are used because of the limitations of static analysis. As seen in Table A.1, manual evaluation of the source code for Bakery Demo, confirms that at least 2 packages are disabled by default. They are installed in the container file system and is therefore detected by Trivy, but they are never used during runtime. Interestingly, the 31 packages that SnakeBPF does not find for the Bakery Demo application, are the same packages as in Table A.1 and *pip*. This may suggest that the 30 packages are not used at runtime at all.

For all applications, SnakeBPF detects between 1-4 packages that are not detected with static analysis. This includes noise such as *81d243bd2c585b0f4821_mypyc*. One possible explanation for this noise is that it is packages that are dependencies to packages detected with static analysis. Another possible explanation is that the noisy packages are filtered out by Trivy or that the packages is not detected by Trivy even though it is located inside the `site-packages` directory.

Overall, SnakeBPF finds almost all packages detected with the static baseline. The exception: Wagtail Bakery Demo, has been manually examined and a majority of the packages that SnakeBPF misses are either not mentioned in the source code or disabled by default as seen in Table A.1.

6.4.2 Targeted Evaluation

Table 5.2 presents the packages detected for the five web applications during manual, targeted functionality testing, as described in Section 4.2.2. The table shows that the number of detected packages for Wagtail Bakery Demo, Docker-Django, and Moments decreased by between 85% and 90%.

To function as an effective prioritization tool, SnakeBPF should detect fewer packages than static analysis, ideally only those that are actively running during the detection period. However, due to the significant cache impact discussed in Section 6.7, it is not possible to conclude that non-detected packages were not running. Nevertheless, it can be concluded that all packages detected during this targeted evaluation were running during the testing period.

For Todoism and Plone, the number of detected packages is very similar for both SnakeBPF and the static analysis results. One possible explanation behind this is that Todoism is a very simple web application. Any user behavior relies on database communication, as the tasks specified for each user are retained over sessions. This means that regardless of whether a user decides to add, check, or remove a specific todo task, the same backend functionality is triggered. This implies that the application utilizes the same packages for all use-case scenarios and that the results in

Table 5.2 for the Todoism application are valid.

SnakeBPF identified an additional package for the Docker–Django application, *django-debug-toolbar*, that was not detected by Trivy. To determine whether this finding was valid, the container was inspected. This confirmed the presence of the package inside the `site-packages` directory. Since the package's main functionality is to provide a debug panel[1], and this panel was in fact present during testing, our conclusion is that this is a valid package used by the application during runtime.

6.4.3 Comparing CVEs

Table 5.3 presents the number of CVEs detected by SnakeBPF under targeted evaluation in comparison to the CVEs detected by static analysis tools based on image analysis. For the three applications Docker-Django, Moments, and Plone, the CVEs detected by SnakeBPF are a subset of those detected by Trivy. This result is expected for Moments and Docker-Django, as the packages detected by SnakeBPF are also generally a subset of the packages detected by Trivy.

For Plone, the packages not included in the intersection are *zc* and *zc_buildout*. However, the difference in the number of detected CVEs is substantially larger than expected based solely on these missing packages. Upon further inspection, this discrepancy is primarily caused by vulnerable Python packages distributed in the egg format. Since this format is deprecated [74] and currently unsupported by SnakeBPF, these packages are excluded during processing. Support for egg packages could be added by extending the processing component and filtering mechanisms. Consequently, the comparison for Plone is less informative than for the other evaluated applications.

The results for Wagtail's Bakery Demo and Todoism differ from the previous applications, as SnakeBPF identifies CVEs that are not detected by Trivy. For Wagtail's Bakery Demo, the three uniquely detected CVEs are all associated with the *django* package. Similarly, for Todoism, the uniquely detected vulnerabilities are associated with the *selenium* and *werkzeug* packages. In both cases, the corresponding packages are also detected by Trivy. There is therefore no immediately clear explanation for why Trivy fails to identify these CVEs despite detecting the affected packages.

When considering the results in Table 5.2 alongside the CVE analysis, the relationship between detected packages and vulnerabilities differs between the evaluated applications. For Wagtail's Bakery Demo and Docker-Django, the number of detected packages is reduced by up to 85%, while the number of detected CVEs is only reduced by up to 73%. For these applications, this suggests that SnakeBPF may reduce noise by excluding non-vulnerable packages. However, this trend is not consistently observed across the remaining applications.

[1] https://pypi.org/project/django-debug-toolbar/

6.5 Multi-container support

As outlined in Section 4.4.5, supporting package detection across multiple simultaneously running applications requires that the detected packages remain consistent regardless of the container environment. Figure 5.13 presents the results of the detection mechanism when the application is executed in isolation, compared to a multi-container environment.

From the results, it is evident that for the Bakery Demo application, the exact same set of packages is detected in both environments. This consistency indicates that the processing component of SnakeBPF is effective in correctly attributing packages to their respective containers. However, when performing Python package detection on the Moments application in a multi-container environment, additional packages are detected compared to the single-container case. These four packages are listed in Table 5.4.

Closer examination of these packages suggests several possible explanations. First, the package *attr* could possibly refer to the package named *attrs* which is detected in both environments. Second, the package *pkg_resources* is included as part of *setuptools* [75]. Since *setuptools* is detected in both environments, the presence of *pkg_resources* is not unexpected. However, it remains unclear why it is not detected in the single-container case.

This discrepancy suggests that either the package detection method or the evaluation methodology may not be fully deterministic. One potential source of variation is the manual components of the testing procedure, such as the triggering of the web scanning tool *ZAP*. Timing differences between actions may therefore have influenced these results.

The remaining two packages listed in the table may also be related to packages detected in both environments. For example, *websocket* could be associated with either *trio_websocket* or *websocket_client*. Another possibility is that these packages originate from the Bakery Demo application but have been incorrectly attributed to the Moments container. This explanation is considered unlikely, as these packages are not known detected packages of the Bakery Demo.

Despite these discrepancies, the overall results show that the vast majority of detected packages are consistent across both environments. This indicates that the package detection mechanism performs reliably in multi-container settings and generally maps packages to their correct containers.

6.6 5G Core Kubernetes Cluster Evaluation

Before interpreting the results presented in Section 5.6, the challenges encountered when deploying SnakeBPF inside a 5G Kubernetes cluster will be introduced. Several of those challenges are related to the PID grouping methodology described in

Section 4.2.3. Recall that the post-processing component will assign the value of a `containerd-shim` process' PID as a container's root PID if the container is created during tracing. After inspecting the obtained `execve` traces, one such event is pod creation. Listing 6.1 illustrates the process of creating a new pod instance inside Kubernetes.

```
containerd-shim,999,1234, publish binary & start pod_cid
containerd-shim,5008,999, create namespace for pod_cid
runc,5020,5008,create systemd cgroup for pod_cid
```

Listing 6.1: Pseudocode for pod creation inside Kubernetes.

The post-processing component will thereby assign the PID 5008 to the new pod instance, which has been verified correspond to the same PID listed for the pod by the worker itself. This pid assignment is therefore assumed to be correct. However, when a container running inside the pod is started and later exits, this is where problem starts to emerge. The process is illustrated in Listing 6.2.

```
runc,5086,5008, create systemd cgroup for cid
runc,5176,5008, start cid

** time passes **

bash,5106,5008, execute init script

** time passes **

runc,5963,5008, kill all in systemd cgroup for cid
runc,5969,5008, delete cid in systemd cgroup
```

Listing 6.2: Pseudocode for container creation and termination inside a Kubernetes pod.

Like before, the container is assigned root PID 5008, which corresponds to the same PID as the pod. However, when inspecting the real PID assigned by the worker, it is in fact 5106, which corresponds to the PID of the bash process in Listing 6.2. To complicate things further, manual inspection of the obtained traces shows that this root PID assignment is not coherent across all container initializations. Sometimes a worker lists the bash process as root PID for a container, on other occasions a completely different type of process is responsible for container initialization.

In addition, a `containerd-shim` process creates multiple sibling processes, such as `runc` and `bash`, to handle container initialization and related events. Combined with the fact that an undefined amount of time pass from when a `runc` process is invoked to when the container is actually initialized, how do we know for certain which PID is the actual container root PID? To handle this case, the PID grouping algorithm must be developed further to handle sibling relationships between processes. Due to time constraints, this has not been achieved within the scope of the thesis and is left as future work (see Section 6.8.5).

As a result of this limitation, the current post-processing component yield the same

number of detected packages used by the terminated container and the pod. We know from manually inspecting the `openat` trace that this is not true, which is why the current methodology does not support detection of package usage per container instance. For this reason, the results presented in Section 5.6 for SnakeBPF includes detected packages per pod only.

As mentioned in Section 5.6, it is surprising to see that SnakeBPF detects more packages than Syft when tracing *custom interactions*. However, after normalizing package names and tracing their origins, most of these uniquely detected packages by SnakeBPF were reasonable findings (see Section 5.6). Furthermore, since the tracing was initialized before application deployment, it is expected that the test scenario exercised a majority of possible code flow paths. Additionally, since the Syft results corresponds to the sum of two SBOMs generated for two containers and not for the pod itself, the fact that SnakeBPF reports a larger package coverage in this experiment is in fact not so surprising.

When considering the results from tracing *default traffic* and *curl* requests, there is a steep decline in the number of detected packages by SnakeBPF. For the *curl* experiment, this might be explained by the in-memory Python cache (see Section 6.7). If the pod deployment traced prior to handling curl request triggered a majority of the available Python packages, this might result in fewer `openat` syscalls invoked later to handle the curl requests. However, tracing the same pod during *default traffic* detects few packages as well, which might indicate that the pod service does not use a lot of Python packages once it has been deployed. Further testing is required to establish an explanation with certainty.

In conclusion, when deploying SnakeBPF in a Kubernetes cluster, the data collection component worked as expected. eBPF was successfully deployed on worker instances and managed to trace kernel events. After post-processing the data, the number of detected packages by SnakeBPF varied significantly depending on the interactions made with the targeted pod. However, there are known issues with the PID grouping algorithm which needs to be resolved in order to provide more fine-grained insights of the package detection performance. If package usage can be reported per container instance, the results would then be more comparable with those obtained from Syft's SBOM generation.

6.7 In-memory Cache Impact

SnakeBPF relies on system call tracing to detect Python packages, and is therefore dependent on observable interactions with the Linux kernel. Consequently, any Python packages that do not trigger system calls fall outside the visibility of the approach. As described in Section 2.5.2, Python uses both an in-memory cache and an on-disk cache for module loading. Notably, the in-memory cache does not require file system interaction (e.g., via the `openat` system call).

Figure 5.16 shows the number of detected packages based on tracing the `openat`

system call. A clear spike is observed during the initial phase of program execution, after which the number of detections stops increasing. At the 9-hour mark, when a second round of web scanning is performed, only three packages are detected.

Given that the second web scan should in theory trigger the same package use as the first web scan, the absence of new detections strongly suggests that these packages are served from the Python in-memory cache. This demonstrates a limitation of SnakeBPF, namely that the method mainly detects packages that are used for the first time or retrieved from the on-disk cache. One exception to this statement is that packages are also detected when one of the sub modules are used for the first time, not necessarily only when the module itself is introduced.

There is no obvious explanation to the three detected packages. One possible explanation is that they contain submodules that were not triggered during the first web scan phase. By examining the raw data logs, this theory can be dismissed as the file paths detected in the second web scan phase are also found during the first web scan. Another alternative explanation is that those packages found during the second phase are cleared from the cache, possibly if the cache is full. However, as seen in Section 2.5.2, the cache is a Python dictionary and lives until the interpreter exits. A third possible explanation is that new Python processes are spawned when the application is interacted with. A new Python process would (if created with fork) have their own interpreter and thereby also individual in-memory cache dictionaries. The three packages could then be explained by new Python processes being created, and the packages detected are file opens when the in-memory cache is still empty. Likewise, there could be multiple Python workers handling requests to the server each with their own interpreter. The packages detected during the second scan might be packages that have been imported somewhere on the server before, but not on this specific worker.

To further investigate this behavior, Figure 5.17 presents results from tracing the `read` system call for the same experiment. Like `openat`, this trace also shows a small number of additional detections during the second web scanning phase. Besides from the reasoning above, it is possible that existing file descriptors opened earlier in the program are reused, allowing subsequent `read` operations without corresponding `openat` calls. However, neither this explanation fully accounts for why only a subset of packages exhibit this behavior, nor why such file descriptors would remain valid after an extended runtime (9 hours).

Overall, the results demonstrate that the data collection component is sensitive to Python's import caching mechanisms. This has direct implications for how detection results should be interpreted. When tracing begins at program startup, with an empty in-memory cache, the detected packages can be considered a close approximation of ground truth. Under these conditions, false negatives are expected to be minimal, as all required modules must be loaded at least once.

However, this assumption does not hold when tracing is initiated after the program

has already begun execution. In such cases, previously imported packages may reside entirely in memory and be reused without triggering observable system calls. This introduces a systematic source of false negatives, undermining the completeness of the detection method.

Mitigating this limitation would require either ensuring that tracing begins before any relevant imports occur, or explicitly clearing the in-memory cache before analysis. The latter remains outside the scope of this thesis. Alternatively, combining multiple system call traces (e.g., `openat` and `read`) may partially improve coverage, as they find different packages during the second web scan. However, the extent to which combining different data gathering approaches can reliably compensate for cache-related blind spots remains unclear.

6.8 Future Work

This section outlines potential future work based on the limitations identified throughout the thesis and discusses possible approaches for addressing them.

6.8.1 Performance and Scalability

Performance considerations were not a primary focus during the development of SnakeBPF. However, several aspects affecting performance can already be identified at this stage. As described in Section 2.4, the data-collecting eBPF programs operate partly in kernel space and partly in user space. Communication between these components relies on a shared buffer. Although SnakeBPF performed sufficiently during evaluation in a 5G core Kubernetes cluster (see Section 6.6), it remains unclear how the approach would scale to larger applications generating substantially more kernel-space events.

One possible improvement would be to reduce the number of events reaching the shared buffer by performing filtering directly in kernel space. For example, events associated with irrelevant file paths or process names could be discarded before being forwarded to user space.

6.8.2 Additional Data Gathering

Another potential improvement is to incorporate additional tracing points and data collection sources. A limitation of the current implementation is that multiple system calls may provide equivalent functionality. For example, processes may be spawned using `clone3` instead of `clone`. Since `clone3` is currently not traced, this may have affected the results in scenarios where no relevant system call traces were generated when new processes were created.

The evaluation results also indicate that combining multiple tracing sources may improve package detection coverage. For example, `read` system call tracing and `open` system call tracing identified partially different sets of packages in the cache

impact test (see Section 6.7). This suggests that incorporating additional tracing points could improve overall package detection.

If additional tracing points are leveraged, it may also be possible to explore behavioral profiling of libraries and packages. The current implementation primarily relies on directory names to identify packages. An alternative approach would be to construct behavioral profiles for packages and match observed runtime behavior against these profiles. Such profiles could, for example, consist of characteristic combinations of system calls associated with specific packages.

6.8.3 Version Gathering

This thesis did not identify a reliable method for determining package versions directly during runtime. Instead, package version information must be collected separately for each container image. The profiling approach described in the previous section could potentially also support runtime version identification if different package versions produce distinguishable behavioral profiles. Further investigation is required to determine whether such differences exist and/or are sufficiently consistent to enable reliable version detection.

6.8.4 Dynamic Baseline

This thesis contains some sources of uncertainty regarding the baseline evaluation. As discussed in Chapter 6, neither Python import-mechanism wrapping nor SnakeBPF successfully detect 30 of the packages included in Wagtail's Bakery Demo unless package detection is performed during the image build process. This may indicate that these packages are not actively used during runtime and that Python import-mechanism wrapping could therefore provide a reliable runtime baseline.

The baseline evaluation could therefore benefit from being repeated using additional web applications or controlled test programs. Such experiments could help determine whether the observed limitations are specific to Wagtail's Bakery Demo or more generally applicable. This was not performed within the scope of this thesis, since doing so could allow the results of SnakeBPF to influence the baseline design, which could introduce bias into the evaluation of SnakeBPF.

6.8.5 Improving the PID grouping algorithm

The current PID grouping algorithm does not fully support Kubernetes cluster environments, since it doesn't handle sibling relationships related to container deployment. As discussed in Section 6.6, the current methodology is coarse-grained when assigning a new container its root PID, resulting in package detection on a Kubernetes pod level, instead of per container. To resolve this issue, an approach that can determine the most likely container root PID given multiple options should be integrated with the current methodology to allow package detection per container. However, for some use cases, container-based results might not be needed. Such

cases include scenarios were pods are microservices representing a specific functionality, and package detection should be reported for that functionality.

6.8.6 Multi-language Support

This thesis focuses exclusively on Python package detection. However, SnakeBPF may also be applicable to other programming languages, since packages and libraries stored on disk typically need to be accessed before being loaded into memory during execution. One promising target is Golang, although the processing component would need to be adapted to account for Go module search paths.

It remains unclear to what extent the approach can be extended to statically linked libraries. In principle, SnakeBPF should be applicable to languages and runtime environments that rely on file system interactions when loading packages or libraries during execution. However, languages that embed dependencies directly into compiled binaries may require alternative detection strategies.

Furthermore, SnakeBPF would benefit from programming techniques such as lazy loading. When using lazy loading, resources like packages are loaded only when they are actually used by a program. This would improve the granularity of SnakeBPF, as file opens would likely be performed when a package is first used and not when it is first imported.

7

Conclusion

This thesis has presented a runtime Python package detection approach based on eBPF tracing of file-system access. The proposed approach SnakeBPF was evaluated across multiple web applications and deployment environments, including containerized environments and Kubernetes clusters. The results clearly demonstrate that SnakeBPF can successfully identify Python packages utilized at runtime. In several scenarios, SnakeBPF detected fewer packages compared with static analysis results. These results showcase that SnakeBPF is capable of reporting packages that were actually used by an application, and thus have potential to support more precise vulnerability prioritization for runtime software analysis. In several cases, the number of detected vulnerabilities represented a subset of those identified through static image analysis, indicating that SnakeBPF may help reduce noise by focusing on packages observed during execution.

At the same time, the results highlight several limitations. In particular, SnakeBPF is sensitive to the effects of Python's in-memory caching mechanisms, which can significantly influence package detection results. Although the detected packages generally represent true positives, the detection coverage is affected by runtime behavior and execution context, posing important challenges for reliable package detection. Another limitation is the adaptability to Kubernetes environments. In these environments, the package detection component performs well and the current PID grouping algorithm supports package per pod mapping. However, if one wish to support package usage per container as well, the post processing component will need some further enhancements.

Despite these limitations, the proposed approach SnakeBPF shows encouraging outcomes for containerized environments and Kubernetes-based deployments, indicating that eBPF-based runtime package detection can be applied in realistic deployment scenarios. Overall, the results suggest that runtime package analysis using eBPF is a promising direction for improving the contextual relevance of software vulnerability assessment, provided that the identified challenges are addressed.

8

Ethical Considerations

Several risks have been identified for this thesis project. First, any vulnerabilities discovered using the proposed runtime vulnerability detection method are not be publicly disclosed. The release of such information could compromise the security of the 5G packet core product line, potentially causing harm to both corporate stakeholders and customers. Consequently, no information regarding detected package names or versions will be disclosed in accordance with Ericsson's internal security policies.

The project is limited to detecting kernel-level runtime libraries. However, there is a risk that kernel-level monitoring could inadvertently expose information beyond the intended scope of data collection. To minimize any potential impact on customers or production systems, all experiments on the 5G core will be conducted within a sandboxed test environment.

Additional ethical considerations concern the limitations of the proposed detection method. As with any security tool, the method may fail to identify certain vulnerabilities. The creators of such tools have an ethical responsibility to clearly communicate these limitations and avoid overstating accuracy or coverage. Nevertheless, the absence of perfect detection does not negate the value of the approach. Given the inherent complexity of modern systems, the benefits of improved vulnerability detection outweigh the risks associated with false negatives, provided that the method is applied with appropriate caution and transparency.

Bibliography

[1] Ericsson, "Securing 5g networks in an evolving threat landscape," *Ericsson Mobility Report*, 2022, accessed: 2025-11-14. [Online]. Available: https://www.ericsson.com/49d3a0/assets/local/reports-papers/mobility-report/documents/2022/ericsson-mobility-report-june-2022.pdf"

[2] European Parliament and Council of the European Union, "Regulation (eu) 2024/2847 of the european parliament and of the council of 23 october 2024 on horizontal cybersecurity requirements for products with digital elements and amending regulations (eu) no 168/2013 and (eu) 2019/1020 and directive (eu) 2020/1828 (cyber resilience act)," Official Journal of the European Union, L series.

[3] S. Tariq, M. B. Chhetri, S. Nepal, and C. Paris, "Alert fatigue in security operations centres: Research challenges and opportunities," *ACM Computing Surveys*, vol. 57, no. 9, pp. 224:1–224:38, 2025.

[4] National Cyber Security Centre (NCSC), "Huawei cyber security evaluation centre oversight board, annual report 2019: A report to the national security adviser of the united kingdom," London, United Kingdom, Tech. Rep., 2019, accessed: 2025-05-11. [Online]. Available: https://nsarchive.gwu.edu/document/18367-national-security-archive-huawei-cyber-security

[5] Department for Digital, Culture, Media Sport, National Cyber Security Centre and The Rt Hon Oliver Dowden CBE MP, "Huawei to be removed from UK 5G networks by 2027," https://www.gov.uk/government/news/huawei-to-be-removed-from-uk-5g-networks-by-2027, 2020, accessed: 2026-05-11.

[6] J. Neubert, "Vulnerable and outdated components: An owasp top 10 threat," accessed: 2026-02-04. [Online]. Available: https://www.invicti.com/blog/web-security/vulnerable-and-outdated-components-owasp-top-10

[7] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer*

and Communications Security, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2187–2200. [Online]. Available: https://doi.org/10.1145/3133956.3134059

[8] J. Huang, N. Borges, S. Bugiel, and M. Backes, "Up-to-crash: Evaluating third-party library updatability on android," in *2019 IEEE European Symposium on Security and Privacy (EuroSP)*, 2019, pp. 15–30.

[9] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 356–367. [Online]. Available: https://doi.org/10.1145/2976749.2978333

[10] O. Jarkas, R. Ko, N. Dong, and R. Mahmud, "A container security survey: Exploits, attacks, and defenses," *ACM Comput. Surv.*, vol. 57, no. 7, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3715001

[11] F. Minna and F. Massacci, "Sok: Run-time security for cloud microservices. are we there yet?" *Computers Security*, vol. 127, p. 103119, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823000299

[12] OWASP, "Vulnerability scanning tools," https://owasp.org/www-community/Vulnerability_Scanning_Tools, 2024, accessed: 2026-01-21.

[13] OWASP, "Source code analysis tools," https://owasp.org/www-community/Source_Code_Analysis_Tools, 2022, accessed: 2026-01-21.

[14] Sysdig Inc., "Threat detection built on falco," accessed: 2026-02-05. [Online]. Available: https://www.sysdig.com/opensource/falco

[15] ——, "Welcome to the sysdig wiki!" accessed: 2026-02-05. [Online]. Available: https://github.com/draios/sysdig/wiki

[16] E. Carter, "Sysdig and snyk use runtime intelligence to eliminate vulnerability noise," Blog post, Sysdig, february 2022, accessed: 2026-01-20. [Online]. Available: https://www.sysdig.com/blog/sysdig-snyk-partnership

[17] Sysdig Inc., "Secure saas release notes 2025 – august 29, 2025 - software composition analysis (sca) integrations," https://docs.sysdig.com/en/release-notes/saas-sysdig-secure-release-notes/2025-archive/, 2025, accessed: 2026-01-20.

[18] Snyk, "Open source risk management made for developers," accessed: 2026-02-05. [Online]. Available: https://snyk.io/product/open-source-security-management/

[19] K. Scarfone and M. P. Souppaya, "Technical guide to information security testing and assessment," National Institute of Standards and Technology, Gaithersburg, MD, Special Publication (NIST SP) 800-115 800-115, 2008, accessed: 2026-01-22. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf

[20] D. Stefanović and S. Nikolić, "Static code analysis tools: A systematic literature review," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020, available online via cloudfront link. [Online]. Available: https://d1wqtxts1xzle7.cloudfront.net/81794567/078-libre.pdf?1646565380=&response-content-disposition=inline%3B+filename%3DStatic_Code_Analysis_Tools_A_Systematic.pdf

[21] OWASP Foundation, "Owasp devsecops guideline – vulnerability scanning (v-0.2)," OWASP Foundation website, 2023, accessed: 2026-01-22. [Online]. Available: https://owasp.org/www-project-devsecops-guideline/latest/02-Vulnerability-Scanning

[22] ——, "Static application security testing (sast)," OWASP DevSecOps Guideline website, 2022, accessed: 2026-01-22. [Online]. Available: https://owasp.org/www-project-devsecops-guideline/latest/02a-Static-Application-Security-Testing

[23] S. Springett, "Component analysis," OWASP Foundation website, 2025, accessed: 2025-11-26. [Online]. Available: https://owasp.org/www-community/Component_Analysis

[24] T. Ball, "The concept of dynamic analysis," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. Berlin, Heidelberg: Springer-Verlag, 1999, p. 216–234.

[25] Aqua Security, "Trivy documentation: Language coverage," https://trivy.dev/docs/latest/guide/coverage/language/, 2025, accessed: 2026-04-08.

[26] ——, "Trivy python coverage documentation," https://trivy.dev/docs/latest/guide/coverage/language/python/, 2025, accessed: 2026-04-07.

[27] Anchore Inc., "Scan Target-Specific Behaviors - Container Image Scan Targets," https://oss.anchore.com/docs/guides/sbom/scan-targets/#container-image-scan-targets, 2026, accessed: 2026-05-13.

[28] ——, "Getting Started - Find packages within a container image," https://oss.anchore.com/docs/guides/sbom/getting-started/, 2026, accessed: 2026-05-13.

[29] OWASP Foundation, "ZAP (Zed Attack Proxy): Web Application Security Scanner (README)," https://github.com/zaproxy/zaproxy/blob/main/

README.md, 2024, gitHub repository documentation, accessed 2026-04-08.

[30] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for sql injection and xss attacks," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 2007, pp. 365–372.

[31] CVE™ Program, "Cve program mission," accessed: 2026-01-22. [Online]. Available: https://www.cve.org/

[32] IBM Security Randori, "Cpe, cve, and cvss information," IBM Security Randori Documentation Website, 2024, accessed: 2026-01-22. [Online]. Available: https://www.ibm.com/docs/en/randori?topic=properties-cpe-cve-cvss-information

[33] National Vulnerability Database (NIST), "Official common platform enumeration (cpe) dictionary," august 2025, accessed: 2026-01-22. [Online]. Available: https://nvd.nist.gov/products/cpe

[34] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunstrom, "The ebpf runtime in the linux kernel," 09 2024.

[35] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005.

[36] D. A. Rusling, "The Linux Document Project: Processes," https://tldp.org/LDP/tlk/kernel/processes.html, 1999, accessed: 2026-04-20.

[37] IBM Corporation, "Processes," https://www.ibm.com/docs/en/aix/7.1.0?topic=processes-, 2023, accessed: 2026-04-28.

[38] M. Kerrisk, "Namespaces in operation, part 1: namespaces overview," https://lwn.net/Articles/531114/, 2013, accessed: 2026-04-28.

[39] Linux Kernel Organization, "The /proc Filesystem Documentation," https://www.kernel.org/doc/html/latest/filesystems/proc.html, 2009, accessed: 2026-04-08.

[40] M. Kerrisk, "Linux manual page - syscalls(2)," accessed: 2026-03-18. [Online]. Available: https://www.man7.org/linux/man-pages/man2/syscalls.2.html

[41] The kernel development community, "The linux kernel - system calls," accessed: 2026-03-18. [Online]. Available: https://linux-kernel-labs.github.io/refs/heads/master/lectures/syscalls.html

[42] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped I/O for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.

USENIX Association, Jul. 2020, pp. 813–827. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/papagiannis

[43] The Linux Vault, "The power of linux kernel hooks," accessed: 2026-03-18. [Online]. Available: https://www.thelinuxvault.net/linux-kernel-basics/the-power-of-linux-kernel-hooks/#types-of-kernel-hooks

[44] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," vol. 51, no. 2, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3158644

[45] M. Desnoyers, "Using the linux kernel tracepoints," accessed: 2026-03-18. [Online]. Available: https://docs.kernel.org/trace/tracepoints.html

[46] H. Megahed, "Tracepoints," accessed: 2026-03-18. [Online]. Available: https://ebpf.hamza-megahed.com/docs/chapter3/3-tracepoints/

[47] K. Jim, S. P. Prasanna, and H. Masami, "Kernel Probes (Kprobes)," https://www.kernel.org/doc/Documentation/kprobes.txt, 2019, accessed 2026-04-08.

[48] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn, "bpftime: userspace ebpf runtime for uprobe, syscall and kernel-user interactions," 2023. [Online]. Available: https://arxiv.org/abs/2311.07923

[49] Redhat, "What is a linux container?" https://www.redhat.com/en/topics/containers/whats-a-linux-container, 2026, accessed: 2026-04-22.

[50] I. Lewis, "Container runtimes part 1: An introduction to container runtimes," https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r, 2017, accessed: 2026-04-27.

[51] The Open Container Initiative (OCI), "About the open container initiative," https://opencontainers.org/about/overview/, accessed: 2026-04-22.

[52] I. Velichko, "Journey from containerization to orchestration and beyond," https://iximiuz.com/en/posts/journey-from-containerization-to-orchestration-and-beyond/#container-runtimes,%20https://www.ibm.com/think/topics/container-management#2014952961, 2022, accessed: 2026-04-27.

[53] C. Brauner, "Runtimes and the curse of the privileged container," https://brauner.io/2019/02/12/privileged-containers.html, 2019, accessed: 2026-04-27.

[54] The Kubernetes Authors, "Overview," https://kubernetes.io/docs/concepts/overview/, 2026, accessed: 2026-05-08.

[55] ——, “Cluster architecture,” https://kubernetes.io/docs/concepts/architecture/, 2026, accessed: 2026-05-08.

[56] ——, “Pods,” https://kubernetes.io/docs/concepts/workloads/pods/, 2026, accessed: 2026-05-08.

[57] IBM Cloud Education, “App stability and availability in a cloud environment,” https://www.ibm.com/think/insights/kubernetes-benefits, accessed: 2026-05-11.

[58] eBPF community, “ebpf documentation - what is ebpf?” accessed: 2026-03-18. [Online]. Available: https://ebpf.io/what-is-ebpf/#what-is-ebpf

[59] G. Fournier, S. Afchain, and S. Baubeau, “Runtime security monitoring with ebpf,” 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:247625852

[60] KungFuDev, “The beginning of my ebpf journey - kprobe adventures with bcc,” accessed: 2026-01-23. [Online]. Available: https://www.kungfudev.com/blog/2023/10/14/the-beginning-of-my-ebpf-journey-kprobe-bcc

[61] Python Software Foundation, “The python language reference - the import system,” accessed: 2026-03-18. [Online]. Available: https://docs.python.org/3/reference/import.html#packages

[62] ——, “The python language reference - glossary,” accessed: 2026-03-18. [Online]. Available: https://docs.python.org/3/glossary.html#term-namespace-package

[63] ——, “The python language reference - the import statement,” accessed: 2026-04-07. [Online]. Available: https://docs.python.org/3/reference/simple_stmts.html#the-import-statement

[64] J. van Rossum and P. Moore, “New import hooks,” Python Software Foundation, PEP 302, 2002. [Online]. Available: https://peps.python.org/pep-0302/

[65] E. Snow, “A modulespec type for the import system,” Python Software Foundation, PEP 451, 2013. [Online]. Available: https://peps.python.org/pep-0451/

[66] Python Software Foundation, “The python language reference - the initialization of the sys.path module search path,” accessed: 2026-04-07. [Online]. Available: https://docs.python.org/3/library/sys_path_init.html#sys-path-init

[67] Python Core Team, “site.py in CPython 3.14,” https://github.com/python/

cpython/blob/3.14/Lib/site.py, 2026, version 3.14, accessed 2026-04-07.

[68] Python Software Foundation, "Python/C API Reference Manual," https://docs.python.org/3/c-api/index.html, 2026, python 3.14, accessed 2026-04-07.

[69] D. Beazley, B. Ward, and I. Cooke, "The inside story on shared libraries and dynamic loading," *Computing in Science Engineering*, vol. 3, no. 5, pp. 90–97, 2001.

[70] GNU Project, *dlopen(3)*, linux man-pages 6.17, accessed 2026-04-09.

[71] Free Software Foundation, *The GNU C Library Reference Manual*, version 2.43, accessed 2026-04-10. [Online]. Available: https://sourceware.org/glibc/manual/latest/html_mono/libc.html

[72] OSV, "A distributed vulnerability database for open source," accessed: 2026-03-18. [Online]. Available: https://osv.dev/#use-the-api

[73] Plone Foundation, "Plone: Enterprise level cms," 2026, accessed: 2026-05-07. [Online]. Available: https://plone.org/

[74] Python Packaging Authority. (2026) Package formats. Python Packaging User Guide. [Online]. Available: https://packaging.python.org/en/latest/discussions/package-formats/

[75] P. J. Eby, "PEP 365 – Adding the pkg_resources module," https://peps.python.org/pep-0365/, 2007, python Enhancement Proposal, Rejected. Accessed: 2026-05-11.

A

Appendix 1: Baseline Selection

This chapter presents Table A.1, showcasing packages detected by Trivy that the other baseline alternatives (see Section 4.3) did not manage to detect.

Table A.1: The 30 Python packages uniquely identified by Trivy static analysis, but remained undetected by the other two baseline candidates. These other two candidates were overriding the ___import___ statement and using a custom meta path finder.

Library	**Found in Source Code**	**Conditional**	**Comment**
aws_requests_auth	✓	✓	
boto3	?	✗	Commonly used with aws_requests_auth.
botocore	✗	✗	Belongs to boto3.
cffi	✗	✗	
cryptography	✗	✗	
django_csp	✓	✓	Not enabled by default.
django_redis	✗	✗	Only used during container build.
django_storages	?	✗	Referenced as string only, but no import of package.
elasticsearch	✓	✓	Not enabled by default.
et_xmlfile	✗	✗	
google_api_core	✗	✗	
google_auth	✗	✗	
google_cloud_core	✗	✗	
google_cloud_storage	✗	✗	
google_crc32c	✗	✗	
google_resumable_media	✗	✗	
googleapis_common_protos	✗	✗	
jmespath	✗	✗	
openpyxl	✗	✗	

Library	**Found in Source Code**	**Conditional**	**Comment**
pip	✗	✗	
proto_plus	✗	✗	
protobuf	✗	✗	
pyasn1	✗	✗	
pyasn1_modules	✗	✗	Dependency of pyasn1.
pycparser	✗	✗	
python_dateutil	✗	✗	
redis	?	✗	Likely used as DB/-cache/message broker.
s3transfer	✗	✗	Part of boto3.
six	✗	✗	
whitenoise	✗	✗	

B

Appendix 2: Data Gathering Approaches

This chapter contains detailed information about traces used to present the figures available in Section 5.2.

B.1 Kernel-space Tracing

Table B.1: Package detection overlap between `mmap`, `openat` and `read` for the Wagtail container subjected to ZAP web scanning. After considering only `site-packages`, the results are presented as a Venn diagram in Figure 5.6b.

Package	Detected By
`google`	`openat` only
`anyascii`	`openat` & `read`
`asgiref`	`openat` & `read`
`beautifulsoup4`	`openat` & `read`
`certifi`	`openat` & `read`
`defusedxml`	`openat` & `read`
`dj_database_url`	`openat` & `read`
`django`	`openat` & `read`
`django_debug_toolbar`	`openat` & `read`
`django_extensions`	`openat` & `read`
`django_filter`	`openat` & `read`
`django_modelcluster`	`openat` & `read`
`django_permissionedforms`	`openat` & `read`
`django_stubs_ext`	`openat` & `read`
`django_taggit`	`openat` & `read`
`django_tasks`	`openat` & `read`
`django_treebeard`	`openat` & `read`
`djangorestframework`	`openat` & `read`
`draftjs_exporter`	`openat` & `read`
`filetype`	`openat` & `read`

Package	Detected By
google_cloud_storage	openat & read
idna	openat & read
laces	openat & read
modelsearch	openat & read
psycopg	openat & read
python_dotenv	openat & read
requests	openat & read
soupsieve	openat & read
sqlparse	openat & read
telepath	openat & read
typing_extensions	openat & read
urllib3	openat & read
wagtail	openat & read
wagtail_font_awesome_svg	openat & read
willow	openat & read
81d243bd2c585b0f4821_mypyc	mmap, openat & read
charset_normalizer	mmap, openat & read
pillow	mmap, openat & read
pillow_heif	mmap, openat & read
psycopg_binary	mmap, openat & read

Table B.2: Package detection overlap between mmap, openat, and Trivy for the Wagtail container during targeted functionality evaluation, presented as a Venn diagram in Figure 5.9.

Package	Detected By
81d243bd2c585b0f4821_mypyc	mmap only
anyascii	Trivy only
asgiref	Trivy only
aws_requests_auth	Trivy only
beautifulsoup4	Trivy only
boto3	Trivy only
botocore	Trivy only
certifi	Trivy only
cffi	Trivy only
cryptography	Trivy only
defusedxml	Trivy only
dj_database_url	Trivy only
django_csp	Trivy only
django_modelcluster	Trivy only
django_permissionedforms	Trivy only
django_redis	Trivy only
django_storages	Trivy only

Package	Detected By
django_stubs_ext	Trivy only
django_taggit	Trivy only
django_tasks	Trivy only
django_treebeard	Trivy only
elasticsearch	Trivy only
et_xmlfile	Trivy only
filetype	Trivy only
google_api_core	Trivy only
google_auth	Trivy only
google_cloud_core	Trivy only
google_cloud_storage	Trivy only
google_crc32c	Trivy only
google_resumable_media	Trivy only
googleapis_common_protos	Trivy only
idna	Trivy only
jmespath	Trivy only
laces	Trivy only
openpyxl	Trivy only
pip	Trivy only
proto_plus	Trivy only
protobuf	Trivy only
psycopg	Trivy only
pyasn1	Trivy only
pyasn1_modules	Trivy only
pycparser	Trivy only
python_dateutil	Trivy only
python_dotenv	Trivy only
redis	Trivy only
requests	Trivy only
s3transfer	Trivy only
six	Trivy only
soupsieve	Trivy only
sqlparse	Trivy only
telepath	Trivy only
typing_extensions	Trivy only
urllib3	Trivy only
uwsgi	Trivy only
wagtail_font_awesome_svg	Trivy only
whitenoise	Trivy only
willow	Trivy only
charset_normalizer	mmap & Trivy
pillow_heif	mmap & Trivy
django	openat & Trivy
django_debug_toolbar	openat & Trivy

Package	**Detected By**
`django_extensions`	`openat` & Trivy
`django_filter`	`openat` & Trivy
`djangorestframework`	`openat` & Trivy
`draftjs_exporter`	`openat` & Trivy
`modelsearch`	`openat` & Trivy
`wagtail`	`openat` & Trivy
`pillow`	`mmap`, `openat` & Trivy
`psycopg_binary`	`mmap`, `openat` & Trivy

C

Appendix 3: Tracing Strategies

This chapter contains detailed information about detected packages depicted in figures presented in Section 5.3.

C.1 Wagtail

Table C.1: Detected packages presented in Figure 5.10a, Figure 5.11a and Figure 5.12a.

Package	Detected By
distutils	Build
django_basic_auth_ip_whitelist	Build
setuptools	Build
uwsgidecorators	Build
aws_requests_auth	Build & Trivy
boto3	Build & Trivy
botocore	Build & Trivy
cffi	Build & Trivy
cryptography	Build & Trivy
django_csp	Build & Trivy
django_redis	Build & Trivy
django_storages	Build & Trivy
elasticsearch	Build & Trivy
et_xmlfile	Build & Trivy
google_api_core	Build & Trivy
google_auth	Build & Trivy
google_cloud_core	Build & Trivy
google_crc32c	Build & Trivy
google_resumable_media	Build & Trivy
googleapis_common_protos	Build & Trivy
jmespath	Build & Trivy
openpyxl	Build & Trivy
pip	Build & Trivy

proto_plus	Build & Trivy
protobuf	Build & Trivy
pyasn1	Build & Trivy
pyasn1_modules	Build & Trivy
pycparser	Build & Trivy
python_dateutil	Build & Trivy
redis	Build & Trivy
s3transfer	Build & Trivy
six	Build & Trivy
uwsgi	Build & Trivy
whitenoise	Build & Trivy
81d243bd2c585b0f4821_mypyc	Build & Run
google	Build & Run
anyascii	Build & Run & Trivy
asgiref	Build & Run & Trivy
beautifulsoup4	Build & Run & Trivy
certifi	Build & Run & Trivy
charset_normalizer	Build & Run & Trivy
defusedxml	Build & Run & Trivy
dj_database_url	Build & Run & Trivy
django_modelcluster	Build & Run & Trivy
django_permissionedforms	Build & Run & Trivy
django_stubs_ext	Build & Run & Trivy
django_tasks	Build & Run & Trivy
django_treebeard	Build & Run & Trivy
filetype	Build & Run & Trivy
google_cloud_storage	Build & Run & Trivy
idna	Build & Run & Trivy
laces	Build & Run & Trivy
pillow_heif	Build & Run & Trivy
python_dotenv	Build & Run & Trivy
requests	Build & Run & Trivy
soupsieve	Build & Run & Trivy
sqlparse	Build & Run & Trivy
telepath	Build & Run & Trivy
typing_extensions	Build & Run & Trivy
urllib3	Build & Run & Trivy
willow	Build & Run & Trivy
django	Build & Run & ZAP & Trivy
django_debug_toolbar	Build & Run & ZAP & Trivy
django_extensions	Build & Run & ZAP & Trivy
django_filter	Build & Run & ZAP & Trivy
django_taggit	Build & Run & ZAP & Trivy
djangorestframework	Build & Run & ZAP & Trivy

draftjs_exporter	Build & Run & ZAP & Trivy
modelsearch	Build & Run & ZAP & Trivy
pillow	Build & Run & ZAP & Trivy
psycopg	Build & Run & ZAP & Trivy
psycopg_binary	Build & Run & ZAP & Trivy
wagtail	Build & Run & ZAP & Trivy
wagtail_font_awesome_svg	Build & Run & ZAP & Trivy

C.2 Docker-Django

Table C.2: Detected packages presented in Figure 5.10b, Figure 5.11b and Figure 5.12b.

Package	Detected By
pip	Build & Trivy
click_didyoumean	Build & Run
distutils	Build & Run
prompt_toolkit	Build & Run
python_dateutil	Build & Run
amqp	Build & Run & Trivy
asgiref	Build & Run & Trivy
billiard	Build & Run & Trivy
celery	Build & Run & Trivy
click	Build & Run & Trivy
click_plugins	Build & Run & Trivy
click_repl	Build & Run & Trivy
gunicorn	Build & Run & Trivy
kombu	Build & Run & Trivy
packaging	Build & Run & Trivy
psycopg	Build & Run & Trivy
ruff	Build & Run & Trivy
setuptools	Build & Run & Trivy
six	Build & Run & Trivy
sqlparse	Build & Run & Trivy
tzdata	Build & Run & Trivy
tzlocal	Build & Run & Trivy
vine	Build & Run & Trivy
wcwidth	Build & Run & Trivy
whitenoise	Build & Run & Trivy
django_debug_toolbar	Build & Run & ZAP
django	Build & Run & ZAP & Trivy
redis	Build & Run & ZAP & Trivy

C.3 Todoism

Table C.3: Detected packages presented in Figure 5.10c, Figure 5.11c and Figure 5.12c.

Package	Detected By
distutils	Build & Run & ZAP
babel	Build & Run & ZAP & Trivy
certifi	Build & Run & ZAP & Trivy
chardet	Build & Run & ZAP & Trivy
click	Build & Run & ZAP & Trivy
faker	Build & Run & ZAP & Trivy
flask	Build & Run & ZAP & Trivy
flask_babel	Build & Run & ZAP & Trivy
flask_cors	Build & Run & ZAP & Trivy
flask_login	Build & Run & ZAP & Trivy
flask_sqlalchemy	Build & Run & ZAP & Trivy
flask_wtf	Build & Run & ZAP & Trivy
httpie	Build & Run & ZAP & Trivy
idna	Build & Run & ZAP & Trivy
itsdangerous	Build & Run & ZAP & Trivy
jinja2	Build & Run & ZAP & Trivy
markupsafe	Build & Run & ZAP & Trivy
pathtools	Build & Run & ZAP & Trivy
pip	Build & Run & ZAP & Trivy
pygments	Build & Run & ZAP & Trivy
python_dateutil	Build & Run & ZAP & Trivy
python_dotenv	Build & Run & ZAP & Trivy
pytz	Build & Run & ZAP & Trivy
requests	Build & Run & ZAP & Trivy
selenium	Build & Run & ZAP & Trivy
setuptools	Build & Run & ZAP & Trivy
six	Build & Run & ZAP & Trivy
sqlalchemy	Build & Run & ZAP & Trivy
text_unidecode	Build & Run & ZAP & Trivy
urllib3	Build & Run & ZAP & Trivy
watchdog	Build & Run & ZAP & Trivy
werkzeug	Build & Run & ZAP & Trivy
wheel	Build & Run & ZAP & Trivy
wtforms	Build & Run & ZAP & Trivy
pkg_resources	Run & ZAP

C.4 Moments

Table C.4: Detected packages presented in Figure 5.10d, Figure 5.11d and Figure 5.12d.

Package	Detected By
attr	Build
py	Build
websocket	Build
yaml	Build
distutils	Build & Run
attrs	Build & Run & Trivy
blinker	Build & Run & Trivy
certifi	Build & Run & Trivy
cfgv	Build & Run & Trivy
click	Build & Run & Trivy
distlib	Build & Run & Trivy
dnspython	Build & Run & Trivy
faker	Build & Run & Trivy
filelock	Build & Run & Trivy
flask_avatars	Build & Run & Trivy
flask_dropzone	Build & Run & Trivy
flask_login	Build & Run & Trivy
flask_mail	Build & Run & Trivy
flask_sqlalchemy	Build & Run & Trivy
flask_whooshee	Build & Run & Trivy
flask_wtf	Build & Run & Trivy
greenlet	Build & Run & Trivy
h11	Build & Run & Trivy
identify	Build & Run & Trivy
iniconfig	Build & Run & Trivy
itsdangerous	Build & Run & Trivy
jinja2	Build & Run & Trivy
markupsafe	Build & Run & Trivy
nodeenv	Build & Run & Trivy
outcome	Build & Run & Trivy
packaging	Build & Run & Trivy
pillow	Build & Run & Trivy
pip	Build & Run & Trivy
platformdirs	Build & Run & Trivy
pluggy	Build & Run & Trivy
pre_commit	Build & Run & Trivy

pyjwt	Build & Run & Trivy
pysocks	Build & Run & Trivy
pytest	Build & Run & Trivy
python_dateutil	Build & Run & Trivy
python_dotenv	Build & Run & Trivy
pyyaml	Build & Run & Trivy
ruff	Build & Run & Trivy
selenium	Build & Run & Trivy
setuptools	Build & Run & Trivy
six	Build & Run & Trivy
sniffio	Build & Run & Trivy
sortedcontainers	Build & Run & Trivy
sqlalchemy	Build & Run & Trivy
trio	Build & Run & Trivy
trio_websocket	Build & Run & Trivy
typing_extensions	Build & Run & Trivy
urllib3	Build & Run & Trivy
virtualenv	Build & Run & Trivy
watchdog	Build & Run & Trivy
websocket_client	Build & Run & Trivy
wheel	Build & Run & Trivy
wsproto	Build & Run & Trivy
wtforms	Build & Run & Trivy
bootstrap_flask	Build & Run & ZAP & Trivy
email_validator	Build & Run & ZAP & Trivy
flask	Build & Run & ZAP & Trivy
idna	Build & Run & ZAP & Trivy
werkzeug	Build & Run & ZAP & Trivy
whoosh	Build & Run & ZAP & Trivy

C.5 Plone

Table C.5: Detected packages presented in Figure 5.12e.

Package	Detected By
distutils	Run
pkg_resources	Run
zc	Run & ZAP
pip	Run & ZAP & Trivy
setuptools	Run & ZAP & Trivy
wheel	Run & ZAP & Trivy
zc_buildout	Trivy

D

Appendix 4: Web Application Evaluation

This chapter contains detailed information about detected packages depicted in figures available in Section 5.4.

D.1 Targeted Evaluation

This section contains all detected packages presented in Table 5.2 in Section 5.4.2.

D.1.1 Wagtail

Package	Detected By
anyascii	Trivy
asgiref	Trivy
aws_requests_auth	Trivy
beautifulsoup4	Trivy
boto3	Trivy
botocore	Trivy
certifi	Trivy
cffi	Trivy
charset_normalizer	Trivy
cryptography	Trivy
defusedxml	Trivy
dj_database_url	Trivy
django_csp	Trivy
django_modelcluster	Trivy
django_permissionedforms	Trivy
django_redis	Trivy
django_storages	Trivy
django_stubs_ext	Trivy
django_taggit	Trivy
django_tasks	Trivy

Package	Detected By
django_treebeard	Trivy
elasticsearch	Trivy
et_xmlfile	Trivy
filetype	Trivy
google_api_core	Trivy
google_auth	Trivy
google_cloud_core	Trivy
google_cloud_storage	Trivy
google_crc32c	Trivy
google_resumable_media	Trivy
googleapis_common_protos	Trivy
idna	Trivy
jmespath	Trivy
laces	Trivy
openpyxl	Trivy
pillow_heif	Trivy
pip	Trivy
proto_plus	Trivy
protobuf	Trivy
psycopg	Trivy
pyasn1	Trivy
pyasn1_modules	Trivy
pycparser	Trivy
python_dateutil	Trivy
python_dotenv	Trivy
redis	Trivy
requests	Trivy
s3transfer	Trivy
six	Trivy
soupsieve	Trivy
sqlparse	Trivy
telepath	Trivy
typing_extensions	Trivy
urllib3	Trivy
uwsgi	Trivy
wagtail_font_awesome_svg	Trivy
whitenoise	Trivy
willow	Trivy
django	SnakeBPF & Trivy
django_debug_toolbar	SnakeBPF & Trivy
django_extensions	SnakeBPF & Trivy
django_filter	SnakeBPF & Trivy
djangorestframework	SnakeBPF & Trivy
draftjs_exporter	SnakeBPF & Trivy

Package	Detected By
modelsearch	SnakeBPF & Trivy
pillow	SnakeBPF & Trivy
psycopg_binary	SnakeBPF & Trivy
wagtail	SnakeBPF & Trivy

D.1.2 Docker-Django

Package	Detected By
django_debug_toolbar	SnakeBPF
amqp	Trivy
asgiref	Trivy
billiard	Trivy
celery	Trivy
click	Trivy
click_plugins	Trivy
click_repl	Trivy
gunicorn	Trivy
kombu	Trivy
packaging	Trivy
pip	Trivy
psycopg	Trivy
ruff	Trivy
setuptools	Trivy
six	Trivy
sqlparse	Trivy
tzdata	Trivy
tzlocal	Trivy
vine	Trivy
wcwidth	Trivy
whitenoise	Trivy
django	SnakeBPF & Trivy
redis	SnakeBPF & Trivy

D.1.3 Todoism

Package	Detected By
distutils	SnakeBPF
pkg_resources	SnakeBPF
babel	SnakeBPF & Trivy
certifi	SnakeBPF & Trivy
chardet	SnakeBPF & Trivy
click	SnakeBPF & Trivy

Package	Detected By
faker	SnakeBPF & Trivy
flask	SnakeBPF & Trivy
flask_babel	SnakeBPF & Trivy
flask_cors	SnakeBPF & Trivy
flask_login	SnakeBPF & Trivy
flask_sqlalchemy	SnakeBPF & Trivy
flask_wtf	SnakeBPF & Trivy
httpie	SnakeBPF & Trivy
idna	SnakeBPF & Trivy
itsdangerous	SnakeBPF & Trivy
jinja2	SnakeBPF & Trivy
markupsafe	SnakeBPF & Trivy
pathtools	SnakeBPF & Trivy
pip	SnakeBPF & Trivy
pygments	SnakeBPF & Trivy
python_dateutil	SnakeBPF & Trivy
python_dotenv	SnakeBPF & Trivy
pytz	SnakeBPF & Trivy
requests	SnakeBPF & Trivy
selenium	SnakeBPF & Trivy
setuptools	SnakeBPF & Trivy
six	SnakeBPF & Trivy
sqlalchemy	SnakeBPF & Trivy
text_unidecode	SnakeBPF & Trivy
urllib3	SnakeBPF & Trivy
watchdog	SnakeBPF & Trivy
werkzeug	SnakeBPF & Trivy
wheel	SnakeBPF & Trivy
wtforms	SnakeBPF & Trivy

D.1.4 Moments

Package	Detected By
attrs	Trivy
blinker	Trivy
certifi	Trivy
cfgv	Trivy
click	Trivy
distlib	Trivy
dnspython	Trivy
faker	Trivy
filelock	Trivy
flask_avatars	Trivy

Package	Detected By
flask_dropzone	Trivy
flask_login	Trivy
flask_mail	Trivy
flask_sqlalchemy	Trivy
flask_whooshee	Trivy
flask_wtf	Trivy
greenlet	Trivy
h11	Trivy
identify	Trivy
iniconfig	Trivy
itsdangerous	Trivy
jinja2	Trivy
markupsafe	Trivy
nodeenv	Trivy
outcome	Trivy
packaging	Trivy
pip	Trivy
platformdirs	Trivy
pluggy	Trivy
pre_commit	Trivy
pyjwt	Trivy
pysocks	Trivy
pytest	Trivy
python_dateutil	Trivy
python_dotenv	Trivy
pyyaml	Trivy
ruff	Trivy
selenium	Trivy
setuptools	Trivy
six	Trivy
sniffio	Trivy
sortedcontainers	Trivy
sqlalchemy	Trivy
trio	Trivy
trio_websocket	Trivy
typing_extensions	Trivy
urllib3	Trivy
virtualenv	Trivy
watchdog	Trivy
websocket_client	Trivy
werkzeug	Trivy
wheel	Trivy
wsproto	Trivy
wtforms	Trivy

Package	Detected By
bootstrap_flask	SnakeBPF & Trivy
email_validator	SnakeBPF & Trivy
flask	SnakeBPF & Trivy
idna	SnakeBPF & Trivy
pillow	SnakeBPF & Trivy
whoosh	SnakeBPF & Trivy

D.1.5 Plone

Package	Detected By
zc_buildout	Trivy
zc	SnakeBPF
pip	SnakeBPF & Trivy
setuptools	SnakeBPF & Trivy
wheel	SnakeBPF & Trivy

D.2 Vulnerability detection

This section contains all vulnerabilities presented in Table 5.3 in Section 5.4.3.

D.2.1 Wagtail

CVEs	Detected By
CVE-2026-35192	SnakeBPF
CVE-2026-5766	SnakeBPF
CVE-2026-6907	SnakeBPF
CVE-2025-8869	Trivy
CVE-2026-1703	Trivy
CVE-2026-28684	Trivy
CVE-2026-3219	Trivy
CVE-2026-6357	Trivy
CVE-2026-44197	SnakeBPF & Trivy
CVE-2026-44198	SnakeBPF & Trivy
CVE-2026-44199	SnakeBPF & Trivy
CVE-2026-44200	SnakeBPF & Trivy
CVE-2026-44201	SnakeBPF & Trivy

D.2.2 Docker-Django

CVEs	Detected By
CVE-2026-1703	Trivy

CVE-2026-3219	Trivy
CVE-2026-33033	Trivy
CVE-2026-33034	Trivy
CVE-2026-3902	Trivy
CVE-2026-4277	Trivy
CVE-2026-4292	Trivy
CVE-2026-6357	Trivy
CVE-2026-35192	SnakeBPF & Trivy
CVE-2026-5766	SnakeBPF & Trivy
CVE-2026-6907	SnakeBPF & Trivy

D.2.3 Todoism

CVEs	Detected By
CVE-2022-28108	SnakeBPF
CVE-2022-29361	SnakeBPF
CVE-2023-5590	SnakeBPF
CVE-2026-23949	Trivy
CVE-2020-25032	SnakeBPF & Trivy
CVE-2020-26137	SnakeBPF & Trivy
CVE-2020-28493	SnakeBPF & Trivy
CVE-2021-20270	SnakeBPF & Trivy
CVE-2021-27291	SnakeBPF & Trivy
CVE-2021-33503	SnakeBPF & Trivy
CVE-2021-42771	SnakeBPF & Trivy
CVE-2022-0430	SnakeBPF & Trivy
CVE-2022-23491	SnakeBPF & Trivy
CVE-2022-24737	SnakeBPF & Trivy
CVE-2022-40896	SnakeBPF & Trivy
CVE-2023-23934	SnakeBPF & Trivy
CVE-2023-25577	SnakeBPF & Trivy
CVE-2023-30861	SnakeBPF & Trivy
CVE-2023-32681	SnakeBPF & Trivy
CVE-2023-37920	SnakeBPF & Trivy
CVE-2023-43804	SnakeBPF & Trivy
CVE-2023-45803	SnakeBPF & Trivy
CVE-2023-46136	SnakeBPF & Trivy
CVE-2023-48052	SnakeBPF & Trivy
CVE-2024-1681	SnakeBPF & Trivy
CVE-2024-22195	SnakeBPF & Trivy
CVE-2024-34064	SnakeBPF & Trivy
CVE-2024-34069	SnakeBPF & Trivy
CVE-2024-35195	SnakeBPF & Trivy
CVE-2024-3651	SnakeBPF & Trivy

CVE-2024-37891	SnakeBPF & Trivy
CVE-2024-47081	SnakeBPF & Trivy
CVE-2024-49766	SnakeBPF & Trivy
CVE-2024-49767	SnakeBPF & Trivy
CVE-2024-56326	SnakeBPF & Trivy
CVE-2024-6221	SnakeBPF & Trivy
CVE-2024-6839	SnakeBPF & Trivy
CVE-2024-6844	SnakeBPF & Trivy
CVE-2024-6866	SnakeBPF & Trivy
CVE-2025-27516	SnakeBPF & Trivy
CVE-2025-50181	SnakeBPF & Trivy
CVE-2025-66221	SnakeBPF & Trivy
CVE-2025-66418	SnakeBPF & Trivy
CVE-2025-66471	SnakeBPF & Trivy
CVE-2025-8869	SnakeBPF & Trivy
CVE-2026-1703	SnakeBPF & Trivy
CVE-2026-21441	SnakeBPF & Trivy
CVE-2026-21860	SnakeBPF & Trivy
CVE-2026-24049	SnakeBPF & Trivy
CVE-2026-25645	SnakeBPF & Trivy
CVE-2026-27199	SnakeBPF & Trivy
CVE-2026-27205	SnakeBPF & Trivy
CVE-2026-28684	SnakeBPF & Trivy
CVE-2026-3219	SnakeBPF & Trivy
CVE-2026-4539	SnakeBPF & Trivy
CVE-2026-6357	SnakeBPF & Trivy

D.2.4 Moments

CVEs	Detected By
CVE-2025-27516	Trivy
CVE-2025-43859	Trivy
CVE-2025-50181	Trivy
CVE-2025-50182	Trivy
CVE-2025-66221	Trivy
CVE-2025-66418	Trivy
CVE-2025-66471	Trivy
CVE-2025-68146	Trivy
CVE-2025-71176	Trivy
CVE-2025-8869	Trivy
CVE-2026-1703	Trivy
CVE-2026-21441	Trivy
CVE-2026-21860	Trivy
CVE-2026-22701	Trivy

CVE-2026-22702	Trivy
CVE-2026-23949	Trivy
CVE-2026-24049	Trivy
CVE-2026-25990	Trivy
CVE-2026-27199	Trivy
CVE-2026-28684	Trivy
CVE-2026-3219	Trivy
CVE-2026-32597	Trivy
CVE-2026-40192	Trivy
CVE-2026-42308	Trivy
CVE-2026-42310	Trivy
CVE-2026-42311	Trivy
CVE-2026-6357	Trivy
CVE-2025-47278	SnakeBPF & Trivy
CVE-2026-27205	SnakeBPF & Trivy

D.2.5 Plone

CVEs	Detected By
CVE-2020-10177	Trivy
CVE-2020-10378	Trivy
CVE-2020-10379	Trivy
CVE-2020-10994	Trivy
CVE-2020-11538	Trivy
CVE-2020-35653	Trivy
CVE-2020-35654	Trivy
CVE-2020-35655	Trivy
CVE-2021-23437	Trivy
CVE-2021-25287	Trivy
CVE-2021-25288	Trivy
CVE-2021-25289	Trivy
CVE-2021-25290	Trivy
CVE-2021-25291	Trivy
CVE-2021-25292	Trivy
CVE-2021-25293	Trivy
CVE-2021-27921	Trivy
CVE-2021-27922	Trivy
CVE-2021-27923	Trivy
CVE-2021-28675	Trivy
CVE-2021-28676	Trivy
CVE-2021-28677	Trivy
CVE-2021-28678	Trivy
CVE-2021-34552	Trivy
CVE-2022-22815	Trivy

CVE-2022-22816	Trivy
CVE-2022-22817	Trivy
CVE-2022-2309	Trivy
CVE-2022-24303	Trivy
CVE-2022-29217	Trivy
CVE-2022-40899	Trivy
CVE-2022-45198	Trivy
CVE-2023-32681	Trivy
CVE-2023-37920	Trivy
CVE-2023-43804	Trivy
CVE-2023-44271	Trivy
CVE-2023-44389	Trivy
CVE-2023-45803	Trivy
CVE-2023-4863	Trivy
CVE-2023-50447	Trivy
CVE-2024-0669	Trivy
CVE-2024-22195	Trivy
CVE-2024-22889	Trivy
CVE-2024-28219	Trivy
CVE-2024-34064	Trivy
CVE-2024-35195	Trivy
CVE-2024-3651	Trivy
CVE-2024-37891	Trivy
CVE-2024-39689	Trivy
CVE-2024-42353	Trivy
CVE-2024-47081	Trivy
CVE-2024-47532	Trivy
CVE-2024-49768	Trivy
CVE-2024-49769	Trivy
CVE-2024-51734	Trivy
CVE-2024-53899	Trivy
CVE-2024-56201	Trivy
CVE-2024-56326	Trivy
CVE-2025-27516	Trivy
CVE-2025-50181	Trivy
CVE-2025-61911	Trivy
CVE-2025-61912	Trivy
CVE-2025-66418	Trivy
CVE-2025-66471	Trivy
CVE-2025-68146	Trivy
CVE-2025-69534	Trivy
CVE-2026-21441	Trivy
CVE-2026-22701	Trivy
CVE-2026-22702	Trivy

CVE-2026-25645	Trivy
CVE-2026-28356	Trivy
CVE-2026-28413	Trivy
CVE-2026-28684	Trivy
CVE-2026-30922	Trivy
CVE-2026-32597	Trivy
CVE-2026-41066	Trivy
CVE-2026-42308	Trivy
CVE-2026-42310	Trivy
CVE-2023-5752	SnakeBPF & Trivy
CVE-2024-6345	SnakeBPF & Trivy
CVE-2025-47273	SnakeBPF & Trivy
CVE-2025-8869	SnakeBPF & Trivy
CVE-2026-1703	SnakeBPF & Trivy
CVE-2026-3219	SnakeBPF & Trivy
CVE-2026-6357	SnakeBPF & Trivy

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY