Sigma Rule based eBPF Logger for Containerized Environments

Master Thesis

Examiner: Prof. Dr. Andreas Grebe

Second Examiner: Thomas Karl M.Sc.

Henrik Wittemeier

Matr-no. 11157323

Technische Hochschule Köln

Faculty of Information, Media and Electrical Engineering
Institue of Computer and Communication Technology

February 3, 2025

Abstract

The introduction of new technologies for mobile networks such as open RAN leads to challenges in the area of security observability. The distribution and protection of containerised components across different nodes in large Kubernetes clusters requires new technologies to maintain an overview of security-relevant events. In order to achieve visibility within containers, this project implements an eBPF-based solution that is deployed distributed across a Kubernetes cluster.
Rules that characterize malicious behaviour play a major role in the development of observability technologies. A good set of rules ensures that neither too many unimportant events are written to a database, but also that all critical processes are logged. To minimize the effort of maintaining this rule set, the community-maintained Sigma rule set is used in this project. This ruleset will be extended from its purpose for SIEM solutions to be used for configuration of the eBPF based tool Tracee. To use the Sigma rules, the pySigma tool is configured to translate the rules from the Sigma format into the Tracee signatures that are implemented in Go.

Contents

List of Figures

List of Tables

List of Code

Acronyms

3GPP	Third Generation Parntership Program
5G	Fifth Generation Mobile Network
API	Application Programming Interface
BBU	Base Band Unit
BPF	Berkeley Packet Filter
CNI	Container Network Interface
CU	Central Unit
CVE	Common Vulnerabilities and Exposures
DFIR	Digital Forensics and Incident Response
DU	Distributed Unit
eBPF	extended Berkeley Packet Filter
JSON	Java Script Object Notation
LLM	Large Language Model
Near-RT RIC	Near-Realtime Ran Intelligent Controller
NFs	Network Functions
Non-RT RIC	Non-Realtime Ran Intelligent Controller
OPA	Open Policy Agent
OS	Operating System
RAN	Radio Access Network
RBAC	Role Based Access Control
RIC	Ran Intelligent Controller
RU	Radio Unit
SIEM	Security Information and Event Management
SMO	Software Management and Orchestration
SSH	Secure Shell
UE	User Equipment

Introduction

Mobile networks are a crucial part of the modern telecommunication infrastructure. The mobile technology evolved greatly from the first generation of mobile networks to the current standard Fifth Generation Mobile Network (5G). 5G delivers high capacity low latency network connections for many devices simultaneously. The fast evolution of the mobile networks lead to an oligopolistic market that is paced and dominated by a few large mobile equipment manufacturers. Currently most of the mobile networks are based on the Third Generation Parntership Program (3GPP) standard. The 3GPP standard provides interoperability between mobile networks and User Equipment (UE) and some interfaces within a mobile network [1]. A mobile network based on this specification is mostly a black box distributed by a single vendor. The infrastructure is therefore built on vendor specific hardware. To enter the market as a mobile network manufacturer, it is necessary to build a complete mobile network solution and produce the corresponding hardware within the short release cycles.

To solve these problems the O-RAN ALLIANCE initiated the Open RAN project. The O-RAN ALLIANCE is an association of vendors, operators and researchers of mobile networks [2]. The Open RAN project aims for dividing the prior black boxed Radio Access Network (RAN) into smaller components with specified interfaces. This Open RAN removes the vendor lock and promotes competition in the RAN ecosystem. Part of the effort is also the shift of workloads from vendor specific hardware to cloud environments based on off-the-shelf hardware. The division into smaller components provides smaller vendors with an easier market entry, resulting in increasing competition. Competitive markets are likely to have a better cost/performance ratio.

1.1 Problem Statement

A new approach of deploying a RAN with multiple new interfaces introduces new challenges regarding security and observability. The threat surface, related to the prior 3GPP architecture, is increased due to more interfaces, multi- vendor environments and the deployment in clouds. With the introduction of the Open RAN, significantly more stakeholders are involved in the deployment. As the threat surface also includes malicious actors or actors that become malicious during the life cycle of a deployment, a perfectly secure Open RAN cannot be built. Fixing vulnerabilities and designing an application with the security-by-design/default pattern is a way to decrease the threat surface. A typical approach for securing software systems is regularly examining them for known vulnerabilities from sources like the Common Vulnerabilities and Exposures (CVE) database [3]. This database lists vulnerabilities that were found in software components and applications. By fixing vulnerabilities from the CVE database, it is possible to reduce the risk of security incidents. However it cannot be completely avoided. This means that successful attacks and intrusions into software systems must always be a concern.
In the situation of a possible intrusion by an attacker, the goal is to reconstruct the way the attacker got access. This can be achieved by adding observability tools to the Open RAN deployment. The observability data needs to include file changes, network related events and processes that are executed. Subsequently to an attack this data could be forensically examined to get information about the entry point of intrusion, the affected component and which further steps where taken by the attacker.
Large systems with high workloads would generate a lot of data. Collecting all process, network and file data from systems and preserving it for further investigations gets quite expensive. The solution is to only collect data which could possibly help to determine malicious behavior. A monitoring solution would be configured by rules that match all forensic relevant events excluding the normal system operation.
This set of rules must always be kept up to date to new attack patterns malware and vulnerabilities.

1.2 Objectives

The effort of maintaining an up to date rule set is very time consuming and susceptible to information deficiency. To have a monitoring solution with a rule database that is updated without any efforts, this project aims for using community maintained rule repositories and

adapting them to a monitoring solution that can be used in an Open RAN deployment.
The deployment of the mobile network in a Kubernetes cloud environment brings along new challenges in terms of observability. To meet the associated requirements, an appropriate monitoring solution is found and integrated into the Open RAN.

1.3 Structure of Work

The development of the monitoring solution for Open RAN deployments involves several steps. At first, the existing Open RAN deployment and related work is analyzed to get an overview about technical and functional requirements. Based on the defined requirements, an existing tool that will be adapted is chosen. The tool is then put into the overall context and a design is worked out which includes the tools involved for developing and deploying the application. The developed design is implemented and integrated into the Open RAN deployment and functional tests are performed to prove the technical functionality.
The application is evaluated in terms of its usefulness, performance and effectiveness. Therefore, tests with performance measurements, typical attacks and comparisons to similar tools are performed. Subsequently, the results are evaluated and discussed.

1.4 Background

This thesis was initiated in the 5G-FORAN project [4, 5]. The project was launched in 2023 by the PROCYDE GmbH together with the Cologne University of Applied Sciences. 5G-Foran aims for increasing the cyber security in containerized environments especially for Open RAN by introducing IT forensics. The project is split into the parts 5G-FORAN-DFIR and 5G-FORAN-ATTACK. Rapid identification, analysis and defense methods against security incidents are developed with the help of attack simulations to provide practical solutions. The idea of the project is to develop attack scenarios to generate attack traces by the 5G-FORAN-ATTACK project, which can be analyzed for forensic artefacts in the 5G-FORAN-DFIR project.

Technical and Research Context

Since this work concentrates on the observability of O-RAN deployments in Kubernetes, related work is reviewed in terms of interesting aspects of security challenges and observability goals in containerized RAN environments. To determine the relevance of observability in O-RAN deployments, researches concerning the threat surface of the RAN software and infrastructure are analyzed.
Finally existing solutions and other investigations are reviewed to check the applicability of sigma rules on the extended Berkeley Packet Filter (eBPF) in this thesis.

2.1 Security in Open RAN

The initiation of the Open RAN project by the O-RAN ALLIANCE is intended to break down the RAN into smaller components with specified interfaces. The given specifications were implemented by the O-RAN Software Community to create a proof of concept. This thesis will focus on an environment that is created on the base of the effort of the O-RAN Software Community. The benefits of an Open RAN are diverse. Through the evolution of a monolithic system to a multi vendor approach, network operators can choose the manufacturer with the best price/performance ratio. Small vendors can concentrate on single components to get a faster market entry. Expensive dedicated 3GPP hardware is replaced by an architecture based on virtualization and cloudification. The prior Base Band Unit (BBU) is disaggregated into Radio Unit (RU), Distributed Unit (DU), Central Unit (CU) and the Ran Intelligent Controller (RIC) [6].

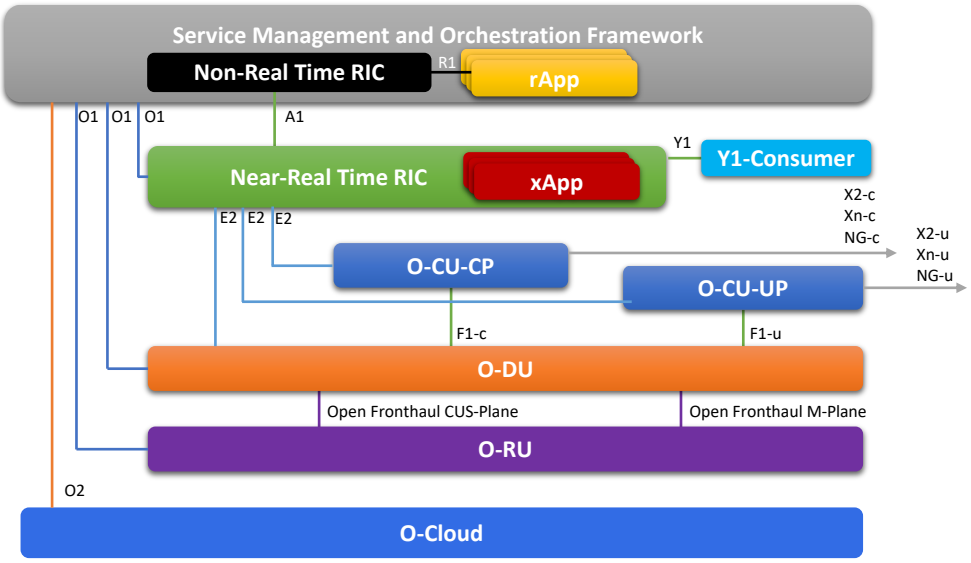


Figure 1: Architecture according to the O-RAN Alliance

The central part of the O-RAN project are the RICs. In the work of the O-RAN Software Community project, these components are deployed in Kubernetes clusters grouped by their time criticality. The deployment is controlled by the Software Management and Orchestration (SMO) Framework to ensure automatic scalability and central configurability. The Near-Realtime Ran Intelligent Controller (Near-RT RIC) handles control loops with a cycle of 10ms-1s and the Non-Realtime Ran Intelligent Controller (Non-RT RIC) handles control loops with a cycle of >1s. The RICs gain their intelligence by the installation of Apps. The Near-RT RIC xApps are deployed to control RAN decisions like handover management. In the Non-RT RIC control functions, like cell shutdowns are managed. The approach of extending the functionality and intelligence of the RAN through xApps and rApps involves containerized Network Functions (NFs) delivered by different Vendors. In future releases, the integration of these Apps should be integrated using Role Based Access Control (RBAC), however, in current releases it is not implemented. All advantages of the Open RAN also lead to new security challenges. Every additional interface or technology, but also the higher number of involved stakeholders in Open RAN, enlarges the threat surface compared to the prior 3GPP architecture. Attackers that were identified in prior researches were Outsiders, Users, Cloud Operators, Insiders and RAN Operators [7].
Following central threat surfaces are considered by the O-RAN ALLIANCE [8]:

- Additional functions: SMO, Non-Real-Time RIC, Near-Real-Time RIC

- Additional open interfaces: A1, E2, O1, O2, Open Fronthaul
- Modified architecture: Lower Layer Split (LLS) 7-2x
- Decoupling increases threat to Trust Chain
- Containerization and Virtualization: Disaggregation of software and hardware
- Exposure to public exploits may be increased due to use of Open Source Code

The rather large threat surface and high number of possible attackers implies logging of security related events. The O-RAN ALLIANCE states in their specifications that "relevant activities events SHOULD be logged and logs collected SHOULD be analyzed in real time" [8]. Beyond that, no specific requirements for logging are mentioned.
The observability over the given threats can be achieved through different solutions. In general, logs for an event should be generated at the location where most available data about the event is available. For example, the usage of a wrong password should be logged by the application where the authentication is performed. In general, application related events should be logged by the application. Applications know about their endpoints and are able log the behavior that is connected to interacting with an application. Attacks that do not interact with the application or exceed the intended usage of the application must be considered on a lower level. As Open RAN is deployed in a Kubernetes Cluster, it is an interesting point to generate logs. Possibilities and challenges for observability in Kubernetes are reviewed in the next chapter.

2.2 Security Challenges in Containerized Environments

The introduction of containerization in the RAN infrastructure aims at enhancing the security and scalability of the infrastructure.

2.2.1 Containerization

Containerization is a technology that introduces an additional isolation between the host Operating System (OS) and the application. In opposite to traditional virtualization technologies, containerization only adds a small overhead to workloads. Instead of running a virtual computer with its own OS and apps on top of the host OS, containers share the kernel with the host OS (see Figure 2).

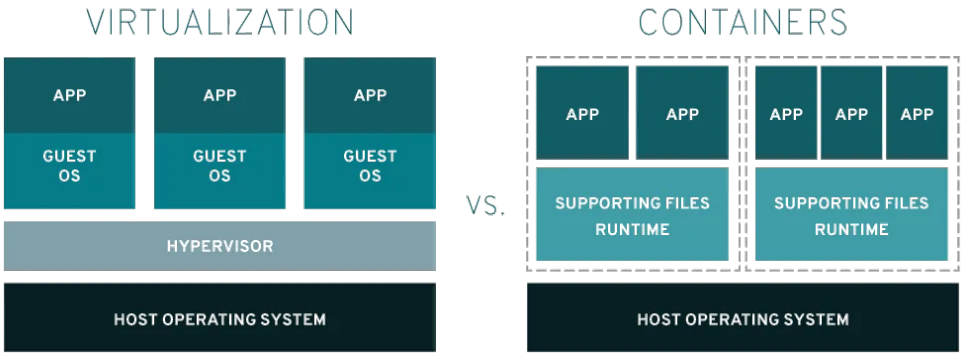


Figure 2: Virtualization vs. Containerization [9]

Containers are built from container images. These images include all necessary libraries and dependencies that an application needs to run. Container images are built in layers. To add a file to a container, a new layer is added to an existing container image. Containers can be created from scratch or using prebuilt images like "ubuntu" which include typical libraries for applications built for Ubuntu. The layer approach makes similar image parts exchangeable between container images decreasing resource usage.

The container image acts like a virtual file system. This virtual file system is available during the lifetime of a container but is not persisted. Each container is started with the original image. Applications inside the container cannot access the host file system. To ensure persistence for specific files or locations in the container, a volume must be mounted. The scheduling of the processes of the container is handled by the host OS kernel. To achieve a sufficient isolation from the host, container processes are scheduled in their own kernel namespace.

The management of the container life cycle is done by a container orchestrator such as Docker. Docker adds an user interface to a container runtime such as containerd. With Docker, additional functions such as exposing a container's ports and mounting a host file system into the container can be achieved with simple command line syntax. [10, 11].

2.2.2 Kubernetes

For larger containerized applications, a command line control for containers is not sufficient. Kubernetes is a container orchestrator with features to scale workloads. It is used to deploy, manage, and operate larger applications that involve multiple dependent containerized services across a cluster of nodes. Kubernetes achieves its scalability by distributing workloads across multiple worker nodes.
The deployment of Applications takes place over an exposed Kubernetes Application Programming Interface (API) with a description in YAML files. The Kubernetes scheduler then compares the desired cluster state with the current cluster state and adds missing resources or increases the number of replications if performance indicators require that. A typical Kubernetes cluster includes a few control-plane nodes and many worker nodes. [12]. While the introduction of containerization increases the security, flexibility and scalability, it also increases the effort of observing the infrastructure.

2.2.3 Observability in Kubernetes

The usual approach of observing activities on a server is to deploy a tool like *Auditd* [13]. *Auditd* interacts with the OS and captures security relevant events and stores them in log file. This has the disadvantage that an attacker that already has gained access to a system could easily manipulate the logs or even change the logging behavior of the tool that is used [14]. In an containerized environment, the most interesting activities happen isolated from the host OS as the Applications run in containers. As Containers have their own filing system, the direct access to their logs is also not possible. The isolation requires a different approach of activity monitoring. The dynamic distribution of containers above nodes has the effect that containers have short life cycles. Possible forensic artifacts would not withstand a redeployment of a container.
A better approach is to observe activities that happen inside a container from the outside. Attackers are unable to manipulate the logging behavior and unable to identify the used logging mechanism. Without their knowledge about the monitoring systems, the effectiveness in detecting suspicious behavior would be higher. This also has the advantage that containers can be kept lightweight as not every container needs the logging framework to be installed.
Kubernetes adds another challenge to monitoring due to its distributed nature. In addition to containers, nodes can also leave the cluster,losing potential artifacts in the process. The implementation of a central logging database is necessary to reduce the effort of data collection and ensure persistence of historic events.

2.3 eBPF based Observability

The kernel of a Kubernetes node is shared between the host OS and the containers. Although isolation prevents observing container activity within the OS, kernel-level events can be traced.
A tool for configuring a logging behavior is eBPF. eBPF is an extension of the Berkeley Packet Filter (BPF) which was historically used for packet filtering and evolved to a generalized in-kernel virtual machine. eBPF can be used to run programs inside the kernel to extend its functionality. As containers and host OS share the same kernel, it is a promising tool for activity monitoring.
The Kernel of an OS is the interface to the hardware of a computer. Every hardware interaction like network traffic or disk writings are executed by kernel functions. To write a file to a disk, several kernel functions and modules are involved. The Linux kernel offers hook points between these functions. With the use of eBPF, it is possible to attach to this hook points for the purpose of reading and manipulating data that is passed to the next function. In this example, it is not intended to manipulate data but to log suspicious activity. The data that is received from the hook point can be processed by the eBPF program. The possibilities of processing in the kernel space are quite limited. eBPF Code is written in a C-like language that is very restricted in its functionality. To prevent heavy resource allocation, following restrictions are applied on eBPF Programs [15]:

- Only prebound loops and no recursion
- 512 Byte stack size
- Maximum of 1 Mio Instructions
- No floating point arithmetic
- No direct file access
- No global Variables
- No out of bounds memory access

These restrictions are enforced in the eBPF compiler and verifier. Programs that fail to be verified are not loaded into the kernel. These strict limitations ensure that the functionality of the kernel is not impacted. The usage of eBPF helper functions can increase these limits but yet is not trivial to implement.

From the kernel-space, it is not possible to collect the events and save them to a file, therefore the data must be transmitted to user-space. This is achieved by using BPF Maps [16], BPF maps provide storage that is shared between kernel and userspace. From the user space, this data could then be retrieved and be further processed or persisted.
With the example of a hook point between functions for disk access, every writing action would trigger the eBPF program that is attached to the kernel hook. To prevent a flooding of unnecessary data, a filter mechanism must be implemented that either filters for interesting events in kernel or user space. As the complexity of eBPF programs is very limited, advanced filterings may not be possible. The possibility of kernel space filtering must be further evaluated.

2.4 Sigma Project

The selection of the rule set format was made in favor of Sigma, because the format comes with the largest open Source rule set. Due to the high acceptance in the community, over 500 contributors and around 20 commits per week assume that this project is actively maintained and is future-proof.
Rego [17] is a format for policies that is standardized and used by many applications. However, available open source rule sets are not applicable to different applications due to their different naming conventions of fields.
The Sigma Ecosystem comprises the Sigma Format, Sigma Tools and Sigma Rule Collections [18]. The purpose of the Sigma project is to have a publicly available Sigma rule set based on an generic detection format. The general scope of the rules is to detect suspicious behavior in previously collected application, system and auditlogs. Sigma rules have a format that is not locked to a product or vendor, but can be converted in to queries for various Security Information and Event Management (SIEM) tools. The Sigma rule repository has around 3500 rules which are grouped into different categories based on whether they belong to general attackers behavior or to specific malware or vulnerabilities [19]. Characteristic for the Sigma detection format is an easy to read generic intuitive structure.

```
1 title: Decode Base64 Encoded Text
2 id: e2072cab-8c9a-459b-b63c-40ae79e27031
3 status: test
4 description: Detects usage of base64 utility to decode arbitrary base64-
    encoded text
5 references: [...]
6 author: Daniil Yugoslavskiy, oscd.community
```

```
date: 2020-10-19
modified: 2021-11-27
tags:
    - attack.defense-evasion
    - attack.t1027
logsource:
    category: process_creation
    product: linux
detection:
    selection:
        Image|endswith: '/base64'
        CommandLine|contains: '-d' # Also covers "--decode"
    condition: selection
falsepositives:
    - Legitimate activities
level: low
```

Code 1: Example Sigma Rule

This example rule shows the general structure of a Sigma rule. Each Sigma rule contains at least three types of fields. The Metadata fields characterize the purpose of the rule, store scientific references and map to tactics of the Mitre ATT&CK matrix, a table where many tactics of attackers in different stages of an attack are listed [20]. Moreover the Sigma rule has a mapping part where a log source is mapped to the rule. Sigma rules are in general not locked to specific products, but possible log sources can be restricted through properties that must be matched. These log sources can be abstract fields like "network_connection" or "file_event" but can also be specific services like *Apache* or *Auditd*. As events can differ between operating systems or are not expected to happen, log sources can also be operating system specific.

The detection part of a Sigma rule characterizes the fields and values that should be included in a log to trigger the rule. In this example, the detection is quite simple and is triggered if a "*/base64 *-d*" is executed. For a string value, it is characterized if its an exact match or if the value is expected at the begin/end of a value. Different field value combinations are then linked with a condition. For more complex log rules, the conditions and fields can be linked in a more sophisticated way.

The field "falsepositives" states why an event could be triggered even if there is no malicious activity on the target host.

The core concept of Sigma rules is to query SIEM solutions for records of suspicious behavior.

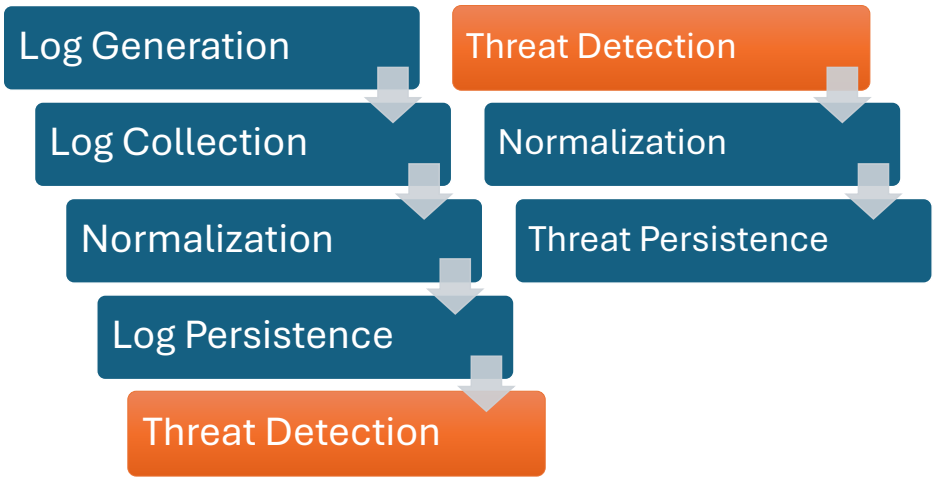


Figure 3: Sigma usage in SIEM (left) vs SIGMA usage in this project (right)

The connection of eBPF with the Sigma detection format would extend the intended usage of Sigma rules. A typical SIEM solution has several functions (see Figure 3). Logs generated by applications or the host OS are collected and normalized. These normalized logs are then persisted. The storing of logs is important because SIEM solutions are used after an incident to reconstruct the attack. After an attack, the Sigma ruleset would be used to detect possible threats based on the persisted logs.

In this thesis, the Sigma rule set is used to directly detect threats without the need of any application or OS logs. Instead, the eBPF is configured with the Sigma rules to only output the classified threats. These threats are then normalized and persisted for further usage. This approach has some advantages and disadvantages. Advantages are lower resource usage as only threats are processed and persisted, and all necessary data to classify a threat is generated. A disadvantage is that possible threats that are not already known in the Sigma rule set cannot be identified afterwards because no additional data is available. In the classic SIEM approach, a lot more data is available, therefore, it is more likely that additional threats can be found afterwards.

Sigma rules are not written with the focus of matching kernel events. Many of the publicly available Sigma rules focus on application-specific logs, such as web server access logs from an Apache web server. The specific format that these logs have would not be observable in the kernel space. Nevertheless, generic rules can also be found in the Sigma rule repository which concentrate on network traffic, scheduled processes or file modifications.

The benefits of utilizing the Sigma detection format for event filtering could be the higher acceptance of an established log format but also the active community that develops rules that react to new threats. As the Sigma detection format has an easy abstract format, it

would also be easy to extend the rules to the specific requirements of the Open RAN deployment.

2.5 Existing Solutions

To the best of my knowledge, there is only one existing solution that combines sigma rules with the eBPF [21]. Other projects that were found during this research all come with their own specific rule format. The project "huakiwi" uses the "eventstream" log source and gains the logs from ebpf. This project seems relatively small as it currently supports only 20 rules with the same logsource. As this project is very focused on the eventstream log source, it looks quite hard to extend to generalized translation of sigma rules. The concept of using eBPF for the observability of security relevant events is not new. There are many larger active maintained projects as Falco or Tetragon which use their own rulesets [22]. Falco uses the linux kernel layer with the help of eBPF to monitor anomalous activity. These generated events are enriched with information such as container runtime metrics. Falco has a ruleset with 160 rules. The rules are written as lists, macros or rules depending on their complexity. Lists can include for example different filename macros and rules can contain more complex behaviors.
Tetragon is similar to Falco, a monitoring tool that uses the eBPF that additionally can also enforce access controls to kernel functions to prevent malicious behavior. Tetragon only comes with a small set of policies that must be extended for sufficient logging.

Concept

The development of a security monitoring solution can have a foreseeable extent. However, the effort in keeping these monitoring solutions up to date by updating the rulesets nearly instantly to new vulnerabilities and attacks is time consuming. This is why the work of this thesis will be the connection of the existing and actively maintained sigma ruleset to an existing observability tool that then can be embedded in the Open RAN deployment.

3.1 Scope

The gained knowledge over the big threat surface of an Open RAN Deployment in section 2.1 would assume a monitoring tool that detects all different possible intrusion points. However, this is not sensible as these threat vectors are very different in nature. The threat vector in this thesis will concentrate on the security impact of "Containerization and Virtualization: Disaggregation of Software and Hardware" [8]. As this threat vector cannot be captured by application level logs, it is necessary to use a tool that has the functionality to monitor on operating system basis. Containers do not have their own operating system but containerization adds isolated containers with their own file system. It is necessary that the monitoring solution is capable of monitoring events that occur inside the container. The focus area of the monitoring tool will be generic events that can occur in an operating system. Application specific logs, such as Webserver Access logs, are not concerned as this tool should be usable independent from any applications. The event types that will be monitored are network events, spawned processes and file events.

3.2 Design Considerations

This chapter will give a detailed insight about the design of the product. The requirements that come from our research will be summarized and specified. The tool that suites these

requirements best is then chosen and fitted into the overall context.

3.2.1 Existing DFIR Environment

The new monitoring solution should integrate seamlessly into the exisitng Digital Forensics and Incident Response (DFIR) tools from the 5G-FORAN-DFIR project.

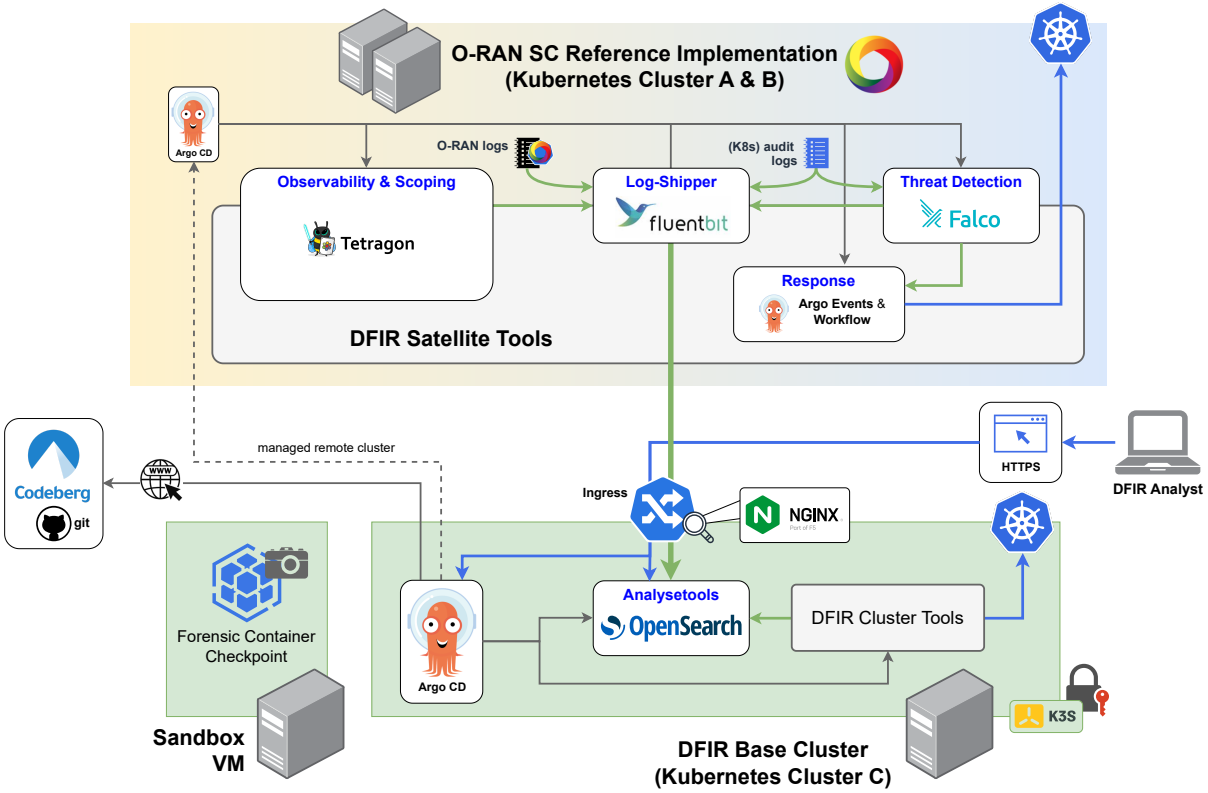


Figure 4: Existing DFIR Environment from 5G-FORAN [23]

Figure 4 shows the DFIR tools where the monitoring tool should be integrated. The base of the deployment is the DFIR Base Cluster. This Kubernetes cluster hosts the central OpenSearch database and the core ArgoCD instance. ArgoCD is used to deploy all Applications on the Base Cluster and to manage the remote clusters. ArgoCD installs the DFIR Satellite Tools on the Open RAN remote Clusters.

The DFIR tools include Tetragon and Falco for Observability and Threat Detection and Fluentbit for shipping those generated events and additional logs from Kubernetes and the Open RAN deployment to the OpenSearch Database.

The generated and persisted Data can then be viewed in an OpenSearch dashboard which is

made available through an Kubernetes Ingress Controller. The cluster state is defined via a Git Repository. All changes in the git repository are automatically applied to the cluster by ArgoCD.

3.2.2 Architecture

The aimed architecture is based on the knowledge gained in the related work section. The Open RAN Cluster is a Kubernetes Cluster with a number of nodes. The first node is a control plane node, the other nodes are worker nodes that host the Open RAN Application.

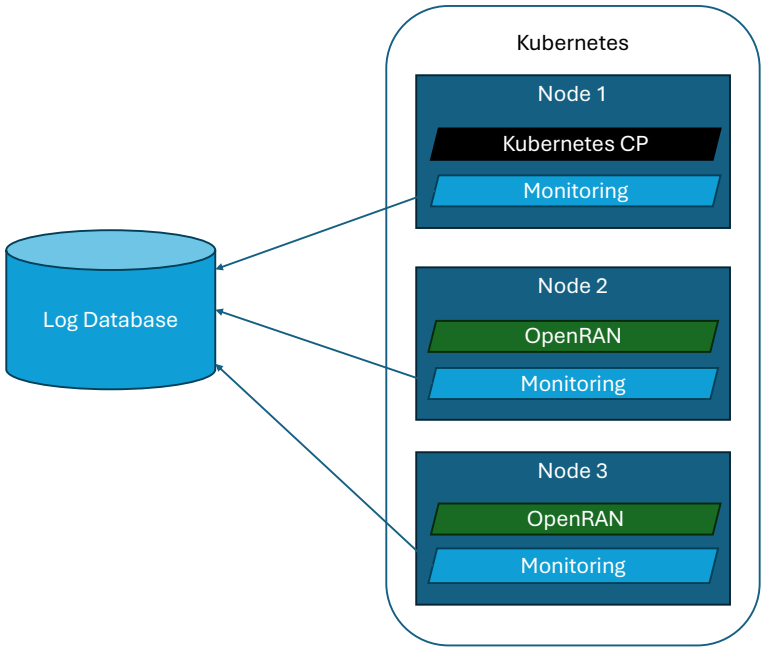


Figure 5: Cluster Deployment

Figure 5 shows the distribution of resources. Every node is deployed with the monitoring component regardless of its function. Even if the Kubernetes control plane does not seem like an entrypoint for an attack, as it is not publicly available, it could be a destination of further lateral movement. The persistence of logs and events is very important for subsequent analyses. Persistence cannot be guaranteed on an infiltrated system, therefore an append only log database must be installed on an external system. A central log database also improves the accessibility to logs as every event can be accessed and analyzed at the same location. When deploying an application on Kubernetes, a description of a desired target state is sent to the control plane. The control plane then creates the resources that meet the target state. The distribution of the resource is mostly a concern of the Kubernetes control plane. It takes

decisions with reference to remaining capacity on the different nodes. Capacity changes on nodes cause movement of resources to other nodes. The state is therefore not permanent and the enrichment of all monitored events with information about the source of the event is required. At the kernel level, containers are namespaces. These namespaces are created when a container is started and deleted after it is stopped. References to these namespaces could help identify the source of an event at runtime, but potentially not at a later date. At least the node, a timestamp, a container name and the container image must be added to each event to identify an unambiguous event source. By storing the container image with its version tag, conclusions about installed software can be drawn. As a Kubernetes cluster itself is also a dynamic component, it is necessary to deploy the monitoring automatically to new nodes that have been added. This is achieved through deploying the application as a Kubernetes daemon set. Daemon sets are used to deploy one replication of a service to every node.

3.2.3 Mapping

Tracing the eBPF is a challenge in terms of filtering for the right events. When attaching to all kernel tracepoints in an idle Linux system, around 99000 events/sec are reported. With Kubernetes running, already around 450000 events/sec are reported(see appendix B). This provides some impression of the degree of filtering that is required. The filtering in this project is performed by using sigma rules. These rules need to be translated to be utilized by the monitoring tool.

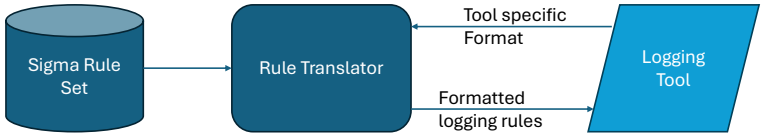


Figure 6: Log Translator

Figure 6 shows the general workflow. The aim of this project is to create a translator that is aware of the tool-specific target format and can apply the format to the Sigma rule set. The key aspect of the translator is that additional rules within the sigma rule set do not require a manual intervention to be translated. This generalistic approach is necessary to reduce the maintenance to a minimum. The detection logic of a sigma rule has 3 components:

Logsource

Possible sigma logsources are mapped to the counterpart of the monitoring tool. As described in section 3.1, file events, network events and spawned processes should be monitored. The sigma logsource of these events is: "file_event", "network_event" and "process_creation". Existing and possible new rules within these logsources should be translated.

Fields

Sigma rules have fields with a value that build the match condition. All fields included in a sigma rule should be supported and translated into the equivalent of the monitoring tool.

Values

The translation or reformatting of values should also be implemented as generalistically as possible. The translation should not fail with new values.

During the lifecycle of the tool, it is possible that the format of sigma rules changes or that log sources and fields are renamed. If the translation for a rule fails due to unknown fields or logsources, the translator should output a warning about the failure but proceed with the next rule, so that one failed rule does not break the whole translation process.

3.3 Tool Evalution

Following the description of the requirements and architectural circumstances, the next step is to choose a tool that meets most of the requirements. The first decision to be made is if the monitoring solution filters in the kernel space or in the user space.
An advantage of kernel space filtering is the low performance impact. For every kernel event that is triggered, an immediate decision is made without further transmission to user space reducing context switching cost. Processing the events directly where they occur also decreases the possibility of tampering by attackers. The disadvantage of kernel space filtering is the complexity of developing kernel space programs. Section 2.3 describes the restrictions for eBPF programs. The limited subset of C functions makes it difficult to implement complexer detection logics.
The biggest advantage of user space filtering is the flexibility. Different languages and libraries can be used to achieve the intended behavior. The access to additional information, like the container runtime information, is unlimited in user space. Errors in user space do not

affect the kernel behavior and are much easier to debug than kernel code. On the base of this arguments, a simple test software was implemented. The two tools “Tracee” and “bpftrace” are tested and rated on how well they meet the requirements.
An integral part of the implementation is the translation. Part of the Sigma project is the pySigma tool which is a tool for writing converters. At the end of this section, it is reviewed if this tool should be used for implementing the converter.

3.3.1 bpftrace

bpftrace is high level abstraction of the BCC language. It uses an awk, C and predecessor tracer inspired language that is compiled into eBPF bytecode [24]. When implementing the rule from section 2.4, the limitations of eBPF are already exceeded.
The rule has two conditions that should be met. The image path should end with “/base64” and the command line arguments should include “-d” at any location. In C language, there would be different ways to achieve this behavior. Approaches would contain the usage of the “strstr()” function that can detect if one string is a substring of another string, or a loop through a string and check if an intersection between the two strings can be found at any location. The problem is that these strings can have an unlimited size. Especially the “base64 -d” command could be given a long base64 encoded string to be encoded. As eBPF does not support loops without predefined length or advanced string functions like “strstr()”, this way would not work. Even a predefined loop with a high number of iterations would work for most cases but not for all. The typical maximum length of a Linux file path is 4096 characters, which would equal a string size of 4096 bytes. The stack limit size is 512 bytes for eBPF programs so any string comparisons are hard to implement. There are possibilities to increase the stack limit by adding helper functions. Unfortunately this approach would increase the complexity significantly and would not fit the approach of a tool that should be configured or created in a generalistic way to translate different rule types.
With this knowledge it can be ruled out that a kernel space filtering solution would succeed in this project.

3.3.2 Tracee

Tracee is an open source project launched by Aquasecurity. Tracee is an eBPF based monitoring tool that is created to detect attacks in containerized environments like docker or Kubernetes [25]. To ensure a seamless integration into existing containerized environments, Tracee is deployed in a Docker container that includes all components of the monitoring chain.

As Tracee is open source, these container images can be modified and rebuilt to implement individual functions [26].

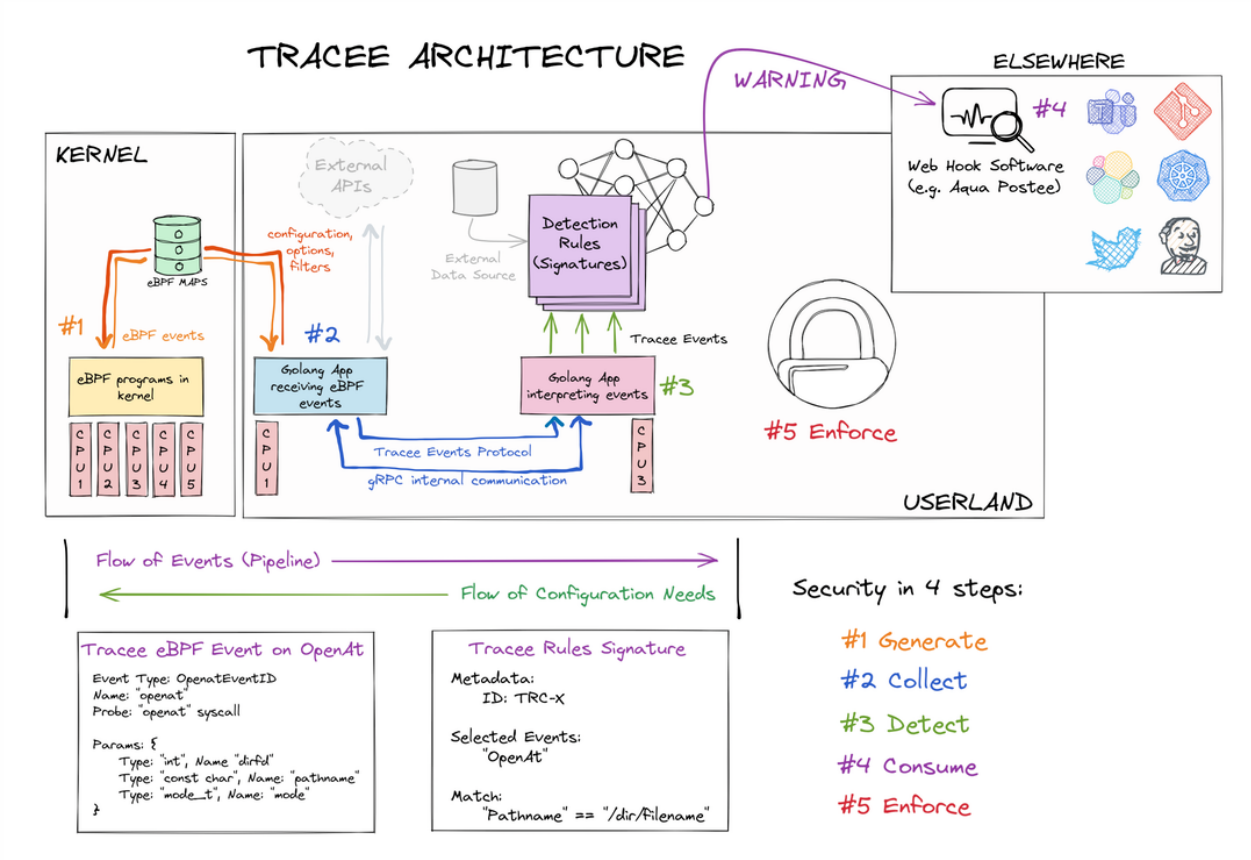


Figure 7: Tracees Architecture [27]

Figure 7 describes the architecture of Tracee. Tracee has three components. The event generation is happening with eBPF in kernel space. Tracee provides a Golang App that configures the eBPF programs and also receives the generated events. The received and formatted events are then enriched with information from the container runtime engine. The events can be unchanged eBPF events or higher level events that are derived from multiple events. An example for a higher level event is the "file_modification" event that is built from "fd_install" and "filp_close" events. Then, all events are forwarded to the interpreting part of the Tracee deployment. There, every event is compared to the predefined signatures to decide if an event is marked as match or is dropped. Filter rules are called signatures in the Tracee environment. Tracee can be configured with Rego or Go signatures. Matched events are output to a file. This file can be monitored for further processing or external persistence. The important difference to bpftrace is the user space filtering which does not restrict the

complexity of filter rules. The difference between the Rego and the Go Signatures is that the Rego signatures can be added at runtime, while the Go signatures must be added at compile time. The Tracee documentation writes about Rego signatures, that come with a performance overhead.
The implementation of the sigma rule from section 2.4, as Rego and Go Signature and its detection part would look like the following.

```
[...]
tracee_match {
    input.eventName == "sched_process_exec"
    cmd_path = helpers.get_tracee_argument("cmdpath")
    endswith(cmd_path, "base64")
    argv = helpers.get_tracee_argument("argv")
    some i
    argv[i] == "-d"
}
[...]
```

Code 2: Implementation of the Sigma rule from section 2.4 as Rego Signature

```
[...]
    func (sig *DecodeBase64EncodedText) GetSelectedEvents() ([]detect.
    SignatureEventSelector, error) {
        return []detect.SignatureEventSelector{
            {Source: "tracee", Name: "sched_process_exec", Origin: "*"},
        }, nil
    }
    func (sig *DecodeBase64EncodedText) OnEvent(event protocol.Event)
    error {
        eventObj, ok := event.Payload.(trace.Event)
        if !ok {
            return fmt.Errorf("invalid event")
        }
        switch eventObj.EventName {
        case "sched_process_exec":
            argv_arr, err  := helpers.GetTraceeSliceStringArgumentByName(
    eventObj, "argv")
            cmdpath, err := helpers.GetTraceeStringArgumentByName(eventObj
    , "cmdpath")
            if err != nil {
                return err
            }
            if strings.HasSuffix(cmdpath, "base64")
```

```
20              {
21                  for _, arg := range argv_arr {
22                      if strings.Contains(arg, "-d"){
23                          [...] //Trigger Event
24                      }
25                  }
26              }
27          }
28  [...]
```

Code 3: Implementation of the Sigma rule from section 2.4 as Go Signature

The Rego format comes from the Open Policy Agent (OPA) ecosystem which is a tool for implementing policy-based control in software systems. As the only purpose of this language is the definition of policies, it is very straightforward to implement the sigma rule. Code 2 line 3 is the matched Tracee logsource for "process_creation". Lines 4 and 6 define which fields should be used, and lines 5 and 8 check if the value matches the sigma rules values. It is remarkable that even the descriptor "endswith" is the same as in the sigma Rule.
The Golang snippet Code 3 shows a lot more code. The layout is the same but not as compact as in the Rego snippet. The logsource is matched in line 4, the fields are retrieved in 14 and 15 and the values are checked in lines 19 and 21.
The Golang snippet might look more complex on first sight, but does indeed offer a lot of flexibility. In the Go signatures, the whole Go language with its packages can be utilized while the Rego language is not a fully capable programming language. As the intention of this work is to write a generalistic translator, the fully customizable codebase of Go is preffered over the Rego format. Another advantage of go files is the potential performance increase through compiling them into the Tracee executable.

3.3.3 pySigma

pySigma is a python library that is used to convert the abstract sigma rule format into queries. pySigma implements classes that can be overridden to implement a specific conversion behavior. Existing implementations are for example "pySigma-backend-sqlite" [28] or "pySigma-backend-elasticsearch" [29]. As the detection conditions and logic of a sigma rule has an abstract format, it makes the implementation of a new Logging Source much easier. The creation of sigma rules is possible without any specific syntax, therefore, it cannot directly be used in database queries or Code conditions.
The backend project itself consists of two parts: A pipeline and a backend. The backend part is used to define a product specific syntax. The pipeline part is to adapt sigma specific

field names or logsources to the destination names and logsources.
The backend is usally utilized to generate short database queries. A Tracee signature needs a lot more logic than a SQL backend for example. To use the pySigma project, a Tracee signature template is needed that is the same for every Signature, despite a small query part. If that is accomplishable, pySigma is a great choice as it already resolves the abstract rule format on its own [30].

3.4 Integration Concept

In the following section, the previous knowledge is combined with the selected tools in an overall design. The overall design also includes automated provisioning via CI/CD and deployment and integration in the Open RAN cluster. Despite the earlier chosen tools, some helper functions are added that support the seamless integration.

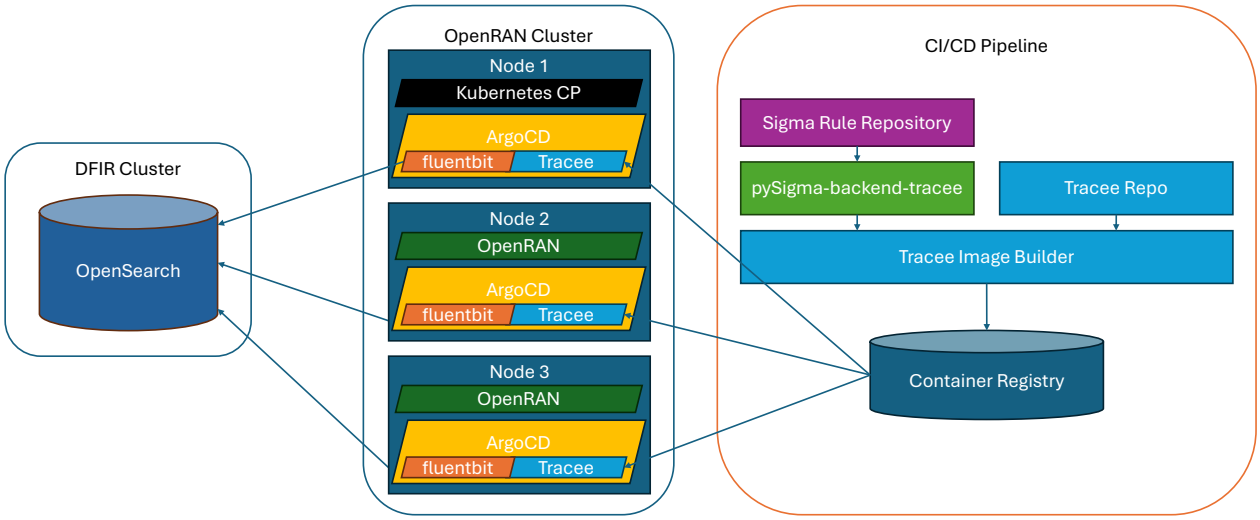


Figure 8: Toolchain of the deployed monitoring solution

3.4.1 Toolchain

To ensure a seamless integration into the existing DFIR Tools and the Open RAN Deployment we use OpenSearch as a Database. All tools that are used in the deployment of Tracee, are briefly introduced.

OpenSearch

The OpenSearch project is used in the DFIR Tools as storage for Java Script Object Notation (JSON) documents. OpenSearch comes with a search engine that enables very efficient filtering and searching and aggregation in unstructured Datasets [31].

Fluentbit

As Tracee does not have any native database connection, an additional tool is required. Tracee can be configured to write data as JSON to a file. FluentBit is able to monitor this file and ship every change to openSearch. Fluentbit can be configured with filters and parsers which are not required in this project because Tracees output format is defined in the Tracee configuration so it can be directly shipped to OpenSearch [32].

ArgoCD

Tracee comes with Helm Charts for an easy Kubernetes Deployment. A Helm chart is a recipe that describes the deployment of Tracee. Tracee can be configured with a policy shipped in a Kubernetes custom resource definition. It is part of the Helm chart and defines which Tracee signatures should be monitored. When deploying Helm Charts to Kubernetes without additional tooling, changes to the Helm charts will not be applied to the Cluster. However, every update in the Sigma rule repository would trigger a change of the Helm charts. As automation has a high priority in this project, Tracee is deployed via ArgoCD to the Cluster. ArgoCD is configured with the repository where the Helm charts are stored, and monitors and applies any changes to the Helm Charts [33].

3.4.2 CI/CD

To decrease the maintanence effort to a minimum, it is important that additional Sigma rules that are pushed to the Sigma rule repository are deployed automatically to our cluster. This is also necessary to always be up to date with the recent exploits and vulnerabilities that should be monitored.
The pipeline has three stages that are executed sequentially. The Pipeline is divided into parts where interesting Artifacts are expected. This is done to achieve traceability of possible build failures or application misbehaviors.

Rule Collection

The first step is to clone the Sigma repository and pack the rules as an artifact for further steps.

Rule Conversion

The previously packed rules are unpacked and translated with the implemented pySigma-backend-tracee. The Artifacts of this stage are the Go signatures that are packed as artifact.

Image build

This stage pulls the Tracee repository into the workspace. The original Tracee signatures are then replaced with the previously generated go signatures. The new signatures are then referenced in the Tracee code to be compiled. The final build of the container image is then executed via a build script that comes within the Tracee repository. The custom Tracee container image with the compiled Tracee binary is then pushed to a local Docker registry. All used Go signatures must be referenced in the Helm chart, so it is updated and pushed to the git repository.

Backend Implementation

The implementation of the converter is based on the developed concept. This chapter will describe the necessary implementation and integration steps for the development of the monitoring solution.

4.1 Development Process

The development process is divided into the development of the rule converter and the steps required to connect all components.
The implementation of a project always starts with the parts that have the highest possibility to fail. As the development of the converter has the highest extent and its feasibility is to be proven, that developement is the first part. Subsequent to a successful rule conversion, the adaptation of the trace output format is concerned. If each component works on its own, the whole system is integrated with the OpenSearch Database. For this integration, the configuration of FluentBit is necessary. All configurations are then described via HelmCharts for an automatic deployment of all components together.

4.1.1 pySigma-backend-tracee

The implementation of the pySigma-backend-tracee that works as the rule converter is the central part of the project. The Sigma project is not typically used to configure the logging behavior of a tool, but rather to search through existing logs from SIEM solutions. Thus, so some additional work needs to be done.
The development of the pySigma-backend-tracee can be split into three parts: The implementation of a signature template, the pySigma backend and the pySigma pipeline.

Signature Template

The pySigma framework is built to create queries that are used to search through databases. As Tracee is configured through Go code, the signatures have a more complex structure than typical database queries. Variables that are used and evaluated in a Sigma rule must first be declared and then initialized.
Each custom Tracee signature derives event data from a built in a Tracee event. To achieve the execution of the newly created event, a callback must be registered at the parent event. The parent event is chosen through setting the "Source:" and "Name:" fields of the "SignatureEventSelector" function (see Code 4). The logsource that is inserted there depends on the Sigma rule and is mapped in the pySigma pipeline subsubsection 4.1.1.

```
func (sig *<SignatureName>) GetSelectedEvents() ([]detect.
    SignatureEventSelector, error) {
    return []detect.SignatureEventSelector{
        {Source: "<rule.logsource.category>", Name: "<rule.logsource.
    product>", Origin: "*"},
    }, nil
}
```

Code 4: Setting the logsource for the signature

In pySigma, it is not possible to use a different template for each logsource, so a template must be created that includes all values and fields for every possible logsource. These fields could then be evaluated for their values in a query with a simple structure like a database would use. As each event delivers different variables, the initialization is performed separately for each event.

```
package main
switch eventObj.EventName {
    case "file_modification":
        file_path, err = helpers.GetTraceeStringArgumentByName(eventObj, "
    file_path")
    case "net_packet_ipv4":
        dst, err = helpers.GetTraceeStringArgumentByName(eventObj, "dst")
        src, err = helpers.GetTraceeStringArgumentByName(eventObj, "src")
    case "sched_process_exec":
        argv_arr, err  := helpers.GetTraceeSliceStringArgumentByName(
    eventObj, "argv")
        for _, arg := range argv_arr {{
            argv = argv + " " + arg
        }}
```

```
        cmdpath, err = helpers.GetTraceeStringArgumentByName(eventObj, "cmdpath")
}
```

Code 5: Initialization of event specific variables

The event source is specified in Code 4, so it can be accessed in Code 5. The variables that are initialized in this section can then be accessed later if needed, within the query. This mapping must be done for every event type that should be supported by the pySigma-backend-tracee.

```
if <query> {
    metadata, err := sig.GetMetadata()
    sig.cb(&detect.Finding{
        SigMetadata: metadata,
        Event:       event,
        Data:        nil,
    })
}
```

Code 6: Template for the detection logic

The parameter "query" is substituted with the query that will be built in the next section. The query includes combinations of comparisons and functions that return boolean values. If the returned boolean is true, the event matched the Sigma rule, so the "sig.cb()" is called with metadata and event information as parameters which ensures that the event is issued.

pySigma Pipeline

The fields in the Sigma rule differ in their names and syntax to the fields in the Tracee signatures. The pipeline part of the pySigma backend is therefore used to rename the fields of a Sigma rule to match the same fields in the Tracee signatures.
The first transformation for each rule is the logsource mapping. As event sources in Tracee have different names than the generic Sigma sources, each logsource must be renamed. The logsource mapping also looks at some Sigma fields to choose the right logsource, so that all fields intersect between Sigma and Tracee sources.
The same applies to the fields. Each field that can be used by a Sigma rule must be mapped to an existing variable in the Tracee Signature Go template.

Sigma Logsource	Sigma Field	Tracee Logsource
linux file_event		file_modification
linux network_connection	Initiated	net_tcp_connect
linux network_connection	DestinationIP	net_packet_ipv4
linux network_connection	DestinationHostname	net_packet_dns
dns	query	net_packet_dns
dns	answer	net_packet_dns_response
proxy	sc-status	net_packet_http_response
webserver	sc-status	net_packet_http_response
proxy		net_packet_http_request
webserver		net_packet_http_request
linux process_creation		sched_process_exec

Table 1: Mapping of Sigma Logsources to Tracee Logsources

The logsource mapping is shown in Table 1. The scope of Sigma logsources is broader than the Tracee counterpart. The Linux network_connection can define packet transfer on different layers. The layer that is needed to be monitored must therefore be gathered from other fields of the Sigma rule. For example the "Initiated" flag states that the TCP three-way handshake was successful, so the field suggests that a TCP connection must be the objective of the rule.
The implementation of the field related log source transformation was done using the "RuleContainsFieldCondition". This condition was not initially supported by pySigma. Thus, an issue with the description of the needed implementation was created in the pySigma repository which was than implemented by the maintainer of the pySigma project Thomas Patzke.

Sigma Field	Tracee Field
c-useragent	httpUserAgent
cs-user-agent	httpUserAgent
c-uri	httpUri
cs-uri	httpUri
cs-method	httpMethod
cs-host	httpHost
sc-status	httpStatusCode
c-uri-extension	httpUri
c-uri-query	httpUri
cs-uri-query	httpUri
cs-uri-stem	httpUri
cs-cookie	httpCookie
dst_ip	dstIP
cs-referer	httpReferer
TargetFilename	file_path
Image	eventObj.ProcessName
DestinationIp	dstIp
DestinationHostname	dstHostname
SourceIp	src
CommandLine	argv

Table 2: Mapping of Sigma Fields to Tracee Logsources

The fieldmapping in Table 2 includes all fields that where found in the implemented Sigma rules. Some fields with different names implement the same logic, for example “c-uri-query” and “cs-uri-query”. The Tracee fields come from the created template and are gathered there from more complex data structures.
Some of the Sigma values have a different format than the Tracee values. For example, the “Image” field which represents the name of an executable in Sigma rules always begins with a “/” because it should match the files path. In Tracee, the name of the executable is taken from the process name which does not include a leading “/”. This difference can be eliminated by using the “ReplaceStringTransformation” with a regular expression.

pySigma Backend

In comparison to the pySigma pipeline part, the pySigma backend part implements a more generic translation. The backend is used to define the operators for different types of comparisons and the general query structure is described. Furthermore, language specific escape patterns and characters are specified. As the task of this backend is to convert something like "image |endswith <string>" into Go code with matching functionality, simple string comparisons do not work. The best way to achieve the functionality is to use regular expressions. Regular expressions can use wildcard operators for ambigous parts of strings. So, the condition "image|endswith <string>" would become "$regexp.MustCompile(`.* < string > \$`).MatchString(image)$". This function returns a boolean value whether the field matches the expression or not.
The concatenation of conditions is done automatically by the pySigma tool. Only the operators must be defined. For other evaluations that cannot be performed with the regexes, such as numeric comparisons, the syntax must also be defined.
The backend combines all steps taken to fill the gaps in the Go template with the mapped log source and the generated query. The result is a complete Go signature with the implemented detection conditions which is than written to a file.

4.1.2 Tracee Output Format

The default output format of Tracce is a JSON which would be the matching format for OpenSearch. However, the Tracee default format is not consistent as it includes fields that have variable types. For some events, these types are values, and for other events, they store arrays. OpenSearch builds its table schema based on the first event it receives. Events with a different format cannot be imported into the schema, causing those events to be dropped. Tracee comes with an output customization tool based on the Gotemplate package [34]. This package is used for filling templated text files with values. To produce the correct JSON format, a template with fixed fields and types is created so it can be imported by OpenSearch.

4.1.3 Fluentbit Integration

Fluentbit is used to send previously generated output of the Tracee container to the OpenSearch Database. To get Fluentbit to work, it needs access to the output of Tracee. FluentBit is deployed via Kubernetes in a container. To provide access to the output, both, the Tracee and FluentBit container need to mount the same folder.

By adding the location of the Tracee output file to an input in the FluentBit configuration, this file is monitored for any file changes. If FluentBit finds any new data, it is send to OpenSearch. As the format of the logs was previously defined to match the OpenSearch Schema, the logs do not need to be parsed by FluentBit anymore. To distinguish logs from other log sources, they are tagged with a "Tracee" tag.

4.1.4 Helm Charts

Tracee comes with Helm Charts for the deployment. These Helm Charts include a default policy that specify which signatures are logged. As this project creates its own Tracee signatures, a policy with the custom Tracee signatures must be created and deployed to the cluster. The Custom Resource Definition just lists all signatures that were created.

```
apiVersion: tracee.aquasec.com/v1beta1
kind: Policy
metadata:
  name: Sigma-tracee
  annotations:
    description: Traces events derived from Sigma Rules
spec:
  scope:
    - global
  rules:
    - event: ADSelfServiceExploitation
    - event: APT40DropboxToolUserAgent
    - event: APTUserAgent
[...]
```

Code 7: Custom Resource Definition for Tracee

4.2 Testing

Software testing is an essential step during development to ensure that the software meets its functional and non-functional requirements. This section focuses on the functional testing of the implemented monitoring solution. The non functional requirements are tested in chapter 5. By systematically testing the monitoring behavior of the application, the tests help validating the software's behavior under various scenarios including normal operations and edge cases. The insights gained through testing are crucial for this type of software product because issues in the detection logic would not be noticeable in productive environments, but

lead to missing security related events.
A usual approach of software testing is to measure the code coverage while executing automated test cases. When each conditional path is tested, the functionality of the application can be confirmed.
Tracee is delivered in a production ready framework which is great for fast deployment of the tool, but would require a lot of effort to implement in an automatic test environment. Tracee has a lot of dependencies to the operating system and Kubernetes, therefore, the testing in this project is executed manually.
The manual workflow of testing Tracee is straightforward. Each Sigma rule has a different logsource with different fields. For a sensible test extent, each logsource is tested with all its fields and with combinations of fields. As combinations have all the same scheme, not every possible combination is tested, but at least one per logsource. Each test is executed based on a matching Sigma rule. To eliminate possible code awareness of the developer, each Sigma rule is translated to a test command using ChatGPT-4o [35]. The generated command is then executed in a Kubernetes Pod on a cluster where the custom version of Tracee is installed. If the event is reported immediately after executing the test, the test is considered as passed. If no event is logged, the test is considered as failed. To prevent the flooding of the database with false positives, edge cases that do not match the rule are also tested. ChatGPT is queried with the following query:
"A Monitoring tool for a Linux system based on the Sigma rule set should be tested for its functionality. Analyze the following Sigma rule for its detection logic and output at least as much commands as needed to fully test the functionality of that rule. Output commands that would not trigger this rule and output commands that would definitely trigger this rule. Remember that software testing should always include edge cases and various combinations of normal operations. Output only the commands without any description. You can use your knowledge about common software issues to write the test commands. Sigma Rule:
<SigmaRule>"
If this example is executed with the Sigma rule from Code 1, the commands in Code 8 are generated.

```
# Commands that would not trigger the rule:
base64 input.txt > output.txt
echo "SGVsbG8gd29ybGQ=" | base64
cat input.txt | base64 > encoded_output.txt
base64 -w 80 < input.txt
echo "Test message" | base64 -w 0
# Commands that would definitely trigger the rule:
```

```
base64 -d input.txt > output.txt
echo "SGVsbG8gd29ybGQ=" | base64 -d
cat encoded.txt | base64 --decode > decoded_output.txt
base64 --decode input.txt > decoded_output.txt
echo "SGVsbG8gd29ybGQ=" | base64 -d | grep "Hello world"
```

Code 8: Test commands generated by ChatGPT

This code is then executed in a container on the destination host. Possible dependencies like files or executables are previously added without further notice.
If a test is not passed, the issue in the code has to be fixed. After the fix, all tests must be executed again until all tests pass. A Large Language Model (LLM) like ChatGPT can make mistakes. All test fails are questioned for the reason of their failure. If the failure is caused by a wrong test command, the command is skipped. To have an overview over the test effort that was made, Table 3 shows which logsource and field combinations where tested with how many different tests. All tests can be found in the pySigma-backend-Tracee project repo [36].

LogSource	Tested Sigma rules	Fields	Positive tests	Negative tests
file_event	1	Image, Targetfilename	4	5
network_connection, dns	2	Image, DestinationHostname, DestinationPort, DestinationIp	31	19
process_creation	2	Image, CommandLine	11	11
proxy_generic, webserver_generic	4	c-useragent, cs-method, c-uri, cs-host, c-uri-extension, cs-cookie, cs-uri-query	30	22
6	9	13	76	57

Table 3: Testcases for the implementation of the pySigma-backend-Tracee

As this tests can be considered as acceptance test, the code base was improved until all tests were passed true positive and true negative. For the further implementation of log sources or different fields, these tests must be expanded. As every change in the codebase could break the functionality, all tests must be repeated before publishing a new release. When the applications complexity will further increase, automatic testing should be concerned in the future.

Evaluation

5.1 Performance Measurements

The implementation of observability always comes with resource costs. In order to assess the extent to which the implementation can be used in a real world scenario, some metrics are collected below.

5.1.1 CPU

The CPU consumption of Tracee is related to the load on the cluster. The more events on the kernel occur, the more filtering must be done by Tracee. To get a good performance insight, a 5 node cluster with limited resources is used. The cluster includes deployments for Tracee, the Open RAN Non-RT RIC, benchmark tools to increase the CPU utilization and tools for performance measurements and visualization. The test cluster has 1 control plane node and 4 worker nodes with each 8 CPU Cores, 8 GB of Memory and 12 GB disk size. These small resources are chosen to have a higher relative utilization of the cluster. The measurement is performed with different loads. The first measurement is an idle measurement where the Non-RT RIC is deployed without any additional load. With 8 CPU cores, the maximal CPU utilization would be 800%.

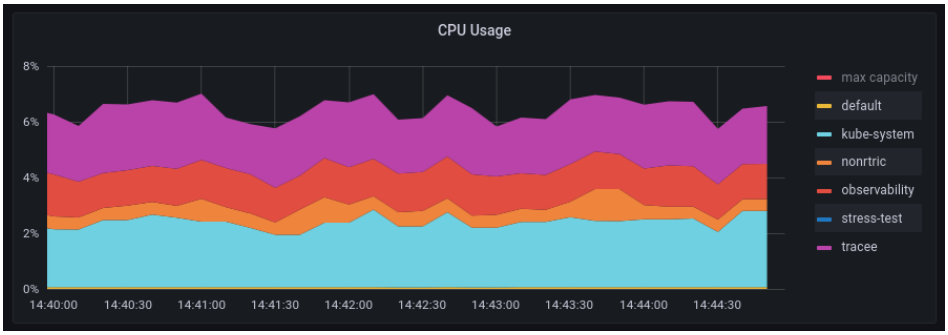


Figure 9: CPU Measurement Idle State. **Tracee 2%**, **Total 7%**

An idle system is not very meaningful to characterize the performance impact of Tracee (see Figure 9). To get a better impression whether Tracee takes a lot of system resources to observe events, the system congestion is measured with artificial workloads.
The custom Tracee deployment observes file changes, network traffic and processes that are spawned. These categories are evaluated with the following measurements.

File Access Stress Test

Tracee observes every file change and checks if it matches any rule that is loaded into Tracee. The expectation is that the CPU utilization increases with the amount of file changes.
To measure the performance, a script is executed that writes files sequentially to the disk.

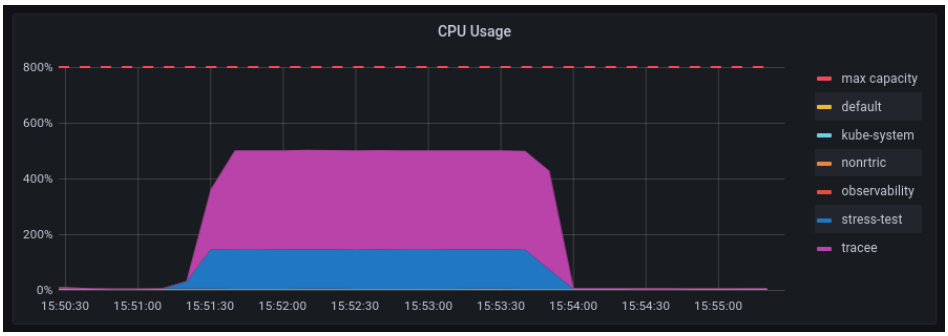


Figure 10: Disk Stress test with 64KB files. **Tracee 355%**, **Total 520%**

Figure 10 shows that Tracee takes around 350% of CPU time which is a significant amount. The same test is executed with larger files to check the relation to the file size.

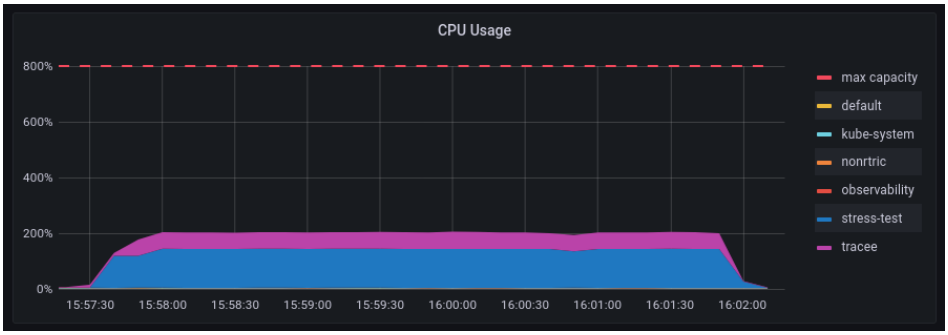


Figure 11: Disk Stress test with 4096KB files. **Tracee 60%**, **Total 200%**

As visible in Figure 11, the file size does matter a lot in terms of Tracees overhead. Tracee does not process the whole file, only the metadata of the file such as path, time and device the file belongs to. By increasing the file size, the number of files written to the disk is reduced and fewer events must be processed by Tracee.

Rule Set size

Another factor that has a huge impact on Tracees performance is the size of the rule set. This test was executed again with the sequential writing of 64KB sized files. By decreasing the size from 212 rules to 1 rule, the CPU utilization is decreased from **355%** to **0.14%**.

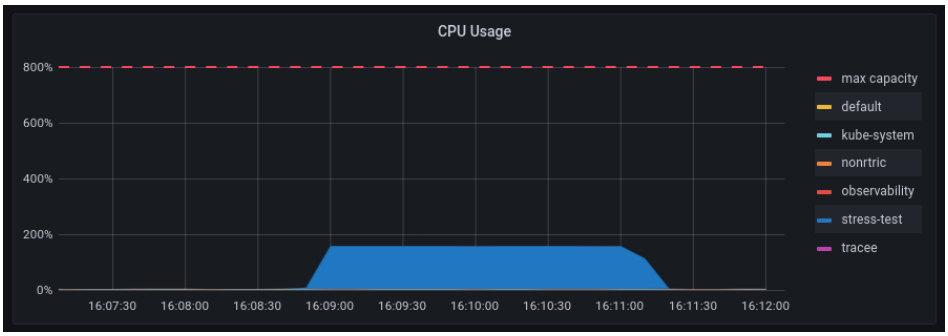


Figure 12: Disk Stress Test with 64Kb Data set and only 1 Sigma Rule loaded **Tracee 0.14%**, **Total 160%**

This high difference in CPU utilization is concerning as it implies that enlarging the rule set would further increase the overhead of Tracee.

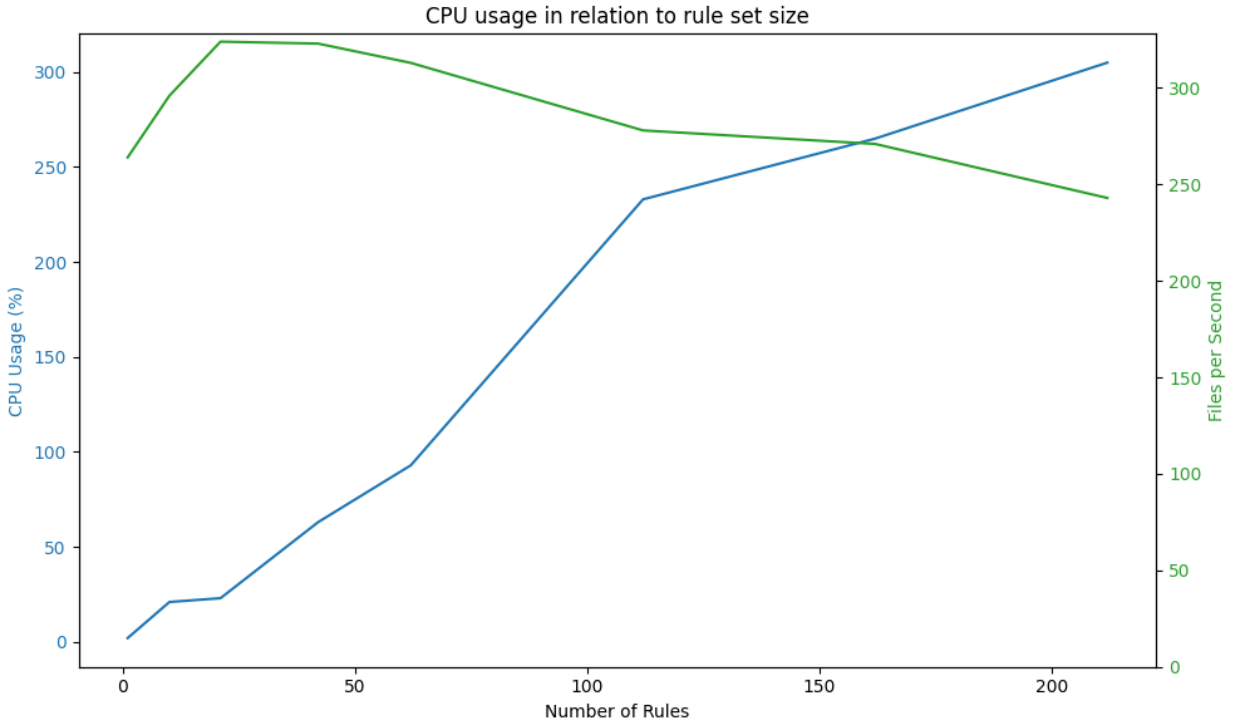


Figure 13: Measurement of performance with different rule set sizes

The blue graph in Figure 13 gives a more detailed view over the relation between the CPU usage and the rule set size. The green line shows how many files were written to the disk per second during the stress test. As expected the Diagram shows that with a growing size the CPU utilization of Tracee increases. The green line shows that the performance of the stress test is also impacted by the rule set size.

Network stress test

To test the Network related performance of Tracee, a container is used that downloads a 6GB file with a speed at around 160Mbit/s.

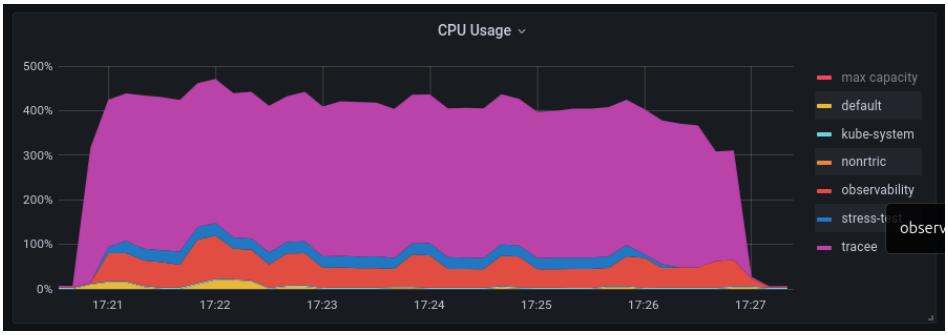


Figure 14: Network Stress Test with 160 Mbit/s download **Tracee 330%**, **Total 450%**

Again Tracee adds significant overhead to the CPU load of the node. When observing the graph of Tracee in relation to the graph of the stress-test container, it is also visible that Tracees graph takes longer to settle down. That comes from Tracees built in caching mechanism that caches events when Tracee is not fast enough to process them.

5.1.2 Memory

The idle memory Consumption of Tracee lays at around 220 Megabytes per node. Even under increased loads, this values does not change. If the load however exceeds a limit where Tracee is not able to process the events as fast as they occur, Tracee begins to cache them. This results in higher memory consumption. The maximum available memory is set in the Tracee configuration file.

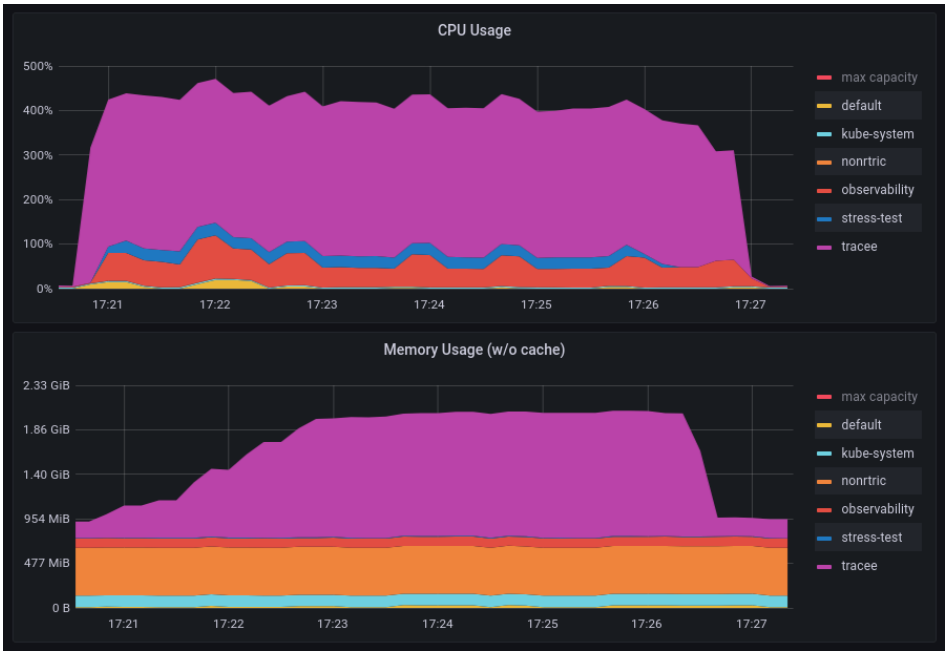


Figure 15: Memory Consumption while Caching. **Tracee 1.35GB**, **Total 2GB**

When the cache is full, additional events are dropped with a warning e.g. "Lost 11988 events".

5.1.3 Time accuracy

The time between the occurrence of an event and the Tracee output is dependent on the congestion of Tracee. If Tracees cache is used heavily, the output of the event is delayed. In normal operating state the output is near to instant. More important as the response time is the accuracy of the timestamps. The timestamping of events happens in the eBPF. Therefore, the timestamp is as accurate as it can be. The caching of Tracee does not have any effect on the timestamps.

5.1.4 Ease of use

The custom version of Tracee is very easy to use. In general, it can be used the same way as the official Tracee. The only thing that has to be changed in the deployment instructions is the container image name, so that the custom Tracee is pulled. Besides the container image, the project related repository includes also Helm charts that lighten the deployment.
On a running Kubernetes Cluster with argoCD installed, the whole tooling can be installed with just adding the repository to argoCD. The rest of the deployment is then automatically

done by ArgoCD. Tracee does not need any more manual intervention. Events can be logged to a file or a database. Within the **5G-FORAN** project, the events are logged to an OpenSearch Database which enables advanced filtering and aggregation mechanisms.

5.1.5 Ease of adaption

The idea of combining Tracee with Sigma improved the useability of Tracee. By using an established abstract rule format, it is not necessary to understand the complex signature format of Tracee and rules can be reused and shared between different tools.
During the testing of Tracee in the Open RAN deployment with some example attacks, some missing detection vectors were found (see below section 5.2.2). The process of developing one of these rules is exemplary evaluated.
The purpose of the created Sigma rule is to monitor changes of the "authorized_hosts" file. This file stores the public part of a SSH key that gives remote access to the system. Attackers could use this way to get a persistent access to an attacked system.
The first step is to fill the Sigma rule with metadata. The metadata is important to understand why the Sigma rule exists and what potential danger is connected to the rule. A known attack vector like the persistence via SSH can be looked up in the Mitre ATT&CK Matrix [20]. The found **T1098.004** Technique is then linked in the Sigma rule with its category in the tag field. The other metadata fields of the Sigma rule are self-explanatory.
The detection of this rule is simple. If a file modification happens and the path includes "authorized_keys", the rule should be triggered.

```
title: Insert key to authorized Keys
id: a9d876ea-d1c3-4bb2-8af4-56c85989cbb0
status: test
description: Detects changes of the .ssh/authorized_keys
references:
    - https://www.ssh.com/academy/ssh/authorized-keys-file
author: Henrik Wittemeier (TH Koeln)
date: 2024-11-15
tags:
    - persistence.account-manipulation
    - persistence.T1098.004
logsource:
    category: file_event
    product: linux
detection:
    selection:
```

```
        TargetFilename|endswith: 'authorized_keys'
    condition: selection
falsepositives:
    - Accepted Changes of Authorized Keys
level: high
```

Code 9: Custom Sigma Rule

When the Sigma rule is pushed to the **pysigma-backend-tracee** repository in the correct directory, a CI/CD pipeline is initiated that converts the Sigma rule to a go signature, compiles it into Tracee and uploads the custom Tracee container image to a registry. Another file that is changed is the Custom Resource Definition which holds all events that should be observed by Tracee.
As the recommended deployment involves argoCD, every change in the Repository is automatically applied to the Kubernetes cluster. The change of the custom resource definition triggers argoCD and initiates a restart of Tracee. During the restart of Tracee, the new container image is pulled.
So, the procedure of expanding the rule set involves only two steps, creating the rule and pushing the rule to the repository. After pushing the rule to the repository, no further action must be taken.

5.1.6 Falsepositives

False positives have an important role in monitoring tools. A high number of false positives floods databases and distracts from true positives.
The Sigma equipped Tracee generates false positives for Network discovery and System Discovery. These rules are triggered by the Container Network Interface (CNI) container executing the “ip”, “iptables” and “hostname” command that needs the information for functioning. These false positives come from the difference of the typical Sigma application area and the adapted Kubernetes context.
The false positive rate is at about 90 hits per minute.

5.2 Security Effectiveness

As mentioned above, this master thesis was initiated in the 5G-FORAN research project. The success of the research project is measured by combining both sub projects. The 5G-FORAN-ATTACK part executes attacks and the 5G-FORAN-DFIR project analyzes their collected logs if any forensic artifacts can be found to reconstruct the attack.

The tools used for generating forensic artefacts where Falco, Tetragon, k8saudit and Tracee with the custom rule set. In this thesis, only the results for Tracee will be analyzed. The following chapter will characterize the security effectiveness by reviewing the generated events of Tracee for specific attacks. Tracee is tested in this chapter configured with the public available Sigma rule set.

5.2.1 Review Methodology

To rate the security effectiveness of the results, following test outcomes are possible

- DS - Successful detection - An event that matches the attack was generated
- PS - Partly successful detection - An event was generated which is part of the attack but not the attack itself
- NS - Detection not successful - No event was generated, creating a matching Sigma rule is possible
- NP - Detection not Possible - No event was generated and it is not possible to create a Sigma rule for this attack

For all attacks that are not successfully detected, it is investigated if a Simga rule could be created to detect that event. Only if no detection would be possible with a custom Sigma rule, the result is "Detection not possible"

5.2.2 Attacks

ID-KD-01

This attack aims at detecting the API endpoints of the cluster using *kdigger* to reveal sensitive information and installed components.
Result: PS
Tracee threw events for the installation of packages and the changing root certificates, which indicates malicious behavior but did not detect the execution of *kdigger* itself. The successful detection could be achieved by implementing a Sigma rule which matches the *kdigger* process name.

ID-FS-01

The attack executes an Secure Shell (SSH) file transfer to a service running on a non-standard SSH port on an external host.
Result: NS
The detection of SSH traffic is not successful as a rule for it is missing in the Sigma ruleset. It is not possible to write a rule for Tracee that detects SSH traffic on non- standard ports because Tracee events can only be evaluated with the TCP header, which does not contain any information about the application protocol. To detect this event, a rule was created, that matches every packet that is send to a non local IP address. As the event is enriched with information about the process and container in the cluster, the malicious behavior can be detected.

ID-KTM-01

This attack uses a cronjob to persist the attackers SSH key to the "authorized_keys" file. The effect is that the attacker can login to the machine using SSH without credentials.
Result: NS
Sigma rules for the modification of cron files under "*/etc/cron.d/*" are existent and would be triggered. The Cronjobs however are not created on the system but in Kubernetes. To monitor the change of the "authorized_keys" file, a Sigma rule was created that was triggered by the attack.

ID-KTM-02

To gain acces to the Kubernetes API, this attack copies the Kubernetes configuration into a Pod and then tests the connectivity by enumerating all running Pods.
Result: NS
The detection was not successful because the Sigma rule set does not contain any Kubernetes related Sigma rules. It would be possible to monitor the copying by creating a Sigma rule that detects file changes from the *kubectl* process but it would be easier to reconstruct those types of events from Kubernetes logs itself.

ID-KTM-03

This attack spawns a shell in a container.
Result: NS
The spawning of shells in a container could be observed by tracking the Kubernetes API

endpoint. Another approach would be the observing of any shell processes such as *bash*, because they are not expected in environments without human interaction.

ID-KTM-04

This attack performs a SSH session to another pod that has a SSH server installed.
Result: NS
As SSH is not generally used for malicious activities on Linux systems, there is no Sigma rule that matches the execution of SSH. A Sigma rule that matches the SSH process name or network traffic on port 22 could detect this behavior.

ID-KTM-05

This attacks uses a daemonset to alter the "authorized_hosts" file for a persistent backdoor.
Result: NS
Just as the Attack **ID-KTM-01** the detection was not successful. With the creation of the Sigma rule mentioned in attack **ID-KTM-01**, the file change got visible. Both attacks change the file from a different attack vector. The created Sigma rule only detects when this file is modified, so detection is successful regardless of the source of the attack.

ID-KTM-06

This attack utilizes a privileged pod to mount the hosts filesystem.
Result: NS
The mounting of the host file system is not monitored by default through a Sigma rule. By creating a Sigma rule with the detection for execution of "mount" commands, this attack can be made visible. The detection of privileged containers is not possible with the current implementation.

ID-KTM-07

This attack is an extension of the prior **ID-KTM-06** and is used to escalate privileges from a container.
Result: NS
The detection is also unsuccessful as the mounting is not observed. With the created Sigma rule for Attack **ID-KTM-06**, it can be made visible.

ID-KTM-08

When specifying the right parameter at pod creation time, it is possible to disable the namespace isolation security mechanism that containers usually have.
Result: NS
As the pod creation process is an Kubernetes internal procedure it is not possible for Tracee to match any Sigma rules for this attack. A possible way to monitor this event is by tracking requests to the Kubernetes API.

ID-KTM-09

The deletion of all Kubernetes related events is a technique used for defense evasion.
Result: NS
As the Kubernetes events are application intern artifacts, it is not possible for Tracee to track the deletion of events by observing file changes. It could be observed by tracking the Kubernetes API requests.

ID-KTM-10

The creation of a pod with a name similar to a Kubernetes system pod is a way to obfuscate malicious Pods.
Result: NP
Because Tracee has little information on the creation of pods, it is very unlikely that this behavior can be observed. The creation of the pod could be observed by tracking requests to the Kubernetes API, but it would require more information to distinguish between malicious and normal creation of Pods.

ID-KTM-11

This attack spawns a shell inside a container and deletes all logs that are located in the default log directory ("*/var/log/*").
Result: DS
The Sigma rule "Clear Linux Logs" is triggered by this attack.

ID-KTM-12

The access to service account tokens is used for interaction with the Kubernetes API
Result: NP

The read-only access of a file cannot be tracked by the current implementation of the custom Tracee. Therefore, it is not possible to Track the access to service account tokens.

ID-KTM-13

This attack lists all service accounts that are mounted to the Pod.
Result: NP
As above the read-only access can not be tracked with the custom Tracee

ID-KTM-14

Sensitive data can be extracted using the "kubectl describe pod" command.
Result: NS
As kubectl utilizes the Kubernetes API, it would be possible to observe this attack by creating a Sigma rule that tracks API requests.

ID-KTM-15

Naabu is used to scan hosts in the Pod subnet.
Result: NS
A Sigma rule matching the *Naabu* process name could observe this attack.

ID-KTM-16

Naabu is used to scan services in the Kubernetes service subnet.
Result: NS
As above it would be possible to observe this event by matching the process name in a Sigma rule.

ID-KTM-17

This attack aims at deleting all Kubernetes resources.
Result: NP
The Tracee worker is part of the Kubernetes cluster. By deleting Kubernetes resources, Tracee would also be removed and not able to detect this attack.

ID-KTM-18

By changing the ConfigMap of the CoreDNS pod, it would be possible to change the DNS resolution for lateral movement.
Result: NS
As ConfigMaps are an Kubernetes internal artifact, they are not trackable by observing the file system. ConfigMap changes could be observed by tracking requests to the Kubernetes API.

ID-XAPP-01

This attack uses an malicious Open RAN xApp that creates a reverse shell connection to a command and control server. The reverse shell could be utilized to exfiltrate data or perform further attacks.
Result: NS
As the reverse shell creates a socket to a random port with the TCP protocol, it is not possible to characterize a connection as reverse shell. The only possible detection mechanism is to monitor IP traffic to a public IP as this kind of traffic is not expected in the environment. This is done with the same rule created for **ID-FS-01**.

ID-XAPP-02

This attack combines three steps. First, a network scan is executed with *nmap*. Second, a *netstat* and *ps faux* is performed. Third a reverse shell connection is initiated using *ncat*. The reverse shell connection is used to execute the commands *ls*, *id* and *whoami*
Result: DS
All steps of the attack are visible in the event log. The first step is monitored with the Sigma rule called "Linux Network Service Scanning Tools Execution". The second step triggers the Sigma rule "System Network Connections Discovery - Linux" and "Process Discovery". The third step triggers the rules "Linux Reverse Shell Indicator", "Local System Accounts Discovery - Linux". As all steps of the attack are logged, it is safe to say that the detection was successful.

ID-XAPP-03

This attack replicates the previous attack but changes the names of the executables, so that detection rules based on process names fail to detect the malicious behavior.
Result: NS

As Sigma rules are highly dependent on the process names, the detection is not possible. The only way that the reverse shell could be monitored is with the previously developed Sigma rule that detects traffic that leaves the local subnet.

ID-XAPP-04

This attack again uses a reverse shell that is used to do network and process discoveries.
Result: DS
This detection was successful as multiple Sigma rules where triggered during the attack. The initiation of the reverse shell involved *curl* which was startet with the *nohup* command. The *nohup* command is used to continue processes even after the user shell aborts. Sigma has the rules "NohupExecution" and "CurlUsageonLinux" that were triggered by the initiation of the reverse shell.

ID-XAPP-05

This attack uses a compromised xApp to intiate a reverse shell, do a *nmap* portscan and then perform API requests to Open RAN specific endpoints.
Detection: PS
The detection of the reverse shell was again not successful until the custom Sigma rule was used. The API requests to the Open RAN endpoints were visible due to the usage of *curl* which triggered the "CurlUsageonLinux" rule. The detection could be further improved by creating specific rules that match the Open RAN endpoints and check if the requests came from a real xApp or from the command line with *curl*.

ID-XAPP-06

The attack again uses an compromised xApp to initiate a reverse shell and analyze the service endpoints of the xApp Manager. The difference to previous commands is that it uses *curl* with an obfuscated name.
Result: DS
The main part of the attack was the service endpoint detection with the obfuscated curl command. The detection was performed without a user-agent string in the http request. This triggered the sigma rule "HTTPRequestWithEmptyUserAgent".

5.2.3 Conclusion

The results of the attacks simulation are concluded in the following table.

Result	Number	Percentage
Detection Successful	5	17%
Partly Successful	3	10%
Not Successful	18	60%
Not Possible	4	7,5%

Table 4: Results of performed Attacks

5.3 Comparative Analysis

The above summarized attacks where executed on a system, where other monitoring tools where installed to. In this section the differences in the detection are described briefly.

5.3.1 K8s Audit

Kubernetes auditing is a function of Kubernetes that writes security relevant events to a log file. The events are enriched with information about the initiator, time and where the event occurred.
The steps of the Kubernetes related attacks, in particular **ID-KTM-12** and **ID-KTM-13**, are visible in the Kubernetes Audit logs. Tracee is not build for the visibility of Kubernetes internal artifacts and does not have this function implemented yet.
To achieve a similar logging behavior as Kubernetes auditing, Tracee would have to observe the Kubernetes API endpoint with some complex Sigma rules.
Kubernetes audit logging does indeed give good insights into Kubernetes internal events, but can not give further information.
The audit logging for example returns for event **ID-KTM-03** "Exec or attach to a pod detected". For further investigation, Tracee could see what happens during the exec session and trigger rules based on any behavior.

5.3.2 Tetragon

Tetragon is utilizing the eBPF like Tracee. The differences in the detection come from the different rulesets. Tracee equipped with the Sigma rule set is not optimized for Kubernetes whereas Tetragon was developed for Kubernetes clusters. Tetragon extends its capabilities of monitoring with the ability of runtime enforcement. Tetragon is a sub project of Cilium. Cilium is known for their Container Network Interface. Therefore Tetragon has a high focus

on network related events. Tetragon considers network traffic in flows whereas Tracee supports per packet analysis.
Tetragon is built for detecting behaviors rather than raw file activities. Tracee has strong support for raw file system operations.
As mentioned above, Tetragons strengths lay in Network monitoring. The attack **ID-FS-01** was detected by Tetragon. Tracee did not detect it, because SSH Traffic would not be characterized as malicious in the typical Sigma application area.
Other attacks as **ID-XAPP-02** where not detected by Tetragon as good as by Tracee, because Sigma known tools and attacks where used to initiate a reverse shell and get information about the system.
In terms of configurability, Sigma has an advantage due to its easier and established rule format over the proprietary Tetragon rule format.

5.3.3 Falco

Falco also uses the eBPF to implement Kubernetes runtime security. Falco is directly developed for Kubernetes, so it covers attacks on the Cluster level better than the Sigma equipped Tracee.
Falco is able to implement higher level rules such as DNS enriched Network filters or insides on connection level instead of only packet by packet classification.
Custom rules are created using the custom Falco rule language.
In the above attacks, Falco had its strength in detecting **ID-KTM-03**. Changes of the authorized hosts in attack **ID-KTM-01** did not trigger a Falco rule.

Discussion

This chapter discusses the elaborated results, rates the fulfillment of the objectives and describes the usability of the developed tool.

6.1 Interpretation of Results

The previously collected statistics and metrics can be used to rate the success of the implementation of the pySigma-backend-Tracee and all its involved components.
The measurements about the resource allocation done in section 5.1 are interesting for the prediction of the scalability of the approach.
While the memory usage of Tracee can be managed with the setting of configuration parameters, the CPU usage is heavily dependent on the nodes workload.
With a memory consumption of around 220Mbytes, Tracee does not add much overhead to the cluster. While the test cluster had a per node RAM size of 8GB, usual production grade nodes would have at least 128GB RAM. Taking this into account the RAM usage of Tracee is negligible. The RAM usage increases when the cache is filled with events. This should only happen during workload peaks and is configurable via configuration parameters. Under high fluctuating workloads, it should be concerned to increase the maximum cache size to prevent the loss of events.
Unlike the RAM usage, the CPU usage will scale linear with the workload. This means that on larger nodes, the relative CPU utilization would be the same as in our test cluster. The cost of an idle value of 2% is acceptable and does not impact the performance of the cluster very much. The CPU usage at congestion is however concerning. During the stress test Tracee makes up around 70% of the total CPU usage. This value is unexpectedly high and does exceed the acceptable overhead that a monitoring solution may add. The same applies to the network utilization. While the network interface is utilized at 16%, Tracee uses 40% of the CPU capacity. In the network measurement, it was also visible that the cache was fully

utilized and additional events were dropped. An attacker with the knowledge about Tracee could overflow the cache with random events, so that security relevant events are dropped before evaluation.
The measurements showed that the high CPU usage is connected to the number of signatures that are evaluated. The Sigma rule set is expected to grow in the future, so the performance is likely to decrease with every additional rule.
The security effectiveness tested with the attacks from the 5G-FORAN-ATTACK project give the impression that the tool is not very effective. Most of the attacks where not or not completely detected by the custom built Tracee. This is not an issue of this project but the lack of Kubernetes related rules in the Sigma rule set and the missing focus on containerization in general. The lack of Sigma rules associated with Kubernetes is exacerbated by a focus on attacks at the infrastructure level. Attacks from inside a container like **ID-XAPP-02** are more similar to attacks on standard Linux computers. Therefore Sigma rules apply better and detect the malicious activities. However, the analysis of the attacks also showed that for 92.5% of the attacks, the creation of Sigma rules would be possible. The development of Sigma rules for some of the attacks showed that the rate of detected attacks could be increased significantly with low effort as the development of new Sigma rules is very straightforward.
The comparison with other tools showed that every tool has its advantages and disadvantages. It is not sensible to built the complete monitoring system based on one monitoring tool like Tracee with Sigma rules as, for example, Kubernetes auditing does a very good job at monitoring changes and actions inside the cluster. As stated in section 2.1, the events should be logged at the location where the most information about an event is available. For Kubernetes resource changes like credentials, it would require the aggregation of many different kernel hook points to generate events that have the same amount of information as a Kubernetes audit log.
In comparison to Tetragon and Falco, the custom Tracee detection was moderately well. For some attacks, Tracee delivered more events and for other events, Tetragon and Falco delivered better output.

6.2 Connection to Objectives

The idea of this thesis was the adaptation of the Sigma rule set for the usage with in the eBPF in Open RAN. The main goal of this project was the minimization of the maintenance

effort.
During the evaluation, it became clear that directly integrating Sigma rules into eBPF programs does not seem to be possible with a generalistic approach. The user space filtering comes with a performance impact but grants access to higher level programming languages and sources for information enrichment. The tool Tracee that already implement most of the required functions was chosen. The possibility to implement detection rules in Go code offered the flexibility to implement the generic translation process of Sigma rules to match eBPF events.
The detection targets file modification, network traffic and spawned processes were implemented in this thesis with success. The pySigma-backend-Tracee translates Sigma rules to Go signatures which were compiled into Tracee. The objectives of the translation were designed in subsection 3.2.3. The Mapping of the Sigma logsources, fields and values was successful implemented. The functional requirements were tested with the use of simple but efficient testing. By generating bash scripts to trigger the rule detection mechanism, the accuracy related to true positives and true negatives was confirmed.
The deployment of the custom Tracee is done through argoCD which enrolls the tool automatically, therefore, the integration into Kubernetes is very simple. Tracee is deployed via a Kubernetes daemon set, so each node gets its own Tracee instance. Fluentbit is also deployed via a daemon set, therefore Events generated are read by the Fluentbit instance and then stored in the OpenSearch Database. All events that are generated include information of the associated Sigma rule and are enriched with information about the container, its image, process IDs and other important information that can be used to reconstruct the events.
The project includes a simple seamless deployment which integrates into the Open RAN DFIR tools very well. Deploying the Sigma rule equipped Tracee is easy, because all tools needed for the integration are already present in the 5G-FORAN DFIR tools. The eventlog of Tracee can be accessed in the central dashboard that is also used by the other tools involved.
A key benefit of the deployment is the automatic update feature. For every update in the Sigma repository, the translation process is triggered and all needed artifacts are built and deployed without manual interaction. If no functions were added to the project and the format and syntax of the Sigma rules would not change, this project could run with an up-to-date version in the future without manual intervention.

6.3 Comparison to existing solutions

Outstanding of this project is, related to the other considered solutions and the official Tracee, the easy adaptability. Through the combination of the simple Sigma rule format that allows creating detection rules without any understanding of programming languages and with the automatic integration process with CI/CD, the addition of rules is possible with minimal time expenditure. Besides the large open source Sigma rule set, some companies develop Sigma rules as a product like SOCPrime [37], so better detection results could even be achieved with purchasing more Sigma rules.
The performance of Tracee however is not outstanding. Tetragon has a similar functionality as Tracee, hence a comparison makes sense. While concrete performance numbers are rare to find, Isovalent published a report about Tetragon advertising it for its high performance. Their research resulted in performance overheads of 0.2% for file monitoring, 1.68% for process execution tracking and a reduction of the network connection rate by 5.9% [38]. These numbers, however, could also come from the small native rule set Tetragon comes with, as Tracee was also observed with very small CPU utilization with smaller rule sets.
For Falco, no such measurements where found.

6.4 Limitations

The usage of user space filtering comes with a performance overhead. The context switching cost and the higher complexity of user space programs decrease the performance of the monitoring solution significantly.
Only Sigma rules were used that have the implemented logsources and that are built for Linux systems. Further checking of the meaningfulness of the rules related to the use case is not performed. As some rules are expected to not enlarge the observability in this use case, they decrease the performance without any benefit.
The measurements showed that Application related traffic like SSH traffic is not detectable, as Tracee evaluates filter rules per packet. To detect SSH traffic, the whole connection must be observed.
For file system related Sigma rules, the custom Tracee only supports file modification events. As no Sigma rules were found that match the readonly access to files, the pySigma-backend-Tracee does not support the conversion of this at the moment.

6.5 Future Research and Development

The biggest downside of the implementation is the poor performance. The performance was measured under heavy synthetic workloads. As these workloads utilize only one kind of resource, the result is not completely transferable to real world scenarios with mixed workloads. For better statements about the usability, Tracee should be executed and measured within real workloads. The performance of Tracee could possibly be further improved by changing the mapping. At the moment, each string comparison is performed with a regular expression. This was necessary because the mapping functions of pySigma where limited. When extending pySigma to support more different mappings for different types of evaluations, regular expressions could be replaced with standard operators resulting in a lower resource usage.
The detection capabilities of the Sigma based Tracee are huge, however, the rules that exist in the Sigma repository did not match the attacks very well. The extension of the Sigma rule set with a higher focus on Kubernetes and containerization related attacks would improve the detection.
Tracee also offers the possibility to match container creation and remove events. In further improvements, it would be valuable if Sigma rules and the mapping of the pySigma-backend-Tracee are extended to match these events.

6.6 Critical Reflection

The project of developing a Sigma rules based eBPF logger for containerized environments was a success. The objectives were achieved and the solution is running and deployable.
The open source Sigma rule set includes many important rules, but for a real world usage of this approach, it would be necessary to use a larger and more comprehensive rule set to have a sufficient observability.
The collected performance metrics also indicate that the implementation is not ready for a real world usage.
The idea, however, is very promising and could establish with further improvements and more specific Sigma rules.

Conclusion

In this work, a monitoring solution was implemented that combines the eBPF with Sigma rules. The work was initiated in the research project 5G-FORAN that aims at implementing DFIR in Open RAN.
The development of the software involved a review of the technical background and the requirements of the implementation. The monitoring tool should be deployed in an Open RAN installation running in a Kubernetes Cluster. The dynamic nature of Kubernetes and Containerization adds challenges in terms of observability because the user space of a container is isolated from the operating systems user space and workloads are distributed over multiple nodes. The solution to overcome this problem is a kernel based logging approach, because the kernel is shared between containers and host OSs. The kernel can be configured to log events using eBPF.
The kernel logging is based on hooking to kernel function calls and using eBPF programs to write out the information that is transmitted to the function. The kernel is the central unit between the operating system and the hardware, therefore, a lot of events will be output by just tracing the kernel function calls. To reduce the amount of data that is output, a filtering mechanism must be implemented.
The central part of a filtering mechanism are its filter rules. A lot of tools that are used for eBPF monitoring come with their own rule collection in a proprietary format. Having a dedicated rule format for each tool adds a lot of overhead to the maintenance effort of the monitoring tool. New insights about attackers behaviors need to be implemented for each tool separately. To overcome this issue, this project utilizes the Sigma project which delivers a standardized rule format together with an open source rule set and tools for adapting these rules for different application purposes. Sigma rules are originally developed to query SIEM solutions for security relevant events. This thesis extended the dedication of Sigma rules to the configuration of a monitoring solution.
An important part of the development of the solution was the decision for kernel or user space

filtering. While kernel space filtering would perform better, the complexity would increase to a level where the generalistic translation process of Sigma rules would fail.
The decision of user-space filtering also involved the choice of a toolchain. Tracee was chosen as it is built for Kubernetes and fulfills a lot of the required tasks. Tracees filtering is configured via Go signatures. For the adaption of the Sigma rules to other formats, the Sigma project provided the tool pySigma. The tool is primarily built for the conversion of Sigma rules to database queries, so it was necessary to built a Go template that reduces the event matching logic to a simple query. The template involved all possible combinations of log sources and fields.
With the development of the pySigma-backend-Tracee and the generated Go signatures, Tracee was equipped and recompiled into a new container image. This Container image is deployed together with Fluentbit on the Kubernetes Cluster. All events that are generated by Tracee are read by Fluentbit and sent to the OpenSearch Instance for persistence.
Functional tests where performed with using ChatGPT to generate test cases for specific Sigma rules with success. All tested Sigma rules could be triggered with their determined behavior without generating any false positive events. Further testing was performed with artificial attacks from the 5G-FORAN-ATTACK part of the 5G-FORAN research project. The security effectiveness associated with attacks that focus on Kubernetes had room for improvement, as the Sigma repository does not have specific rules for containerized environments.
The evaluated resource usage of Tracee is related to the size of the Sigma rule set and can take under heavy workloads up to 70% of the CPU utilization which is higher than acceptable. The biggest benefit of the implementation is the ease of adaption from using an easy and established rule format together with the automatic rebuild and deployment via CI/CD.
In summary the implementation was successful with room for improvements in terms of resource usage and Sigma rule set coverage.

Bibliography

[1] 3GPP. 3gpp website. https://www.3gpp.org, 2024. Accessed: 2024-12-16.

[2] O-RAN Alliance. O-ran alliance website. https://www.o-ran.org/, 2024. Accessed: 2024-12-16.

[3] CVE® Program. Common vulnerabilities and exposures (cve) database. https://www.cve.org, 2024. Provides unique identifiers for publicly disclosed cybersecurity vulnerabilities.

[4] Procyde GmbH. 5g foran project website. https://www.5g-foran.com, 2024. Accessed: 2024-12-01.

[5] BSI. Projekt 5g-foran. https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/5-G/KoPa45/5G-6G-Netzwerksicherheit-Open-RAN/TSP5-5G-FORAN/TSP5-5G-FORAN_node.html, 2024.

[6] O-RAN Alliance. *O-RAN Architecture Description 12.0*. O-RAN Alliance, 2024. Version 12.0.

[7] Stefan Köpsell, Andrey Ruzhanskiy, Andreas Hecker, Dirk Stachorra, and Norman Franchi. Open ran risk analysis. Technical report, BSI, February 2022. Version 1.2.1.

[8] O-RAN Alliance. *O-RAN Security Threat Modeling and Risk Assessment 4.0*. O-RAN Alliance, 2024. Version 4.0.

[9] RedHat. Containers vs. vms. https://www.redhat.com/en/topics/containers/containers-vs-vms, 2023. Accessed: 2024-12-01.

[10] Docker, Inc. What is a container? https://www.docker.com/resources/what-container/, n.d. Accessed: 2024-12-01.

[11] Docker, Inc. Docker overview. https://docs.docker.com/get-started/docker-overview/, n.d. Accessed: 2024-12-01.

[12] Kubernetes Project. Kubernetes concepts: Overview. https://kubernetes.io/docs/concepts/overview/, n.d. Accessed: 2024-12-01.

[13] Shubham Agarwal, Arjun Sable, Devesh Sawant, Sunil Kahalekar, and Manjesh K. Hanawal. Threat detection and response in linux endpoints. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 447–449, 2022.

[14] Bharat Manral and Gaurav Somani. Establishing forensics capabilities in the presence of superuser insider threats. *Forensic Science International: Digital Investigation*, 38:301263, 2021.

[15] Artem Dinaburg. Pitfalls of relying on ebpf for security monitoring (and some solutions). https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/. Accessed: 2025-1-27.

[16] The kernel development community. The linux kernel - bpf maps. https://docs.kernel.org/bpf/maps.html. Accessed: 2024-12-27.

[17] Open Policy Agent Community. Open policy agent - policy language. https://www.openpolicyagent.org/docs/latest/policy-language/, 2025. Accessed: 2025-01-30.

[18] Sigma Project. Sigma: Generic signature format for siem systems. https://sigmahq.io, n.d. Accessed: 2024-12-01.

[19] Sigma Project . Sigma repository. https://github.com/sigmahq/sigma, n.d. Accessed: 2024-12-01.

[20] MITRE Corporation. Mitre att&ck matrix. https://attack.mitre.org, 2024. Accessed: 2024-12-01.

[21] https://github.com/0x00000013. Huakiwi is an edr powered by ebpf and sigma. https://github.com/0x00000013/huakiwi/, n.d. Accessed: 2024-12-01.

[22] eBPF.io authors. Major ebpf applications. https://ebpf.io/applications/, 2024. Accessed: 2024-12-01.

[23] 5G-FORAN. 5g-foran research project, 2025.

[24] bpftrace Project. bpftrace - a high-level tracing language for linux. https://github.com/bpftrace/bpftrace, 2024. Accessed: 2024-12-01.

[25] Aquasecurity. Aqua tracee: Runtime ebpf threat detection engine. https://www.aquasec.com/products/tracee/, 2024. Accessed: 2024-12-01.

[26] Tracee Project. Aqua tracee. https://github.com/aquasecurity/tracee, 2024. Accessed: 2024-12-01.

[27] Aquasecurity. Tracee docs. https://aquasecurity.github.io/tracee/v0.8.3/architecture/. Accessed: 2024-12-01.

[28] pySigma SQLite Backend project Community. pysigma sqlite backend. https://github.com/SigmaHQ/pySigma-backend-sqlite, 2025. Accessed: 2025-01-28.

[29] pySigma Elasticsearch Backend project Community. pysigma elasticsearch backend. https://github.com/SigmaHQ/pySigma-backend-elasticsearch, 2025. Accessed: 2025-01-28.

[30] pySigma Project. pysigma - a python library that parses and converts sigma rules into queries. https://github.com/SigmaHQ/pySigma, 2024. Accessed: 2024-12-01.

[31] OpenSearch Project. Opensearch. https://opensearch.org, 2024. Accessed: 2024-12-01.

[32] Fluent Bit Project. fluentbit - an end to end observability pipeline. https://fluentbit.io, 2024. Accessed: 2024-12-01.

[33] ArgoCD Project. Argo cd - declarative gitops cd for kubernetes. https://argo-cd.readthedocs.io/en/stable/, 2024. Accessed: 2024-12-01.

[34] Go community. Go template package. https://pkg.go.dev/text/template, 2025. Accessed: 2025-01-30.

[35] OpenAI. Chatgpt-4o. https://openai.com, 2025. Accessed: 2025-01-28.

[36] Henrik Wittemeier. Pysigma-backend-tracee. https://git.dn.fh-koeln.de/foran/pysigma-backend-tracee.git, 2024. Accessed: 2024-12-01.

[37] SOC Prime. Soc prime website. https://socprime.com/. Accessed: 2024-12-27.

[38] Thomas Graf. Tetragon 1.0: Kubernetes security observability & runtime enforcement with ebpf. https://isovalent.com/blog/post/tetragon-release-10/. Accessed: 2024-12-27.

Appendix

A Repository

All code is available in the project Repository:
`https://git.dn.fh-koeln.de/foran/pysigma-backend-tracee`

B Measurement of eBPF activity

```
BPFTRACE_MAX_BPF_PROGS=2200 BPFTRACE_MAX_PROBES=2200 bpftrace -e \
'tracepoint:* { @["total_events"] = count(); } interval:s:100 { exit(); }'
```

C Go Template

```
package main

import (
    "fmt"
    "regexp"
    "github.com/aquasecurity/tracee/signatures/helpers"
    "github.com/aquasecurity/tracee/types/detect"
    "github.com/aquasecurity/tracee/types/trace"
    "github.com/aquasecurity/tracee/types/protocol"
)

type {title} struct {{
    cb                detect.SignatureHandler
    releaseAgentName string
}}


```

```
var {title}Metadata = detect.SignatureMetadata{{
    ID:          "{rule.id}",
    Version:     "1",
    Name:        "{rule.title}",
    EventName:   "{title}",
    Description: "{rule.description.replace("\\","\\\\").replace("\"","\\\\\\\"").replace("\n"," ")}",
    //TraceeLogSource: "{rule.logsource.category}"
    Properties: map[string]interface{{}}{{
        "Severity":             "{rule.level}",
        "Category":             "{rule.tags[0].namespace}",
        "Technique":            "{rule.tags[0].name}",
        "Kubernetes_Technique": "",
        "id":                   "{rule.id}",
        "external_id":          "{rule.id}",
    }},
}}

func (sig *{title}) Init(ctx detect.SignatureContext) error {{
    sig.cb = ctx.Callback
    return nil
}}

func (sig *{title}) GetMetadata() (detect.SignatureMetadata, error) {{
    return {title}Metadata, nil
}}

func (sig *{title}) GetSelectedEvents() ([]detect.SignatureEventSelector, error) {{
    return []detect.SignatureEventSelector{{
        {{Source: "{rule.logsource.category}", Name: "{rule.logsource.product}", Origin: "*"}},
    }}, nil
}}

func (sig *{title}) OnEvent(event protocol.Event) error {{
    eventObj, ok := event.Payload.(trace.Event)
    if !ok {{
        return fmt.Errorf("invalid event")
    }}
    var file_path, dst, src, dstIP, argv, cmdpath, dstHostname, txtAnswer, httpUserAgent, httpMethod,httpUri,httpHost, httpCookie, httpReferer
```

```
    string
56   var dstPort, httpStatusCode int
57   var err error
58   switch eventObj.EventName {{
59   case "file_modification":
60       file_path, err = helpers.GetTraceeStringArgumentByName(eventObj, "
    file_path")
61       if err != nil {{
62           return err
63       }}
64   case "net_packet_ipv4":
65       dst, err = helpers.GetTraceeStringArgumentByName(eventObj, "dst")
66       if err != nil {{
67           return err
68       }}
69       src, err = helpers.GetTraceeStringArgumentByName(eventObj, "src")
70       if err != nil {{
71           return err
72       }}
73   case "net_tcp_connect":
74       dstIP, err = helpers.GetTraceeStringArgumentByName(eventObj, "
    dstIP")
75       if err != nil {{
76           return err
77       }}
78       dstPort, err = helpers.GetTraceeIntArgumentByName(eventObj, "
    dstPort")
79       if err != nil {{
80           return err
81       }}
82   case "sched_process_exec":
83       argv_arr, err  := helpers.GetTraceeSliceStringArgumentByName(
    eventObj, "argv")
84       if err != nil {{
85           return err
86       }}
87       for _, arg := range argv_arr {{
88           argv = argv + " " + arg
89       }}
90       cmdpath, err = helpers.GetTraceeStringArgumentByName(eventObj, "
    cmdpath")
91       if err != nil {{
```

```
92              return err
93          }}
94      case "net_packet_dns":
95          dns, err := helpers.GetProtoDNSByName(eventObj, "proto_dns")
96          if err != nil {{
97              return err
98          }}
99          if len(dns.Questions) > 0{{
100             dstHostname = dns.Questions[0].Name
101         }}
102     case "net_packet_dns_response":
103         dns, err := helpers.GetProtoDNSByName(eventObj, "dns_response")
104         if err != nil {{
105             return err
106         }}
107         for i:=0;i<len(dns.Answers);i++ {{
108             if dns.Answers[i].Type == "TXT"{{
109                 for j:=0;j<len(dns.Answers[i].TXTs);j++ {{
110                     txtAnswer = txtAnswer + " " + dns.Answers[i].TXTs[j]
111                 }}
112             }}
113         }}
114     case "net_packet_http_request":
115         arg, err := helpers.GetTraceeArgumentByName(eventObj, "
    http_request", helpers.GetArgOps{{DefaultArgs: false}})
116         if err != nil {{
117             return err
118         }}
119
120         http, ok := arg.Value.(trace.ProtoHTTPRequest)
121
122         if !ok {{
123             return nil
124         }}
125         httpUserAgent = http.Headers.Get("User-Agent")
126         httpReferer = http.Headers.Get("Referer")
127         httpMethod = http.Method
128         httpUri = http.URIPath
129         httpHost = http.Host
130         httpCookie = http.Headers.Get("Cookie")
131         dstIP, err = helpers.GetTraceeStringArgumentByName(eventObj, "
    dstIP")
```

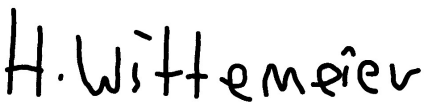
```
132
133     case "net_packet_http_response":
134         arg, err := helpers.GetTraceeArgumentByName(eventObj, "
    http_response", helpers.GetArgOps{{DefaultArgs: false}})
135         if err != nil {{
136             return err
137         }}
138
139         http, ok := arg.Value.(trace.ProtoHTTPResponse)
140
141         if !ok {{
142             return nil
143         }}
144         httpStatusCode = http.StatusCode
145     }}
146
147
148
149
150
151     if {query} {{
152         metadata, err := sig.GetMetadata()
153         if err != nil {{
154             return err
155         }}
156         sig.cb(&detect.Finding{{
157             SigMetadata: metadata,
158             Event:       event,
159             Data:        nil,
160         }})
161     }}
162     _ = dstIP
163     _ = dstPort
164     _ = file_path
165     _ = src
166     _ = dst
167     _ = argv
168     _ = cmdpath
169     _ = dstHostname
170     _ = txtAnswer
171     _ = httpUserAgent
172     _ = httpMethod
```

```
173     _ = httpUri
174     _ = httpHost
175     _ = httpStatusCode
176     _ = httpCookie
177     _ = httpReferer
178     return nil
179 }}
180 
181 func (sig *{title}) OnSignal(s detect.Signal) error {{
182     return nil
183 }}
184 func (sig *{title}) Close() {{}}
```

Code 10: Template for a Tracee Signature

Declaration

I confirm that I have written this thesis independently. I have marked all passages taken verbatim or in spirit from published or unpublished works of others or from the author himself as taken. All sources and aids that I have used for the work are indicated. The thesis has not been submitted to any other examination authority with the same content or in substantial parts.



Bonn, February 3, 2025

(Place, Date)

(Signature)