POLITECNICO DI TORINO

Master's Degree
in Computer Engineering

Master's degree thesis

Development and Optimization of a Firmware for Real-Time Monitoring of Network Traffic and Threat Detection

Supervisor
prof. Andrea Atzeni

Candidate
Gianluca Iadicicco

Academic Year 2024-2025

Ad Anna
Ai miei genitori Anna e Lucio
Alla mia famiglia e a tutti coloro che credono in me

Summary

The increase in cyber threats has made it essential to adopt advanced solutions for monitoring and securing network traffic. Among these, Intrusion Detection Systems (IDS) represent a fundamental tool for identifying and preventing suspicious activities within an IT infrastructure.

An IDS is designed to analyze network traffic or system activities with the aim of detecting anomalous or potentially harmful behavior. There are two main types of IDS: Network-based IDS (NIDS), which monitor network traffic in real time, and Host-based IDS (HIDS), which focus on analyzing operations performed on a single device or server.

Threat detection can occur according to two approaches: signature-based detection, which identifies already known attacks by comparing traffic with a database of predefined signatures, and anomaly-based detection, which analyzes network behavior in search of deviations from a normal model.

IDS can operate in passive mode, generating alerts without directly intervening, or integrate with prevention systems (IPS) to automatically block suspicious activities. These tools are essential to ensure the security of corporate networks, critical infrastructures and cloud environments, offering advanced protection against cyber attacks, malware and unauthorized access.

Among the most advanced open-source IDS, Suricata stands out for its multi-threading support, high configurability and the ability to analyze a wide range of protocols. However, this flexibility entails a high consumption of resources, making it less suitable for embedded devices or contexts in which it is necessary to monitor only a subset of the supported protocols.

The goal of this thesis is to optimize Suricata 8.0.0 to improve its efficiency on platforms with few resources, while maintaining its effectiveness in detecting threats. In particular, a system was developed that allows to make support for network protocols optional already at the compilation stage, thus reducing the software footprint and improving its performance. The approach adopted required modifications to the Suricata source code, modifying various elements such as the protocol decoders, the detection engine and the build configuration system.

The changes had a measurable impact, quantified using three metrics: RAM consumption, executable size, and log file size. Tests were conducted on different Suricata configurations running on a Virtual Machine with Ubuntu 24.10, revealing that enabling or disabling specific protocols—based on the network being analyzed—can significantly optimize the software. In particular, removing HTTP and the associated libhtp library,

a complex library specifically designed to parse and analyze HTTP traffic, drastically reduced all three metrics.

Once Suricata was optimized, its performance needed to be tested on an embedded device with limited resources. For this purpose, the project used a Raspberry Pi 3 Model B, which has an ARM architecture and only 1 GB of RAM and a relatively low-power processor. Given these constraints, OpenWrt was chosen as the reference operating system due to its open-source nature, flexibility in integrating third-party software, and low resource requirements.

OpenWrt is a Linux-based OS, mainly developed for routers and low-resource devices. Despite its lightweight nature, it offers a wide range of features explored in this thesis, including routing and relatives protocols, firewall with both iptables and nftables, VPN support, and QoS management.

When implementing Suricata on OpenWrt 23.05.5, three different compilation approaches were explored:

- Direct compilation on OpenWrt: Failed because the lack of some development libraries required by Suricata.

- Using the official OpenWrt Software Development Kit (SDK): Failed due to the SDK's limited set of libraries, causing compatibility issues during compilation.

- Creating a custom OpenWrt image with Suricata pre-integrated: This method enabled the inclusion of all necessary libraries and extensions.

Once this phase was completed and a functional OpenWrt image with Suricata was successfully built, the next step was its integration on the board. However, due to the board's limited resources, the compilation had to be performed on another machine. Additionally, because of the architectural differences between the host and the target, cross-compilation was required. This introduced several challenges, including compatibility with the toolchain and managing libraries not designed for cross-compilation.

The biggest issue that has been encountered was caused by Suricata-lua-sys, a Suricata library that was not designed for cross-compilation. As a result, when compiled, it produced an output compatible only with the host machine's architecture (x86_64), rather than the Raspberry Pi 3's architecture (aarch64).

Once a functional OpenWrt instance with Suricata was created, an additional optimization was implemented: an autoconfiguration system based on traffic analysis. This system identifies the most frequently used protocols and adapts Suricata's configuration accordingly.

Using a combination of Bash and Python scripts, a cooperative process was established between OpenWrt and the host machine. The bash script initially start Suricata that monitors network traffic, then it passes the captured packets log to the python script that detects the most common protocols, and then re-compiles and re-installs an optimized configuration.

This optimization further reduces resource consumption without compromising Suricata's threat detection capabilities, as it is specifically tailored to the analyzed network.

The final phase involves performance testing to assess the system's efficiency. These tests were conducted using both Ethernet and wireless connections, under normal conditions and during an attack.

The attack chosen to evaluate the system's performance was the DoS attack, as it is commonly used to overwhelm a system's resources. This made it an ideal choice to determine whether Suricata had issues with detection and whether it consumed excessive resources, potentially causing a system crash. To carry out this attack, two tools were used:

- tcpreplay, which replays captured network traffic logs.
- hping3, which was specifically used to perform a SYN flood attack, a type of DoS attack that exploits the TCP three-way handshake mechanism.

The evaluation was based on three key metrics: Suricata's RAM and CPU consumption, as well as the system's idle percentage, to determine the risk of a crash. As highlighted in previous tests, different Suricata configurations impact performance—the fewer protocols a configuration supports, the lower the RAM and CPU consumption, resulting in a more efficient operating system.

Finally, the comparison with another open-source IDS, Snort, confirmed the advantages of Suricata and the optimizations applied to it. Test results showed that while Snort have a more stable CPU consumption, even if slightly higher than Suricata's, its RAM consumption is significantly higher, especially during the SYN flood attack.

In conclusion, this thesis demonstrates how an advanced IDS can be adapted to work effectively in resource-limited contexts, without sacrificing detection capabilities. The implemented modifications can make Suricata suitable for areas such as the Internet of Things (IoT) or Industrial IoT (IIoT), perimeter security and distributed network infrastructures, improving cybersecurity in scenarios where lightness and efficiency are critical factors.

Acknowledgements

Desidero esprimere la mia sincera gratitudine al Professor Andrea Atzeni, relatore di questa tesi, per il costante supporto, l'interesse e i preziosi consigli ricevuti durante la stesura dell'elaborato.

Un sentito ringraziamento va anche a Saverio Milo, tutor aziendale, per la sua disponibilità, competenza e assistenza nella parte pratica dell'implementazione. La sua esperienza e i suoi suggerimenti hanno contribuito in modo significativo al raggiungimento degli obiettivi di questa tesi. Inoltre, i suoi consigli si sono rivelati preziosi anche oltre questo lavoro, aiutandomi a chiarire le idee e a orientare le mie scelte per il futuro professionale.

Ad entrambi va la mia gratitudine per aver reso possibile portare a termine questo percorso con entusiasmo e determinazione.

Desidero dedicare un sentito ringraziamento ai miei genitori, Anna e Lucio, che con il loro sostegno, il loro amore e i loro sacrifici mi hanno permesso di arrivare fino a qui. Senza il vostro aiuto, sia morale che economico, questo traguardo sarebbe stato solo un sogno lontano.

Un ringraziamento speciale va a mia mamma, che non ha mai smesso di credere in me, anche nei momenti di difficoltà. Grazie per la tua forza, la tua perseveranza e per essere stata sempre al mio fianco, spronandomi a dare il meglio di me. Questo risultato è anche il tuo.

Un grande grazie a mio papà per la sua costante presenza, il suo supporto e il suo spirito sempre positivo. Il tuo modo di affrontare la vita con ottimismo è stato per me un grande esempio.

Grazie anche a mio fratello Emanuele e a Francesca per il loro affetto e il loro supporto, e a mio fratello Mattia, che tra una presa in giro e l'altra, riesce comunque a strapparmi un sorriso. E poi c'è Beatrice, la mia nipotina arrivata da un anno per scombussolare tutti gli equilibri di famiglia e farmi guadagnare il titolo di zio per la prima volta, con suoi sorrisi è stata capace di migliorare le brutte giornate.

Un grazie di cuore a mio cugino Luca, che è stato una presenza costante in questo percorso, sempre pronto a sostenermi e a darmi preziosi consigli. Il tuo supporto ha fatto la differenza.

Un doveroso ringraziamento a "Riunione Bilderberg", il gruppo di menti eccelse che ha seguito e sostenuto questo mio percorso, spesso a distanza ma sempre con lo spirito giusto. Grazie per i consigli, le risate, il supporto e per aver dimostrato che l'amicizia non ha bisogno di riunioni segrete... o forse sì. Senza di voi, questi cinque anni sarebbero stati molto meno divertenti (e probabilmente molto più produttivi, ma chi vuole davvero quello?).

Inoltre, un applauso a "Gli Incredibili Tre", il trio accademico più caotico di sempre, composto da me e le mie due cugine, che nonostante la distanza abbiamo condiviso ogni passo (e ogni crollo emotivo) dei nostri percorsi universitari. Grazie per aver ascoltato le mie crisi, per averne avute di vostre da condividere con me e per avermi dimostrato che il panico da esami è molto più sopportabile quando si è in tre a disperarsi. Se siamo riusciti a superare questo, possiamo affrontare qualsiasi cosa... o almeno così ci piace credere.

Un enorme grazie alla mia famiglia — nonni, zii e cugini — per il sostegno, la pazienza e per aver sempre trovato il modo di farmi sorridere, anche nei momenti più complessi. La vostra presenza è stata il miglior promemoria che non si cresce mai da soli.

Un grazie va anche a Gennaro, che oltre ad aver cresciuto una persona speciale, ha sopportato anche me. E grazie anche per gli ottimi consigli dati lungo il percorso.

Infine, un grazie speciale ad Anna, la mia roccia e il mio porto sicuro. Grazie per esserci sempre stata, per aver creduto in me anche quando io stesso non lo facevo, per la tua forza che mi ha sostenuto e per il tuo amore che mi ha dato energia nei momenti più difficili. Sei stata la mia più grande tifosa, la mia motivatrice e, quando serviva, anche la mia dolce rompiscatole. Senza di te, questo percorso sarebbe stato molto più difficile (e sicuramente meno bello). Questo traguardo è anche tuo, perché ogni passo l'abbiamo fatto insieme. Tamu.

Vi voglio bene. (anche se non lo dico spesso, quindi segnatevi questa data)

Contents

List of Tables

List of Figures

Passwords are like underwear: don't share them, don't leave them lying around, and change them often.

[Chris Pirillo, About password choices]

Part I

IDS: Concepts and Fundamentals

Chapter 1

Introduction to IDS

Over the past few decades, the growing dependence on computer networks has transformed the way individuals, organizations, and governments operate. While this technological evolution has led to greater efficiency and connectivity, it has also exposed digital infrastructures to a wide range of cyber threats, from targeted attacks to malware to unauthorized intrusions.

In this context, Intrusion Detection Systems (IDS) have emerged as essential tools to ensure the security of networks and computer systems[13]. IDS are designed to monitor network traffic or system activities in order to identify anomalous behavior or potential security breaches. They not only detect threats but also provide many valuable information that allows network administrators to intervene promptly and mitigate risks.

The importance of IDS is accentuated by the continuous evolution of attack techniques used by cyber criminals, which are becoming more and more sophisticated and difficult to detect with traditional tools. Therefore, the development and optimization of modern IDS solutions, such as the one addressed in this thesis, represent a crucial step towards a more robust and proactive cybersecurity.

In this chapter, we will introduce the basic concepts of IDS, analyzing the way they work, purposes, and role in cybersecurity. We will also explore the main types of IDS, highlighting the advantages and limitations of each.

1.1 Characteristics and purposes

Intrusion Detection Systems (IDS) are tools designed to monitor and analyze network traffic and system activity in real time, with the aim of identifying intrusion attempts, suspicious behavior, or violations of security policies. In other words, an IDS acts as a cybersecurity alarm, alerting administrators of potential threats.

An IDS differs from other security tools, such as firewalls, in its passive approach to detection: while a firewall actively blocks unauthorized traffic, an IDS simply observes, records, and reports anomalies or malicious activity. The primary function of an IDS is not to prevent attacks, but to detect them, allowing operators to act quickly to mitigate the damage.

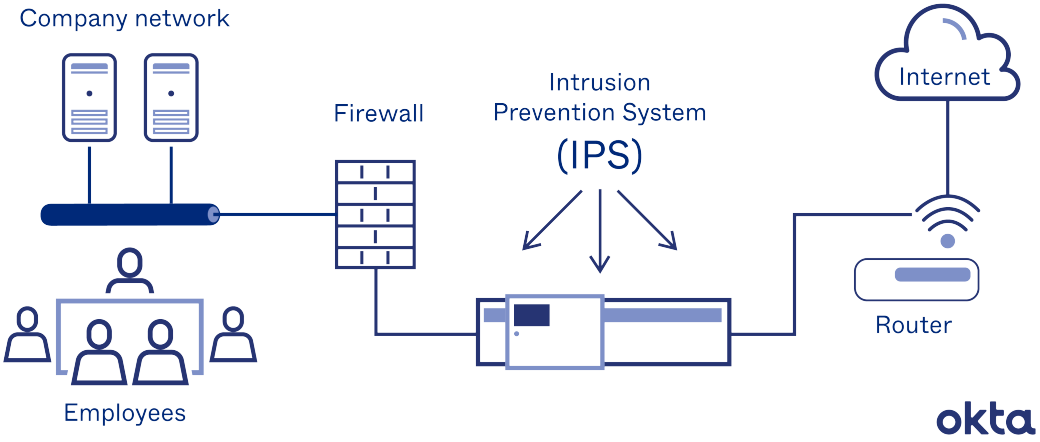

Figure 1.1. Schema of an IDPS.

The main purpose of an IDS can be divided into three basic objectives:

1. **Malicious activity detection**
   An IDS can identify behaviors that indicate possible attacks, such as unauthorized access attempts, malware, or lateral movement of an intruder within a compromised network.

2. **Proactive protection**
   Although they do not directly prevent attacks, IDS help strengthen security by providing detailed information about suspicious events[1]. This data allows administrators to improve the configuration of other defense systems, such as firewalls and antivirus, and take appropriate countermeasures.

3. **Regulatory compliance** Many cybersecurity regulations, such as GDPR[24][16] or the NIST Cybersecurity Framework, require organizations to implement monitoring and analysis systems to detect any breaches. IDS comply with these requirements, generating logs and reports useful for audits and verification.

IDS, therefore, not only represent an essential component of modern cybersecurity infrastructures, but also integrate into broader strategies, such as those based on the concept of "Defense in Depth" (layered defense), where different layers of protection work together to safeguard digital assets. Their effectiveness depends on the ability to accurately detect threats, minimizing false positives and false negatives, and on their integration with other security tools.

1.2 Role in Cybersecurity

Intrusion Detection Systems (IDS) play a crucial role in the cybersecurity ecosystem, representing an essential tool to protect networks and computer systems from increasingly sophisticated attacks. In a context where cyber threats continue to evolve, IDS provide an indispensable line of defense, complementing other security solutions such as firewalls, antivirus and intrusion prevention systems (IPS).
Let's analyze the contribution of IDS to cybersecurity from different perspectives.

1.2.1 Continuous monitoring of activities

IDS operate continuously, analyzing network traffic or system logs to identify anomalies, security policy violations, or unauthorized access attempts. This constant surveillance capability allows organizations to have a real-time view of potential threats, reducing the time to detect and respond.

1.2.2 Advanced threat detection

Unlike more static tools like firewalls, which often simply block traffic based on predefined rules, IDS are designed to detect complex and unpredictable attack patterns. Using signature-based detection techniques or behavioral analysis, they can identify threats such as:

- Advanced malware (es. ransomware, spyware).
- Network attach (es. DDoS, ports scanning).
- Actions of intruders within the network.

1.2.3 Incident Response Support

An IDS does more than just report anomalies, it provides detailed information about the suspicious activity detected. This data is critical for security analysts, who can use it to respond quickly to incidents and prevent further damage. IDS also facilitate the creation of logs and reports useful for post-incident investigations and security audits.

1.2.4 Integration into multi-layered defense strategies

IDS fit into the cybersecurity model known as "Defense in Depth", where multiple layers of protection work together to address threats on multiple fronts. In the context of a corporate network, for example, an IDS can integrate firewalls, VPNs, and authentication systems, contributing to a holistic defense strategy.

1.2.5 Adaptation to specific scenarios

IDS are particularly useful in contexts where security must be customized to specific needs. For example:

- In IoT environments, an IDS can monitor resource-constrained devices to detect anomalies.
- In cloud systems, IDS support distributed monitoring and analysis of traffic between nodes.
- In industrial environments, IDS can detect suspicious activity on SCADA systems or critical networks.

1.2.6 Limitations

Despite their importance, IDS are not without limits. Managing false positives (alerts generated by legitimate activity) is a major challenge, as it can overwhelm administrators and reduce the effectiveness of the system. Additionally, an IDS is typically a passive system, which does not block detected threats unless combined with an IPS.

In short, the role of IDS in cybersecurity is to provide an additional layer of monitoring and detection, helping to build a strong and resilient defense against modern threats. Their ability to adapt to different scenarios and identify complex threats makes them an essential element in any organization's security strategies.

1.3 Typologies

Intrusion Detection Systems are divided into different types[6] based on the method of detecting intrusions, their location in the network, and the mode of analysis. Each type has specific advantages and disadvantages, and the choice depends on the specific security needs of a network or organization.
The main types of IDS are:

1.3.1 Based on the position in the network

1. Network-based IDS (NIDS)
   Network-based IDS monitor traffic on a network and analyze data packets in transit. These systems are typically placed at strategic points in the network, such as gateways, routers, or switches, to monitor all traffic entering or leaving the network. NIDS are very effective at detecting attacks that target the network, such as port scans, denial of service (DoS), and anomalous traffic. However, they may have limitations in monitoring encrypted traffic (for example, HTTPS).

2. Host-based IDS (HIDS)
   Host-based IDS monitor the internal traffic of a single device (host), such as a server, computer, or other network device. HIDS focus on the host's behavior, monitoring log files, system activity, and running processes. They can detect suspicious changes to system files, malware, and abnormal behavior at the individual device level. However, their monitoring scope is limited to the host, making them less effective at detecting complex network attacks.

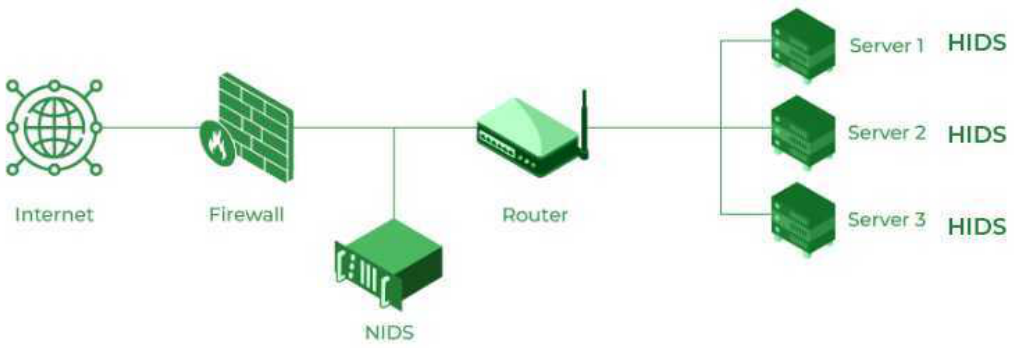

Figure 1.2. Graphic difference between a NIDS and a HIDS.

1.3.2 Based on the detection method

1. Signature-based IDS (SIDS)
   Signature-based IDS detect intrusions by comparing network traffic or system activity to a database of known attack signatures or patterns. Each signature represents a behavior or sequence of events that corresponds to a known attack. This type of system is very effective at detecting known and well-documented attacks, but it is not able to detect new or unknown threats (zero-day attacks). SIDS are used to detect common attacks such as viruses, worms, and Trojans.

2. Anomaly-based IDS (AIDS)
   Anomaly-based IDS rely on the analysis of normal network or host behavior to detect suspicious activity. Initially, a model of "normal behavior" is built, and any significant deviation from this model is flagged as a possible intrusion. This type of IDS can detect new and unknown attacks, such as advanced malware or hacking attempts, but can be more susceptible to false positives, as some legitimate variations in traffic can be interpreted as threats.

3. Hybrid-based IDS
   Hybrid IDS combine the features of signature-based and anomaly-based systems. This approach seeks to leverage the strengths of both methods, providing more accurate detection for both known and unknown threats. Hybrid IDS are particularly effective at balancing reliability and detection capacity, reducing false positives without sacrificing the ability to detect new threats.

1.3.3 Based on the analysis mode

1. Active IDS (Intrusion Prevention Systems - IPS)
   An active IDS, or more precisely an Intrusion Prevention System (IPS), not only detects intrusions, but also takes immediate action to prevent the attack. If an attack is detected, the IPS can block suspicious traffic, terminate compromised connections, or even isolate vulnerable systems. This type of system is more proactive than passive IDS, which simply report the threat without taking direct action. One potential problem with this type of system is false positives.

2. Passive IDS
   Passive IDS simply monitor traffic or activity and generate alerts or logs when a potential intrusion is detected. They do not block attacks directly, but provide crucial information for timely response by system administrators. Their strength lies in the accuracy of detection and the ability to analyze and record detailed attack information, which can be used to improve overall security.

1.3.4 Based on architecture

1. Centralized IDS
   A centralized IDS relies on a single processing point where all information is collected and analyzed. This configuration is particularly advantageous in simpler environments, such as small or medium-sized networks. By concentrating operations in a single node, it is easier to manage and configure the system, as well as to keep the signature database up to date.

   However, this architecture has some inherent limitations. The presence of a central point represents a potential bottleneck, as all information must be processed in the same place, with the risk of overload in high-traffic networks. Furthermore, the central node becomes a single point of failure: an attack or malfunction could render the entire system unusable.

   Centralized IDS are a practical solution for situations where the amount of data to be analyzed is relatively limited and a highly distributed coverage is not required. For example, small offices or local area networks (LAN) can benefit from the ease of implementation and maintenance of a centralized system.

2. Distributed IDS (DIDS)
   Distributed IDS rely on a network of nodes – often referred to as sensors or agents – that work cooperatively to monitor different segments of a network or system. This architecture is ideal for large networks or complex infrastructures, where a centralized approach would not be scalable or robust enough.

   Each sensor can perform a preliminary analysis of local activities and, if necessary, communicate with other nodes or a central analysis engine for further processing. This decentralized model allows for greater resilience: even if one node is attacked or fails, the others can continue to operate, ensuring monitoring continuity.

   On the other hand, distributed IDS require more complex management. Synchronizing and configuring nodes can be a challenging task, and data transmission between sensors can introduce latencies, especially in environments with high traffic volumes.

   This architecture is particularly suited to contexts such as large enterprises, distributed geographic networks and cloud environments, where it is essential to manage large amounts of data from numerous sources, while maintaining a high threat detection capacity.

Centralized IDS vs. Distributed IDS Comparison

A comparison of the two approaches highlights that the choice depends mainly on the size of the network, the traffic load and the security requirements:

	Centralized IDS	**Distributed IDS**
Ease of management	High	Low
Scalability	Limited	High
Risk of failure	High (single point of failure)	Low (redundancy)
Performance on large networks	Poor	Excellent
Adaptability to modern threats	Limited	High

Table 1.1. Centralized vs. Distributed IDS Comparison

Chapter 2

Architecture and operating principles of IDS

To fully understand how Intrusion Detection Systems work and their role in cybersecurity, it is essential to analyze their architecture and main components. IDS are designed to monitor, detect and report suspicious activity in a system or network, and they do so thanks to an architecture composed of distinct elements, each of which contributes to the process of analyzing and reporting threats.

In this chapter, we will explore the main components that make up an IDS, from the data sources to the analysis engine, up to the alert system that notifies any anomalies or attacks detected.

This overview is essential to understand not only how an IDS performs its functions, but also how it can be integrated into network infrastructures of varying complexity, adapting to specific security needs.

2.1 Main components

Intrusion Detection Systems are made up of a series of essential components that work together to monitor and analyze network or system activity, detecting any anomalous or dangerous behavior. Each component plays a specific role in the detection process, allowing IDS to operate efficiently and accurately.

2.1.1 Data sources

Data sources are the first level of input for an IDS. They are responsible for collecting the information needed to identify potential intrusions. The main data sources include:

- Network traffic
  Network-based IDS analyze data packets in transit on a network. This data can be collected by traffic mirroring from switches or routers.

- Systems log
  Host-based IDS monitor logs generated by operating systems and applications, looking for suspicious events such as unauthorized access attempts or changes to configuration files.

- Audit files
  Some IDS use information from audit files, which contain a detailed log of system activity, useful for identifying unauthorized changes.

These sources provide raw data that the system must process, and their quality and completeness directly influence the detection capabilities of an IDS.

2.1.2 Analysis Engine

The analysis engine is the operational heart of an IDS. It is responsible for processing the collected data, analyzing it and comparing it with predefined models (signature-based analysis) or normal behaviors (anomaly-based analysis).

The analysis engine must be able to handle large volumes of data in real time, maintaining a balance between accuracy and processing speed.

2.1.3 Alert system

The alert system is the interface between the IDS and network or system administrators. When the analysis engine identifies a potential attack, the alert system takes care of notifying the event. Alerts can take different forms:

- Internal Logs: Detailed event logging within the logging system.
- Visual Notifications: Display alerts on dashboards or GUI.
- Real-time Messaging: Send emails, push notifications or SMS messages to network administrators.

A good alert system must be configurable, to reduce false positives and ensure that only critical events are reported immediately.

Chapter 3

Challenges and usage scenarios

Intrusion Detection Systems are fundamental tools for ensuring the security of computer networks, but their use is accompanied by a series of technical and operational challenges. The effectiveness of an IDS depends not only on its ability to detect malicious activities, but also on its accuracy, scalability and ability to adapt to increasingly diverse and complex technological contexts.

In this chapter, we will analyze some of the main issues related to the operation of IDS, such as the balance between performance and accuracy, and their adaptation to continuously evolving threats. Next, we will explore the application scenarios of IDS, focusing on specific environments such as corporate networks, cloud computing and the Internet of Things (IoT), where security takes on different characteristics and priorities. Finally, we will discuss the emerging technologies that are shaping the future of IDS, including the introduction of artificial intelligence algorithms and advanced machine learning techniques.

The goal of this chapter is to provide an overview of the challenges that IDS face to remain effective and relevant in a rapidly evolving technology and threat landscape, while highlighting the opportunities that innovation offers.

3.1 Challenges in Accuracy and Adaptation

IDS play a crucial role in detecting suspicious or malicious activity, but their operation is influenced by a number of critical factors that determine their effectiveness and usefulness in real-world operating environments. These include the balance between performance and accuracy, the management of false positives and negatives, and the ability to quickly adapt to new threats.

3.1.1 Performance: The trade-off between speed and accuracy

Real-time network monitoring requires extremely fast processing of large volumes of data. IDS must analyze network packets, system logs, and other inputs in fractions of a second to detect anomalies or suspicious behavior.

A high-performance IDS can process massive amounts of data without introducing significant delays (latency) or compromising the quality of detection. However, increasing speed can reduce the accuracy of the system[3], potentially leaving more sophisticated malicious activity undetected. This trade-off is one of the main challenges in IDS development and optimization.

3.1.2 False Positives and False Negatives: The Accuracy Problem

The accuracy of an IDS is measured by its ability to distinguish between legitimate activity and malicious behavior. However, errors are inevitable:

- False positives: Occur when the IDS flags legitimate activity as malicious. These events can generate unnecessary alarms, increasing the workload for security operators and reducing confidence in the system.
- False negatives: Occur when the IDS fails to detect a real threat. These cases pose a serious security risk, as they leave attacks undetected and unresolved.

Balancing the ratio of false positives to false negatives is a complex challenge, requiring accurate and constantly updated detection models. Machine learning and artificial intelligence techniques are emerging as promising solutions to reduce these types of errors.

3.1.3 The importance of continuous updating

The cyber threat landscape is constantly evolving, with increasingly sophisticated attacks exploiting zero-day vulnerabilities and advanced obfuscation techniques. Traditional IDS, based on static rules or signatures, may be ineffective against these new or unknown threats.

To address this challenge, IDS must:

1. Be frequently updated with new signatures and rules to identify emerging threats.
2. Incorporate behavioral techniques, which analyze anomalous traffic patterns rather than relying solely on predefined signatures.
3. Integrate machine learning algorithms to detect unconventional patterns and autonomously adapt to new attack scenarios.

Adaptability is especially important in dynamic environments such as cloud computing and IoT, where network parameters and attack vectors can change rapidly.

3.2 IDS in Companies, Cloud and IoT Networks

Intrusion Detection Systems are essential to many technology scenarios, but their use varies greatly depending on the environment in which they are deployed. This requires adapting IDS technologies to the specific needs of enterprise networks, cloud infrastructures, and IoT devices, each of which presents unique security challenges.

3.2.1 IDS in Corporate Networks: Protection at Scale

Corporate networks are complex environments, characterized by a variety of devices and intense and continuous network traffic. In this context, IDS are used to:

- Monitor internal and external traffic.
- Identify suspicious activity, such as lateral movement of an attacker within the network.
- Detect advanced attacks, such as Advanced Persistent Threats (APT).

IDS for enterprise networks must be scalable and capable of analyzing large volumes of data in real time. In addition, it is essential to integrate the IDS with other security tools, such as Security Information and Event Management (SIEM), to correlate events and obtain a complete view of threats.

Another critical aspect is the use of advanced behavioral analysis techniques, which allow to detect anomalies in traffic and report unusual activity that could indicate a security breach.

3.2.2 Cloud IDS: Visibility and Scalability Challenges

Cloud computing introduces new dynamics for IDS, especially due to the distributed and virtualized nature of resources. In this environment, the main challenges include:

- The lack of direct access to the physical infrastructure.
- The need to monitor traffic distributed across multiple regions or cloud providers.
- The protection of shared resources in multi-tenancy environments.

Cloud IDS must be designed to monitor not only network traffic, but also application-level events, such as unauthorized access or suspicious configuration changes. In addition, automation plays a crucial role: many cloud-native security solutions use automated tools for incident detection and response.

A practical example is the use of IDS based on lightweight software agents, which can be installed directly on virtual machines or containers, providing deep visibility without compromising performance.

3.2.3 IDS in IoT: Protecting a Heterogeneous Ecosystem

The Internet of Things (IoT) has transformed the cybersecurity landscape, connecting devices ranging from household appliances to industrial systems. However, the wide variety of protocols and standards used, combined with the limited processing power of many IoT devices, makes their protection particularly challenging.

In this context, IDS must be designed to operate in heterogeneous networks, where devices with very different processing capabilities coexist. Lightweight solutions, based on traffic behavior analysis, are often the most practical approach to detect anomalies.

A distinctive aspect of IDS for IoT is the need for a distributed architecture, where IoT gateways play a central role in monitoring and aggregating traffic. These systems not only help reduce the load on IoT devices, but also enable the task of in-depth analysis to be transferred to more powerful and sophisticated central IDS systems.

Securing IoT networks also requires constant attention to signature updates and vulnerability management, as many devices are not designed to receive regular updates, increasing the risk of targeted attacks.

3.2.4 Integration and synergies between different environments

The convergence of enterprise networks, cloud and IoT requires IDS solutions that can operate effectively in hybrid and complex environments. The adoption of distributed IDS, where the various modules work together to analyze traffic and share information, represents a step forward towards the complete protection of increasingly integrated digital ecosystems.

Emerging technologies, such as machine learning, are further enhancing the capabilities of IDS, allowing them to quickly adapt to changes in traffic behavior and detect new threats, regardless of the environment in which they operate.

3.3 SIEM: Integration and Cooperation with IDS

Security Information and Event Management (SIEM) is an essential component of modern security infrastructures[9]. By integrating data from multiple sources, including IDS, SIEM provide a centralized, comprehensive view of threats and anomalies in the network. In this section, we explore the key features that an IDS must have to work effectively with a SIEM.

3.3.1 Requirements for Integration

To ensure optimal integration between an IDS and a SIEM, some technical and functional requirements must be met:

- **Standardized Output Formats:** IDSs should support common log formats to make data processing and analysis easier for the SIEM, such as:

1. JSON (JavaScript Object Notation): A structured, machine- and human-readable text format that is ideal for representing hierarchical or complex data due to its object and array structure.

   Example:

   ```
   {
    "timestamp": "2024-12-27T10:00:00Z",
    "event": "login_attempt",
    "status": "success",
    "user": "gian"
   }
   ```

2. CSV (Comma-Separated Values): A simple format for representing tabular data, where each row represents a record and fields are separated by commas (or other delimiters).

   Example:

   ```
   timestamp,event,status,user
   2024-12-27T10:00:00Z,login_attempt,success,gian
   ```

3. Syslog: A standard protocol for sending system log messages, widely used to centralize and analyze events from network devices and applications.

   Example:

   ```
   <34>1 2024-12-27T10:00:00Z mymachine app - ID47
   [exampleSDID@32473] event="login_attempt"
   status="success" user="gian"
   ```

   with the following schema:

   PRIORITY VERSION TIMESTAMP HOST-NAME APP-NAME PID MSGID STRUCTURED-DATA MSG

- **Generating Relevant Events:** It is essential that the IDS filters and classifies events to avoid flooding the SIEM with meaningless data.
- **Real-Time Integration:** Support for protocols such as Kafka allows to send events to the SIEM in real time, ensuring a timely response to threats.

3.3.2 Benefits of IDS and SIEM Cooperation

The synergy between IDS and SIEM offers numerous advantages, including:

1. **Event Correlation:** SIEM can combine IDS events with data from firewalls, endpoints, and other sources to identify complex attack patterns.

2. **Noise Reduction:** The IDS can be configured to detect only high-impact events, while the SIEM takes care of the in-depth analysis.

3. **Regulatory Compliance:** Through integration, SIEM make it easy to generate reports that comply with regulations such as GDPR or ISO 27001.

3.3.3 Challenges and Future Opportunities

Despite the benefits, there are some challenges in integrating IDS and SIEM:

- **Scalability:** In large network environments, the amount of data generated by the IDS can overload the SIEM.
- **Event Accuracy:** The generation of false positives by the IDS can negatively impact the effectiveness of the SIEM.
- **Interoperability:** Not all IDS offer native support for popular SIEM, requiring custom solutions.

In the future, the adoption of open standards and the use of artificial intelligence to analyze IDS data could further improve the effectiveness of SIEM integration.

3.4 Emerging and currently used technologies

The evolution of Intrusion Detection Systems (IDS) is closely linked to technological innovation and the need to respond to increasingly sophisticated cyber threats. Current IDS technologies represent the state of the art for intrusion detection, but emerging approaches are rapidly transforming the cybersecurity landscape, providing new opportunities to improve the effectiveness and efficiency of these systems.

3.4.1 Current Technologies

Traditional IDS technologies rely on two foundational methodologies: signature-based detection and anomaly-based detection, as previously described. However, modern IDS solutions have evolved to incorporate event correlation techniques, which enhance the contextual understanding of network activity. By aggregating and analyzing information from multiple data sources, these techniques enable IDS systems to identify patterns and relationships that may not be evident in isolated events. This not only improves the precision of detections but also helps to significantly reduce false positives, offering a clearer and more actionable view of potential threats.

3.4.2 Emerging Technologies

Emerging technologies are revolutionizing the way IDS operate, making them more intelligent, adaptable, and capable of dealing with sophisticated threats.

- **Machine Learning:**
  The use of machine learning is becoming increasingly common in IDS[17], due to its ability to analyze large amounts of data and identify complex patterns. Supervised learning models can be trained with labeled data to detect specific attacks, while unsupervised models can discover anomalies without the need for predefined signatures. This technology enables IDS to evolve and adapt to new threats proactively.

- **Artificial Intelligence (AI):**
  AI is an extension of machine learning, using advanced techniques such as deep neural networks to improve detection capabilities. AI-based IDS can process data in real time, providing accurate detection even in complex and dynamic environments. AI can also be used to automate incident response, reducing response times and improving the effectiveness of countermeasures.

- **Distributed and decentralized detection:**
  With the rise of cloud and IoT networks, IDS are adopting distributed architectures that allow them to monitor traffic locally, reducing the processing load on central systems. Technologies such as blockchain are being explored to ensure the integrity and transparency of collected data, creating a more reliable detection infrastructure.

- **Contextual and adaptive detection:**
  Next-generation IDS are moving toward contextual approaches that consider not only network traffic, but also user behavior and system conditions. For example, an IDS could increase detection sensitivity during off-hours, when anomalous activity is easier to identify.

3.4.3 Synergies between existing and emerging technologies

Emerging technologies do not necessarily replace those currently in use, but rather complement them, creating more powerful and versatile solutions. For example, machine learning can be integrated with signature-based detection to improve accuracy and reduce false positives. Similarly, AI-based automation can support human analysts in managing reports, allowing them to focus on complex threats.

This combination of techniques allows IDS to adapt to an ever-changing threat landscape, ensuring effective protection against both traditional attacks and advanced threats.

Part II

Suricata: Architecture, Applications, and Challenges

Chapter 4

An Introduction to Suricata

In recent years, Network Intrusion Detection Systems (NIDS) have assumed a crucial role in protecting computer networks, allowing to monitor traffic and detect potential threats in real time. Among the different tools available, Suricata stands out as one of the most advanced and versatile open-source NIDS, thanks to its modern architecture and its ability to support a wide range of network protocols.

The choice to adopt Suricata for the project described in this thesis arises from the combination of several factors. First, Suricata is a mature open-source project and actively supported by the community, ensuring regular updates and excellent documentation. Furthermore, its scalable architecture, designed to exploit multi-threaded hardware, makes it particularly suitable for complex network scenarios and high traffic volumes.

Compared to the available alternatives, Suricata offers a superior level of modernity and flexibility. Although tools like Zeek or Snort have found widespread use, each has limitations: Zeek, while excellent at forensic analysis, is not designed as a pure IDS, while Snort, despite its long history, is less efficient and versatile than Suricata. The latter has therefore proven to be an ideal choice, thanks to its ability to combine high performance, configuration flexibility and support for modern protocols.

In this chapter, we will analyze the main features of Suricata, focusing on its internal functioning, practical applications in enterprise and large-scale contexts, and a critical assessment of its strengths and limitations. This will allow us to better understand the reason for its choice and lay the foundations for its integration with lightweight environments, such as OpenWrt, described in the following chapters.

4.1 What is Suricata and why is it important?

Suricata is an advanced network traffic analysis engine designed to provide intrusion detection (IDS), intrusion prevention (IPS) and deep traffic monitoring (NSM) capabilities. Developed by the Open Information Security Foundation (OISF), Suricata is an open-source project known for its flexibility, scalability and ability to adapt to a wide range of network scenarios.

Suricata's distinguishing feature is its modern architecture, designed to meet the demands of today's increasingly complex and high-performance networks. With native support for protocols such as HTTP, DNS, TLS and many others, Suricata not only detects threats but also provides a detailed understanding of the analyzed traffic. This makes it a key tool for network security, capable of addressing the increasingly sophisticated challenges posed by emerging attacks and vulnerabilities.

Unlike many traditional solutions that process traffic sequentially, Suricata takes full advantage of the parallel processing capabilities of modern multi-core CPU. This approach ensures superior performance, especially in high-traffic environments such as large enterprise networks or cloud infrastructures.

In addition, Suricata stands out for its ability to provide detailed information on network flows, through advanced logging and analysis capabilities. This provides a complete view of network activity, which goes beyond simple attack detection, allowing network administrators to identify trends, anomalies and potential weaknesses in the configuration.

Suricata is also particularly important in the context of modern networks because of its open-source nature, which not only ensures accessibility, but also fosters continuous innovation through the contribution of the global community of developers and researchers. This collaborative approach ensures that the software stays ahead of evolving threats, offering regular updates and new features to meet emerging needs.

Thanks to these features, Suricata represents a natural choice for those looking for a flexible, high-performance and continuously improvable monitoring and detection solution, making it a point of reference in the field of cybersecurity.

4.2 Comparison with other NIDS

In the landscape of intrusion detection tools, several solutions have established themselves, offering different approaches and features. Among the most popular open-source alternatives are Snort and Zeek (formerly known as Bro), while in the field of proprietary solutions, Cisco Secure IPS is one of the main players. As shown by Waleed, Jamali, and Masood [25] or by Shah and Issac [23] in their articles, while sharing the goal of protecting networks by monitoring and analyzing traffic, Suricata stands out for its advanced technical features and the flexibility guaranteed by its open-source nature.

4.2.1 Snort vs Suricata

Snort is a well-established and widely used open-source IDS. However, it has some limitations compared to Suricata:

- **Performance:** Snort, up until version 2.9, used a single-threaded architecture, which was a bottleneck in high-traffic environments. Starting with version 3.0, Snort introduced multi-threading support, significantly improving performance[2]. In comparison, Suricata was designed with a native multi-threaded architecture from the start, effectively leveraging multi-core CPU to deliver superior performance in complex network scenarios[11].
- **Modern Protocol Support:** Suricata offers native support for complex protocols (e.g. HTTP, DNS, TLS) and advanced features such as network flow monitoring, while Snort focuses primarily on rule-based inspection.
- **Additional Features:** Suricata includes Network Security Monitoring (NSM) capabilities, making it more versatile than Snort, which is designed primarily as a traditional IDS.

4.2.2 Zeek (Bro) vs Suricata

Zeek is well-regarded for its behavioral analysis and forensic capabilities, but it is not a pure IDS[22] like Suricata. This distinction arises from fundamental differences in their design goals, architecture, and use cases:

- **Purpose:** Suricata is explicitly designed for real-time intrusion detection and prevention, leveraging signature-based rules to immediately identify and mitigate threats. Zeek, on the other hand, acts primarily as a network monitoring and analysis tool, focusing on the collection of detailed data about network activity for post-event investigation and analysis. This difference means that Zeek is less effective in scenarios where immediate threat detection and response are critical.
- **Threat Detection:** Suricata relies on a robust set of pre-defined rules for signature-based detection, allowing it to recognize known threats in real time. Zeek, however, uses custom scripts written in its domain-specific language to perform behavioral analysis. While this provides flexibility and makes Zeek powerful for identifying

unknown or emerging threats. Furthermore, Zeek's focus on behavioral patterns may result in a higher rate of false positives, especially if the scripts are not finely tuned to the specific environment.

- **Real-time Mitigation:** Unlike Suricata, which can actively block malicious traffic when configured as an Intrusion Prevention System (IPS), Zeek lacks native capabilities for real-time traffic blocking. This limitation means that Zeek is better suited for forensic analysis and monitoring rather than active defense.

4.2.3 Cisco Secure IPS vs Suricata

Cisco Secure IPS is a proprietary solution integrated into the Cisco Firepower platform, used primarily in enterprise and government environments. Compared to Suricata, it has distinctive features:

- **Integrated Architecture:** Cisco Secure IPS is part of a broader security ecosystem that includes firewall, VPN, and centralized management, providing complete control over the network. Suricata, being open source, must be manually integrated with other tools to replicate a similar environment.
- **Automation and AI:** Cisco Secure IPS uses AI algorithms to automatically identify new and emerging threats. Suricata, while powerful, relies primarily on manual rules and configuration, with limited support for machine learning technologies.
- **Cost:** One of the main differences is the licensing model. Cisco Secure IPS requires a significant investment, with recurring costs for licensing and upgrades. Suricata, being open source, is free, although it does require resources to configure and maintain.
- **Flexibility:** Suricata is best suited for advanced users who need customization and integration into complex environments. Cisco Secure IPS, on the other hand, provides a pre-configured, easy-to-manage package, but is less flexible for custom scenarios.

4.2.4 Conclusions and use cases

The choice between these solutions depends on the context and specific needs:

- **Suricata** excels in high-traffic environments and scenarios that require flexibility and customization, especially for those looking for a powerful, open-source solution.
- **Snort** is suitable for less complex environments, where simplicity and reliability are priorities.
- **Zeek** is ideal for forensic analysis and behavioral monitoring, rather than real-time intrusion detection.
- **Cisco Secure IPS** is suitable for large enterprises looking for an integrated, automated solution with dedicated technical support.

With its combination of high performance, multi-protocol support and flexibility, Suricata represents a modern and competitive choice in the NIDS landscape.

Chapter 5

Suricata Architecture

To fully understand the capabilities of Suricata, it is essential to analyze its internal architecture, which is the heart of its operation and determines its distinctive capabilities. Suricata is designed with a modular and highly parallel architecture, which allows it to process large volumes of network traffic in real time while maintaining high accuracy.

This chapter will explore the main components that make up Suricata, including:

- **Packet Processing**, responsible for capturing and analyzing network traffic.
- **Detection Engine**, the detection engine that applies rules and signatures to identify potential threats.
- **Logging and Reporting**, the recording and reporting systems that provide detailed visibility into detected events.

The goal of this chapter is to provide a clear and detailed understanding of how these elements work together to transform Suricata into a state-of-the-art monitoring and intrusion detection tool. Through this analysis, it will be possible to highlight the strengths and challenges of its implementation, preparing us for subsequent discussions on the optimizations and customizations needed for specific scenarios.

5.1 A multi-language architecture

One of Suricata's distinctive features is the multi-language approach, which combines C and Rust to implement different parts of the system. This choice reflects a strategy aimed at balancing performance and security, taking advantage of the strengths of each language.

5.1.1 The C language

C is one of the most popular programming languages in the world of low-level systems and applications. Due to its proximity to hardware and its ability to directly manage memory and resources, it is often the ideal choice for implementing high-performance software, such as Suricata. However, this proximity to hardware also entails a greater risk of security errors, such as buffer overflows and dangling pointers.

The role of C

Most of Suricata's architecture was originally written in C, especially the components related to resource management and performance-critical features. Some examples include:

- **Packet Capture**: The code to interface with libraries such as `libpcap`, `AF_PACKET` and `DPDK` is written in C, allowing for direct, high-performance handling of network packets.
- **Inspection Engine**: The core functions for parsing and decoding packets, as well as applying detection rules, are implemented in C.
- **Interface with external libraries**: Suricata uses C to integrate external libraries such as `libhtp` for HTTP processing.

5.1.2 The Rust language

Rust is a relatively new programming language that stands out for its strong focus on safety and memory management[15]. Thanks to an innovative property system, Rust eliminates entire categories of memory-related bugs, such as buffer overflows and use-after-free. Furthermore, Rust offers performance comparable to that of C, making it an attractive choice for developing software.

Rust in Suricata

Suricata has adopted Rust for some targeted components, with the goal of improving the security and maintainability of the code without sacrificing performance. Some examples include:

- **Advanced Parsing**: Some protocol parsers, such as the one for TLS, have been rewritten in Rust to reduce the risks associated with manual memory management.
- **Experimental Modules**: New features, such as parts of the advanced analysis of encrypted traffic, are implemented in Rust to take advantage of its inherent security.
- **Shared Libraries**: Suricata uses Rust components for common libraries, contributing to the improvement of the open-source ecosystem.

5.1.3 A balance between performance and security

The choice to combine C and Rust allows Suricata to take advantage of the best of both worlds: the high performance and flexibility of C for critical operations, combined with the security and maintainability of Rust for more complex or potentially risky features. This multi-language approach allows it to address the technical challenges of a modern IDS/IPS engine, while maintaining a focus on quality and innovation.

5.2 Analysis of the main components

Suricata's architecture is composed of distinct modules, each designed to handle specific functionalities. This modularity allows Suricata to effectively address a wide range of network and threat scenarios, while maintaining high flexibility.

5.2.1 Packet Processing: traffic management and analysis

Packet Processing is the first step in processing network data. This module is responsible for capturing packets, decoding them, and extracting relevant information for subsequent analysis stages.

Packet Capture

Suricata uses efficient packet capture libraries. Capture occurs directly at the network driver level, allowing Suricata to access packets as they pass through the monitored network. This is optimized by:

- Multi-threaded support: Suricata distributes the workload across multiple CPU cores, capturing packets from multiple interfaces or a single interface with traffic split into flows.
- Offloaded capture: In high-speed environments, Suricata can leverage technologies such as offloaded NIC to improve performance and reduce CPU load.

Among the libraries used for capture we have:

1. **libpcap**[10]
   It is a widely used open-source library for capturing and filtering network packets at the user level. It is the heart of tools such as Wireshark and, indeed, Suricata, providing an interface to directly access network packets from the card in promiscuous mode. This allows developers to create network analysis applications. Its simplicity and portability between different operating systems, such as Linux, macOS and Windows, make it very popular. However, it has performance limitations in high-traffic environments, as packets must be transferred from kernel mode to user-space, which introduces overhead.
2. **AF_PACKET**
   This is a native interface of the Linux kernel that allows raw packets to be received and sent directly to network cards, without going through the full TCP/IP stack. This approach operates at a lower level than `libpcap`, making it ideal for applications that require direct and fast packet handling. One of its distinguishing features is its support for "fanout", which allows packets to be distributed across multiple threads or processes, improving parallel processing. While it offers better performance than `libpcap` in high-traffic environments, it is more complex to manage and is not portable to operating systems other than Linux.

3. **DPDK (Data Plane Development Kit)**[7]
   represents an even more advanced solution for high-performance network packet handling. It moves packet processing from kernel space to user space, eliminating many of the bottlenecks associated with the traditional TCP/IP stack. By using polling-mode drivers, DPDK provides direct access to network hardware, allowing it to process millions of packets per second. It requires specialized hardware to take full advantage of its capabilities and involves complex configuration. Additionally, it is primarily supported on Linux, limiting its adoption in other environments.

Protocol Decoding

Suricata supports a wide range of network protocols, from IP and TCP/UDP to application protocols such as HTTP, TLS, DNS and SMB. This decoding capability allows for detailed and contextual analysis of traffic.

- Hierarchical Pipeline: Decoding occurs following a hierarchy of protocols, from the physical layer down to the application layer.
- Dynamic Detection: Suricata can automatically identify the protocol in use, even on non-standard ports, by analyzing the packet contents.

Data Normalization

Captured traffic is normalized to eliminate unnecessary variations that could mask attack patterns. For example, Suricata reassembles IP packet fragments and reconstructs TCP flows to obtain a complete view of connections.

5.2.2 Detection Engine: the heart of detection

The Detection Engine is the central component of Suricata, responsible for detecting threats using a rules and signature-based approach.

Detection Engine Architecture

The Detection Engine works by applying rules defined in a specific language, designed to describe patterns or conditions that allow suspicious or malicious activity to be identified.

One of its key features is the adoption of a parallel pipeline, which allows for the simultaneous analysis of multiple packets or data flows. This capability takes full advantage of multi-threaded support, ensuring high efficiency even in environments with significant traffic volumes.

Additionally, the engine implements rule optimization mechanisms, splitting rules into sets based on pre-filtering criteria. This approach reduces the number of rules that need to be applied to each packet, thus improving overall performance without sacrificing analysis accuracy.

Rule Types

- Signature-based rules: Identify known threats through predefined patterns (e.g. specific strings in payloads).
- Behavior-based rules: Detect suspicious activity by observing anomalous sequences of events.
- Custom rules: Users can define specific rules for particular scenarios, improving the system's adaptation to local needs.

Multi-protocol detection

Unlike many traditional IDSs, Suricata offers native support for application-layer threat detection. For example:

- DNS query analysis to detect tunneling or suspicious domains.
- HTTP traffic inspection to detect web exploit attempts.
- TLS certificate decoding to detect anomalies or insecure configurations.

5.2.3 Logging and Reporting: Visibility into Network Events

Suricata's logging system is essential to provide visibility and context into detected events. Proper log management allows to analyze incidents, monitor rule effectiveness, and identify anomalous behavior in network traffic.

Log types

Suricata supports different logging formats, adapting to multiple usage scenarios:

- **EVE JSON**: Structured format based on JSON, used for advanced and configurable logs. It is ideal for integration with tools such as ELK (Elasticsearch, Logstash, Kibana) and SIEM. For example:

```
{
 "timestamp": "2023-12-27T12:34:56.789Z",
 "event_type": "alert",
 "src_ip": "192.168.1.1",
 "dest_ip": "192.168.1.2",
 "alert": {
  "signature": "ET MALWARE Possible Malware Traffic",
  "severity": 2
 }
}
```

- **PCAP**: Binary format that captures raw network packets, useful for forensic analysis with tools such as Wireshark. Contains data that is not directly readable as text but is useful for reconstructing network traffic.
- **Syslog**: Standard protocol for sending centralized logs. Useful for integrating with log servers like Rsyslog or Splunk. An example:

```
<165>1 2023-12-27T12:34:56.789Z suricata-host suricata
1234 - - [event] alert: ET MALWARE Possible Malware
Traffic src_ip=192.168.1.1 dest_ip=192.168.1.2
```

- **File Log**: Logs in separate text files for specific protocols, such as HTTP, DNS, and TLS. For example:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: curl/7.68.0
```

- **Alert Logs (Plain text)**: Plain text file containing detected alerts, readable directly without the need for advanced tools. For example:

```
12/27/2023-12:34:56.789 [**] [1:12345:1] ET MALWARE
Possible Malware Traffic [**] [Priority: 2] {TCP}
192.168.1.1:12345 -> 192.168.1.2:80
```

External System Integration

Suricata can send logs to external platforms for centralized and advanced management:

- SIEM (Security Information and Event Management): Like Splunk or Graylog, to correlate network events with other data sources.
- Monitoring Dashboards: Tools like Kibana allow you to create interactive visualizations of traffic and detected threats.

Reporting and Automation

Suricata supports the generation of custom reports based on log data, giving administrators a detailed overview of system effectiveness and network activity.

5.3 Suricata Rule Format and Possible Actions

Rules[8] in Suricata are text strings structured according to a specific format, designed to identify particular events or behaviors in network traffic. Each rule combines match conditions with a specific action to take when those conditions are met.

5.3.1 General Rule Structure

A rule in Suricata is composed of two main parts:

1. **Header:** Specifies the type of traffic to monitor. Composed by:
   - Action: Specifies the action to take (e.g. alert, log, drop).
   - Protocol: The network protocol to apply the rule to (e.g. TCP, UDP, ICMP).
   - Addresses and Ports: Specifies the source and destination of the monitored traffic (e.g. `any any` or `192.168.1.1 80`).
   - Direction: Identifies the flow of traffic (e.g. ->, <-, <>).
2. **Body:** Contains options that define the matching criteria:
   - msg: A descriptive message that is logged if there is a match.
   - content: Specifies the data or pattern to search for in the packet payload.
   - SID: A unique identifier for the rule.
   - Other options: These can include criteria such as packet size, TCP flags, patterns in application protocols, etc.

Example rule:

```
alert tcp any any -> 192.168.1.1 80 (msg:"Possible HTTP attack";
content:"malicious"; sid:100001;)
```

The defined action is `alert`, which indicates that when the rule conditions are met, only an alert will be generated. This rule applies to TCP traffic, as specified by the `TCP` attribute.

The monitored traffic is that which starts from any source IP address and port, indicated by `any any` respectively, and is directed to the IP address `192.168.1.1` on port `80`, typically used for the HTTP protocol. The symbol `->` specifies that the direction of the traffic considered is unidirectional, i.e. from the source IP/ports to the destination IP/ports. To monitor bidirectional traffic, the symbol `<->` could be used.

Inside the round brackets, you can find the rule options. The option `msg:"Possible HTTP attack";` defines the message associated with the alarm, which is recorded in the logs to indicate the cause of the activation. The `content:"malicious";` condition specifies that the `"malicious"` string must be present in the TCP packet payload for the rule to be triggered. The analysis focuses only on application data, making this condition critical for detection. Finally, the `SID:100001;` option assigns a unique identifier to the rule. Identifiers below 100000 are reserved for official rules, while those above 100000 are used for custom rules.

This rule is used to monitor suspicious activity on an HTTP server, detecting content that may indicate an attack. For example, if a packet to IP `192.168.1.1` on port `80` contains the `"malicious"` string in its payload, an alert will be generated with the message `"Possible HTTP attack"`.

5.3.2 Available Actions

1. **alert:** Generates an alert and logs the event. This action is used to identify potential threats and create logs for analysis.

   Example:

   ```
   alert tcp any any -> any 80
   (msg:"Alert example"; content:"attack"; sid:1;)
   ```

2. **pass:** Stops further inspection of the packet. Typically used to whitelist known safe traffic or reduce false positives.

   Example:

   ```
   pass tcp any any -> any 80
   (msg:"Pass example"; content:"safe_traffic"; sid:2;)
   ```

3. **drop:** Drops the packet without forwarding it and generates an alert. This action is primarily used in IPS mode to silently block malicious traffic.

   Example:

   ```
   drop tcp any any -> any 80
   (msg:"Drop example"; content:"malicious"; sid:3;)
   ```

4. **reject and rejectsrc:** Blocks the packet and sends a TCP RST(TCP flag that signal that the sender should reset the connection state) or ICMP unreachable error message to the sender. This provides feedback to the source of the matching packet.

   Example 1:

   ```
   reject tcp any any -> any 80
   (msg:"Reject example"; content:"deny"; sid:4;)
   ```

   Example 2:

   ```
   rejectsrc tcp any any -> any 80
   (msg:rejectsrc example"; content:"source_block"; sid:5;)
   ```

5. **rejectdst:** Sends the RST/ICMP error to the receiver (destination) of the packet.

   Example:

```
rejectdst tcp any any -> any 80
(msg:rejectdst example"; content:"dest_block"; sid:6;)
```

6. **rejectboth:** Sends RST/ICMP error packets to both the source and the destination of the communication.

   Example:

   ```
   rejectboth tcp any any -> any 80
   (msg:rejectboth example"; content:"both_block"; sid:7;)
   ```

5.3.3 Some Advanced Options

- threshold: Limits the alert rate to reduce noise and false positives.
- classtype: Categorizes the rule into a specific class for organization and priority.
- flow: Specifies the flow direction (e.g. from_client, to_server).
- http_*: Dedicated options to analyze HTTP traffic (e.g. http_method, http_uri)

5.3.4 Advanced Example: SQL injection attempt

The following rule:

```
alert http any any -> any 80 (msg:"SQL Injection Attempt";
content:"SELECT"; nocase; http_uri; sid:200002;)
```

This rule applies only to the HTTP protocol, indicated by the `http` prefix. It monitors traffic from any IP address and port, to any destination IP on port 80, commonly used for HTTP traffic. When triggered, the rule generates an `alert` action, indicating the potential presence of a threat. The message associated with the alert is `"SQL Injection Attempt"`, which is useful for quickly identifying the event in the logs generated by Suricata.

The content searched for in the communication is the `"SELECT"` string, a distinguishing feature of SQL queries that may suggest an SQL injection attempt. The `nocase` modifier ensures that the search is not case-sensitive, which ensures that the rule works regardless of the capitalization used in the input. Additionally, the `http_uri` directive specifies that the search should be limited to the HTTP request URI, a common location for such attacks.

5.4 Suricata's Two-Phase Architecture

Suricata's architecture can be broadly divided into two distinct phases: the setup phase and the runtime phase. Each phase is designed to handle specific tasks essential for efficient packet analysis and threat detection.

5.4.1 Setup Phase

During the setup phase, Suricata initializes its internal structures and prepares the system for traffic inspection. This involves loading the configuration files, which specify rules, output formats, and other operational parameters. Various components, such as protocol parsers, packet decoders, and logging outputs, are registered and made available.

Rules are optimized through a process known as *prefiltering*, where each rule contributes one primary condition—often the most computationally efficient one—for pre-matching. This step significantly reduces the number of rules that require full inspection during runtime. A core element of this optimization is the use of *Multi-Pattern Matching (MPM)*, also referred to as *fast_pattern*. The MPM engine extracts a single unique pattern from each rule and compiles them into an efficient pre-matching mechanism.

Protocol parsers are linked to their respective handlers, enabling Suricata to understand and process different network protocols. Similarly, decoders for protocols like Ethernet, IP, and TCP/UDP are initialized to ensure that packets can be accurately dissected. Output modules, such as JSON, Syslog, or file-based logging, are configured to handle alerts and event logs.
This phase ensures that Suricata is fully prepared to process network traffic effectively.

5.4.2 Runtime Phase

The runtime phase begins when Suricata starts capturing traffic from the network. This phase is where the actual inspection and analysis of packets occur. The process follows a well-defined pipeline:

When a packet is captured by Suricata, it is first passed to the packet acquisition layer. This layer interacts with the underlying network stack or capture engine, such as `libpcap`, `AF_PACKET`, or `DPDK`, depending on the configuration. After acquisition, the packet enters the decoding stage.

At the decoding stage, Suricata dissects the packet into its individual layers, starting from the link layer (e.g., Ethernet) and moving up through the network (IP) and transport layers (TCP/UDP). Each layer is parsed using the decoders initialized during the setup phase. If a protocol-specific payload is identified, it is forwarded to the appropriate parser for deeper inspection.

Once decoded, the packet enters the detection engine, where the *prefiltering* mechanism comes into play.

In the prefiltering step, the Multi-Pattern-Matcher or MPM engine scans the packet payload against the patterns extracted during the setup phase. Only if the MPM detects a match does the packet undergo further detailed inspection against the full set of rules associated with the matched pattern. This tiered approach ensures high performance by eliminating the need to evaluate every rule for every packet.

If a packet triggers a rule match, Suricata may execute one of several configured actions, such as generating an alert, logging the packet, or dropping it entirely. Additionally, output modules configured during the setup phase handle the recording of logs and alerts in formats such as JSON or PCAP, allowing for integration with external analysis tools or storage systems.

Packets that do not match any rules continue through the pipeline without interruption, ensuring minimal impact on non-malicious traffic.

This structured and modular pipeline enables Suricata to deliver high-performance and accurate threat detection across diverse network environments.

5.5 Multi-Threading and Multi-Protocol Support in Suricata

Suricata stands out among Intrusion Detection System (IDS) tools for its ability to scale efficiently on modern hardware thanks to its native support for multi-threading and recognition of a wide range of protocols. These features make it particularly suitable for complex and high-performance environments.

5.5.1 Multi-Threading Support

One of the key innovations of Suricata is the ability to fully exploit multi-core architectures, ensuring high performance even on networks with significant traffic volumes. The analysis engine is designed to divide the workload between multiple threads, each of which handles a portion of the traffic.

- **Multi-Threaded Architecture:** Suricata uses a flexible threading model, where packets are distributed to threads based on available CPU. This approach minimizes bottlenecks and maximizes the use of hardware resources.
- **Load Balancing:** Traffic can be distributed to threads using algorithms such as:
  - Flow-based balancing: Packets belonging to the same flow are assigned to the same thread to ensure consistency in the analysis.
  - Packet-based balancing: Packets are dynamically distributed to balance the load between cores.
- **Performance Tuning:** The user can manually configure the number of threads, adapting it to the characteristics of the network and hardware. The configuration is done through the `suricata.yaml` file, which also allows to optimize the use of memory and CPU cache.

5.5.2 Multi-Protocol Support

Suricata is designed to analyze network traffic at various OSI model layers, offering support for a wide range of protocols. This capability allows you to identify and manage threats that leverage less common protocols, often ignored by traditional IDS.

- Layer 3 and Layer 4 Protocols such as IP, TCP, UDP, ICMP, IPv6 and offers in-depth header analysis and IP fragment handling.

- Layer 7 (Application) Protocol: Suricata can decode high-level protocols such as HTTP, FTP, SMB, SMTP, DNS, and TLS/SSL. Native protocol decoding provides detailed traffic insights, improving detection accuracy.
- Automatic Protocol Identification: An advanced feature of Suricata is the ability to identify application protocols regardless of the port used (Protocol Identification, DPI). This means that HTTP or DNS traffic is recognized even if it is not traveling on standard ports (e.g. 80 or 53).

5.5.3 Operational Benefits

With these features, Suricata offers several advantages:

- Scalability: It can handle complex enterprise networks with high bandwidth.
- Accuracy: Multi-protocol support provides in-depth analysis even for encrypted traffic or lesser-known protocols.
- Efficiency: The multi-threaded model ensures low latencies and high throughput, reducing processing times.

5.5.4 Limitations and Challenges

Despite the many advantages, Suricata's multi-threaded and multi-protocol support is not without its challenges. First, optimizing performance on specific hardware can be complex, requiring advanced technical skills and in-depth knowledge of available configurations. Second, the multi-threaded model, while ensuring high performance, requires significant resource allocation, especially CPU and memory, which may be a limitation in environments with limited hardware. Finally, analyzing encrypted traffic is one of the most significant challenges for intrusion detection systems like Suricata. Although Suricata supports protocols such as Transport Layer Security (TLS), which is used to secure communications over the Internet, deep inspection of encrypted traffic is a complex task. When data is encrypted, its content becomes unreadable without a decryption key, preventing Suricata from applying deep analysis techniques such as pattern searching or anomaly detection in transmitted data.

Suricata can handle some aspects of the TLS protocol, such as analyzing initial negotiations (handshakes) and certificates. However, this capability is limited to processing the information visible in the clear during the connection establishment phase. Subsequent traffic, encrypted using TLS, remains inaccessible unless advanced techniques are implemented, such as decryption using private key access or Man-in-the-Middle (MitM) approaches. However, these techniques raise significant legal implications and privacy concerns, making them difficult to use at scale. Furthermore, the adoption of technologies such as Perfect Forward Secrecy (PFS) makes these solutions even more difficult to implement, as the ephemeral keys used in sessions make access to the master keys useless.

The difficulty of analyzing encrypted traffic increases when it comes to detecting advanced attacks hidden within protected connections, such as "Man-in-the-Browser" or

malware that extracts sensitive data. Without direct access to the transmitted data, such threats risk going unnoticed. Despite these limitations, there are opportunities for improvement, Suricata could improve the analysis of TLS certificates or integrate decryption tools that work transparently to analyze traffic for specific environments (e.g. enterprise), while balancing security and privacy needs.

Chapter 6

Strengths and Weaknesses

Suricata is recognized as one of the most advanced IDS/IPS systems due to its scalable and flexible architecture. However, like any technology, it has strengths that make it ideal for certain scenarios and weaknesses that must be managed to get the most out of its use.

In this chapter, we will highlight the benefits of multi-threading and multi-protocol support. We will then explore the limitations that emerge in standard configurations, where inefficiencies or difficulties in adapting it to specific contexts may arise. Finally, we will discuss the need for optimization and customization, which are fundamental to ensure effective integration in different operating environments.

The goal is to provide a comprehensive overview that allows you to understand not only the benefits of Suricata, but also the challenges to be faced in order to fully exploit its potential.

6.1 Hardware Performance

One of Suricata's most distinctive qualities is its ability to leverage modern hardware to deliver high performance, especially in environments with heavy network traffic. This is achieved through an advanced design that combines multi-threading support, the use of advanced network adapters, and optimal management of available resources.

Native multi-threading support allows Suricata to distribute the workload across multiple CPU cores, processing packets in parallel. This feature is particularly beneficial for hardware with multi-core processors, where high throughput can be achieved while maintaining low latency[5].

Suricata can also leverage advanced network adapters that support technologies such as SR-IOV (Single Root I/O Virtualization) and DPDK (Data Plane Development Kit). These features provide significant benefits:

- SR-IOV allows direct interaction with the NIC (hardware component that allows a device to connect to a network), reducing kernel bottlenecks.
- DPDK allows packet processing at wire speed, bypassing the traditional TCP/IP stack to improve overall performance.

These optimizations are critical in high-performance environments, such as enterprise networks or data centers. In these environments, Suricata scales linearly with increasing hardware resources, managing network speeds above 10Gbps without sacrificing analysis accuracy[12].

However, even on high-end hardware, there are some challenges. For example, the coordination between threads introduced by the multi-threaded model can lead to significant overhead if configurations are not well-optimized. Additionally, processing large volumes of traffic can saturate available memory, negatively impacting overall system performance.

Despite these limitations, Suricata remains an excellent choice to fully exploit the capabilities of powerful hardware. Adopting targeted configurations and optimizing resources are key to ensuring maximum performance in complex and traffic-intensive network scenarios.

6.2 Limitations in Standard Configurations

Although Suricata is a high-level IDS/IPS, its standard configurations can have significant limitations that affect its effectiveness and usability, especially in complex operating environments. These limitations mainly arise from the need to balance performance, flexibility and ease of use in the default settings.

6.2.1 Non-Optimized Performance

Suricata's default configurations are designed to work in a wide range of scenarios, but this generality can be inefficient in specific contexts. For example, the default rules may not be adequately calibrated for the traffic volumes or threat types of a given network environment. This leads to:

- Increased resource consumption: Some processes may require more CPU or memory than necessary.
- Reduced speed: Failure to optimize rules and processes can slow down traffic processing.

6.2.2 Rules Configuration

Suricata's rules are the heart of its operation, but their default configuration can be too generic. This leads to two main problems:

1. Too many active rules: The default rules are designed to cover a wide range of scenarios, including many rules not needed for specific environments, which increases processing time.
2. Non-custom rules: The lack of environment-specific rules increases the likelihood of false positives and false negatives.

6.2.3 Interface and Configuration Complexity

Suricata's interface and configuration system, while powerful, can be challenging, especially for less experienced users. While using YAML files simplifies many tasks, some key options, such as load balancing between threads or advanced NIC optimization, require in-depth technical knowledge to configure correctly.

The documentation, while comprehensive, can be difficult to interpret, especially for those new to the system. This can significantly slow down the configuration process. In addition, incorrect settings or incomplete configurations can compromise performance, making the system less effective and more resource-intensive.

6.2.4 Generic Approach to Protocols

Suricata supports a wide range of protocols, providing considerable versatility in detecting threats in different network scenarios. However, standard configurations are not optimized for networks that use a narrow subset of protocols. This generic approach can lead to inefficiencies, as highlighted in the chapter 10. For example, in a network that primarily uses protocols such as HTTP and DNS, default support for lesser-used protocols such as FTP or SMTP can lead to unnecessary resource consumption, slowing down the overall system operation.

This issue is particularly relevant when analyzing the source code of Suricata and its modular functionality. In the next part, we will delve into this aspect in more detail, with a detailed analysis of the changes that can be made to the code to selectively disable support for non-essential protocols, thus optimizing the performance and efficiency of the system.

Part III

Suricata in Practice: Customization and Optimization

Chapter 7

Project objectives

Suricata offers a wide range of features and support for numerous network protocols, however, this flexibility comes at a cost in terms of resource usage. Each protocol supported by Suricata defines global structures and processes that, even if not used, can increase memory and processor consumption.

In many operational scenarios, such as specialized networks or environments with limited resources, the need to monitor only a subset of protocols makes it inefficient to keep all supported protocols active. From this observation comes the main goal of the project: to make protocol support optional in Suricata, allowing administrators to disable unnecessary ones, resulting in a reduction of the software footprint and improved performance.

In this chapter, we will define the project requirements, linking them to the operational scenarios that benefit most from a modular approach.

7.1 Target Scenarios

The changes to Suricata have been designed to adapt to specific contexts where network monitoring requires greater lightness and optimization. Among these, the following main scenarios stand out:

1. **Enterprise networks with limited protocols**
   In many organizations, network traffic is dominated by a few core protocols, such as HTTP, HTTPS, and DNS. In such environments, maintaining support for lesser-used protocols, such as FTP (File Transfer Protocol) or SCTP (Stream Control Transmission Protocol, transport protocol that allows the creation of multiple independent flows within the same connection), consumes valuable resources. Optimizing Suricata to monitor only the protocols that are actually needed increases efficiency and reduces system load.

2. **Resource-constrained devices**

   - **Embedded and IoT environments:** Routers, switches, and firmware-based network devices, such as OpenWrt, are often resource-constrained. A more

modular and streamlined Suricata can run in these environments by reducing memory and CPU consumption.

- **Lightweight virtualization:** In containers, reducing software size and resource usage is crucial to maintaining a high container density without compromising overall performance.

3. **High-Performance Networks**
   In high-speed environments, such as data centers and clouds, the ability to process large volumes of traffic is essential. By eliminating support for unnecessary protocols, Suricata can focus its resources solely on analyzing critical traffic, improving performance.

7.2 Project Requirements

To ensure that the proposed changes are practical and easily adoptable, it was necessary to establish some fundamental requirements:

1. **Modularity of protocol support**
   It is essential to make Suricata more flexible, allowing users to enable or disable support for certain protocols during the configuration or compilation phase. This requires a modular approach, where each protocol is separated into independent components, which can be activated via compilation flags or configuration options.

2. **Compatibility with existing configurations**
   The changes must not break the functionality of standard Suricata installations. To this end, the new features will be implemented in a non-invasive way, leaving the default behavior of the system unchanged.

3. **Resource optimization**
   The goal is to reduce CPU and memory consumption without sacrificing effectiveness in detecting threats. Data structures and processes related to the excluded protocols must be optimized to avoid consuming unnecessary resources.

4. **Adaptability to different contexts**
   Suricata must be easily customizable for different types of environments, from corporate networks to IoT systems, through cloud infrastructures. This involves providing well-documented configuration options that make it easy to adopt changes.

5. **Code maintainability**
   Changes must follow the official project guidelines, avoiding introducing unnecessary complexity. Code modularization will allow the software to be kept updatable and manageable, facilitating any future developments.

Chapter 8

Build Flow and Core Files

The Suricata build process follows a well-defined pipeline that combines configuration, generation of intermediate files, and the actual build. This flow is orchestrated by a series of key files, such as `configure.ac` and `Makefile.am`, that determine the features included, the dependencies managed, and the optimizations applied.

This chapter describes the build flow and analyzes the core files that make up the project architecture.

8.1 The Suricata compilation process

Suricata compilation follows the classic three-step process common to many open-source software: `./configure`, `make`, and `make install`. Each of these steps plays a specific role in preparing, compiling, and installing the software.

8.1.1 Running `./configure`

The `./configure` command starts the project configuration process, generating the files needed for the next compilation phase using the instructions defined in the `configure.ac` file. This script checks for the presence of required dependencies, such as libraries and development tools (for example, `libpcap`, `libhtp`, and `gcc`), and configures the Makefiles according to the system characteristics. During this phase, it is possible to enable or disable specific features using options such as `--enable-nfqueue`, feature that allows Suricata to interact directly with network packets through kernel-managed queues so that it can analyze the packets and decide whether to allow or block them. At the end of the execution, using the `Makefile.am` files (input files manually written by the developers) as a basis, the `Makefile` files are generated, which guide the compilation of the source code.

8.1.2 Running `make`

The `make` command uses the `Makefile` files generated in the previous phase to start the actual compilation. In this phase:

- The source code written in C and Rust is compiled, producing object files (`.o`) and shared libraries (`.so`).
- The parsers and other modular components are linked together via the linker, creating the main Suricata executable.
- The options configured in `./configure` determine which modules and protocols are included in the final binary.

At the end of this phase, the main executable (`suricata`) and the necessary libraries are ready for installation.

8.1.3 Running `make install`

The `make install` command copies the compiled files and default configurations to the appropriate system directories. Typically:

- The Suricata executable is installed in a directory such as `/usr/local/bin`.
- Configuration files (e.g. `suricata.yaml`) are copied to `/usr/local/etc/suricata`.
- Required libraries and files are placed in `/usr/local/lib`.

After this step, Suricata is fully installed and ready to run. You can start it by using the `suricata` command and specifying the desired configuration file and listening port.

8.2 `configure.ac`

The `configure.ac` file is one of the core elements of the Suricata build system, used to generate the `./configure` script. This script allows you to configure the project based on the characteristics of the target system, checking for the presence of dependencies, libraries and required tools. `configure.ac` is written using the `m4` language, which is specific to the `autotools` tools.

8.3 `/src/Makefile.am`

Taking into account the compilation flow described above, the starting point of the analysis of Suricata was the analysis of the file `Makefile.am`, in the directory `/src`, which defines how the various components of the system are included in the main library `libsuricata.a`. The variable `libsuricata_c_a_SOURCES` is central to this process, listing the source files needed for the compilation.

Each component contributes significantly to Suricata's capabilities, but also to its resource footprint. The indiscriminate inclusion of source files in the build process means that each supported protocol adds complexity and memory usage, even when unnecessary.

Here is an overview of the main files included in `libsuricata_c_a_SOURCES` and what they do:

8.3.1 Decoder (decode, decode-*)

These files are responsible for interpreting network packets at a low level. Each `decode-*` file handles a specific protocol, for example:

1. **decode-tcp:** Decoder for the TCP protocol.
2. **decode-sctp:** Decoder for the SCTP protocol.
3. **decode-ipv6:** Decoder for IPv6 packets.

The main `decode.c` file initializes the specific decoders and coordinates their invocation based on the type of packet received, so removing a decoder of a certain protocol means removing its reference from the `decode.c` file

Example: `decode-arp.c` The `decode-arp.c` file implements the `DecodeARP` function, which is responsible for decoding Address Resolution Protocol (ARP) packets. This function follows a rigorous flow of checks and interpretations to ensure that the packets are valid and compliant with standards.

The `DecodeARP` function analyzes ARP packets by performing a series of operations:

- **Stats Increment**: At the beginning of the function, a specific counter for processed ARP packets is updated:

```
StatsIncr(tv, dtv->counter_arp);
```

- **Packet Length Check**: Check if the packet is long enough to contain a minimum ARP header:

```
if (unlikely(len < ARP_HEADER_MIN_LEN)) {
 ENGINE_SET_INVALID_EVENT(p, ARP_PKT_TOO_SMALL);
 return TM_ECODE_FAILED;
}
```

- **ARP header parsing**: The `PacketSetARP` function extracts and parses the ARP header and assigns it to the packet structure:

```
const ARPHdr *arph = PacketSetARP(p, pkt);
if (unlikely(arph == NULL))
return TM_ECODE_FAILED;
```

- **Validity checks**: The function checks the main fields of the ARP header:
  - **Hardware type**: Must be Ethernet (`ARP_HW_TYPE_ETHERNET`).

- **Protocol**: Must be IPv4 (`ETHERNET_TYPE_IP`).
- **Opcode**: Checks whether the ARP operation (e.g. `Request` or `Reply`) is supported.

An example of a check:

```
if (SCNtohs(arph->hw_type) != ARP_HW_TYPE_ETHERNET) {
 ENGINE_SET_INVALID_EVENT(p, ARP_UNSUPPORTED_HARDWARE);
 PacketClearL3(p);
 return TM_ECODE_FAILED;
}
```

- **Conclusion**: If all checks pass, the function returns `TM_ECODE_OK`, indicating that the packet was decoded successfully:

```
return TM_ECODE_OK;
```

8.3.2 App-Layer (app-layer, app-layer-*):

These files handle logging and parsing of application layer protocols. They are responsible for invoking parsing functions for each protocol, such as HTTP, DNS, or TLS. Each `app-layer-*` file corresponds to an application layer protocol:

1. **app-layer-http:** Parsing HTTP requests and responses.
2. **app-layer-dns:** Handling DNS queries.
3. **app-layer-tls:** Parsing the TLS protocol.

Example 1: `app-layer-ftp.c`

The `app-layer-ftp.c` file implements support for analyzing the FTP (File Transfer Protocol) protocol, which is a network protocol used to transfer files between clients and servers over a TCP/IP network, characterized by the use of two connections: one used for data exchange and one used for sending commands from the client and receiving responses from the server. This file is responsible for decoding and analyzing the control connection, detecting any anomalies or suspicious behavior.

Main features

1. Protocol registration The `app-layer-ftp.c` file registers the FTP protocol in the Suricata application engine during the initialization phase. This is done through a function like `FTPAppLayerInit`, which maps the protocol to parsing and management functions:

```
void FTPAppLayerInit(void) {
 AppLayerParserRegister(IPPROTO_TCP, "ftp", ALPROTO_FTP,
 FTPParser, NULL);
}
```

In this example, the FTP protocol is registered for the TCP transport layer and maps to the `FTPParser` function for packet analysis.

2. Command and response parsing The main parsing function, `FTPParser`, analyzes the data from the control connection. This function identifies FTP commands and responses based on protocol delimiters, such as end-of-line characters (`\r\n`):

```
int FTPParser(Flow *f, void *state, AppLayerParserState *pstate,
uint8_t *input, uint32_t input_len) {
 // Parsing FTP commands
 FTPCommandParse(input, input_len);
 // Parsing FTP responses
 FTPResponseParse(input, input_len);
}
```

Each command or response is parsed and mapped to an internal state to track the flow of the connection.

3. Managing protocol state `app-layer-ftp.c` uses a data structure to track the state of the FTP connection, associating commands and responses. This is useful for identifying anomalies, such as invalid command sequences or the use of potentially malicious commands:

```
typedef struct FTPState {
 uint8_t cmd[MAX_CMD_LEN];
 uint8_t response[MAX_RESP_LEN];
} FTPState;
```

4. Callbacks and Engine Integration The file includes callbacks that are invoked when specific events are detected, such as a `USER` or `PASS` command. These callbacks allow the FTP parser to be integrated with other components of the system, for example to generate alarms:

```
if (strncmp(cmd, "USER", 4) == 0) {
 GenerateAlert(ALERT_FTP_USER_COMMAND);
}
```

Example 2: `ssh.rs`

The `ssh.rs` file, written in Rust, implements support for decoding the SSH (Secure Shell) protocol. SSH is a critical protocol used for secure connections, remote administration, and file transfers. This file takes care of decoding SSH traffic and provides features for identifying anomalies and collecting statistics.

Main Features

1. Parsing SSH packets The `ssh.rs` file implements a parser that analyzes SSH protocol messages, breaking them down into their main components. The `parse_ssh_packet` function is responsible for interpreting the received raw data and transforming it into a readable structure:

```
pub fn parse_ssh_packet(input: &[u8]) -> Result<SshPacket, SshError> {
 let mut cursor = Cursor::new(input);

 let packet_length = cursor.read_u32::<BigEndian>()?;
 let padding_length = cursor.read_u8()?;
 let payload = cursor.read_exact(packet_length as usize)?;

 Ok(SshPacket {
  length: packet_length,
  padding_length,
  payload: payload.to_vec(),
 })
}
```

This function extracts the main fields of an SSH packet such as the packet length, padding length and the actual payload, using the standard `Cursor` module that allows treating an in-memory buffer of data as an input source, similar to a file or stream.

2. Protocol data structures The file defines specific data structures to represent SSH messages. These structures are designed to be safe and easy to use during protocol analysis:

```
pub struct SshPacket {
 pub length: u32,
 pub padding_length: u8,
 pub payload: Vec<u8>,
}
```

Each SSH packet is represented by an instance of the `SshPacket` structure, which contains the decoded fields.

3. Error Handling Rust provides a robust error handling model, which is leveraged in the SSH decoder to identify and handle malformed packets. Errors are represented by a `SshError` enumeration:

```
#[derive(Debug)]
pub enum SshError {
 InvalidPacket,
 IoError(std::io::Error),
}
```

This approach allows to clearly distinguish between decoding errors (e.g. malformed packets) and input/output errors.

4. Packet parsing The `process_ssh_packet` function is used to parse packets received by the network engine and generate events:

```
pub fn process_ssh_packet(packet: &[u8]) -> Result<(), SshError> {
 let ssh_packet = parse_ssh_packet(packet)?;
 // Additional parsing logic
 Ok(())
}
```

5. Integration with the main engine The SSH decoder integrates with the rest of the system by exposing its main features through the `app-layer-ike.c` file that defines the function:

```
void RegisterIKEParsers(void)
{
 rs_ike_register_parser();
}
```

This function integrates the previously described functions written in Rust into the Suricata source code, written in C.

8.3.3 Detection (detect, detect-*)

Responsible for applying detection rules to identify suspicious or malicious activity:

1. **detect-engine:** Central engine for interpreting rules.
2. **detect-engine-register:** Part of the detection engine, they manage the registration of rules associated with different protocols. Each protocol has specific detection patterns, which are mapped to rules via these files.
3. **detect-content:** Implementation of content-based matching.
4. **detect-pcre:** Detection with regular expressions.

Example: `detect-smb-version.c`

The `detect-smb-version.c` file implements a detection rule, registered in `detect-engine-register`, specific to the SMB (Server Message Block) protocol, which is mainly used for sharing files and resources in networks. This file is responsible for identifying the SMB protocol versions within network traffic and associating them with the detection rules configured by the user.

Main features

1. Registering the SMB Version rule The `DetectSmbVersionRegister` function registers the SMB Version rule within the detection engine. This is done using the `SigMatchSignRegister` function, which maps a rule option to a processing function:

```
void DetectSmbVersionRegister(void) {
 SigMatchSignRegister("smb_version", DetectSmbVersionSetup, 0);
}
```

In this example, the `smb_version` rule option is mapped to the `DetectSmbVersionSetup` function for configuration.

2. Rule Configuration The `DetectSmbVersionSetup` function is called during the rule parsing phase. This function parses the parameters specified in the rule and associates them with a data structure representing the matching condition:

```
static int DetectSmbVersionSetup(DetectEngineCtx *de_ctx,
Signature *s, const char *str) {
 SmbVersionData *data = SCMalloc(sizeof(SmbVersionData));
 if (data == NULL)
 return -1;

 data->version = atoi(str);
 SigMatchAppendSM(s, DetectSmbVersionMatch, data);
 return 0;
}
```

In this snippet, the SMB version specified in the rule is extracted and saved in the `SmbVersionData` structure, which will be used during the matching phase.

3. Packet matching The `DetectSmbVersionMatch` function is the heart of the detection. When processing a packet, this function compares the packet data to the conditions specified in the rule:

```
static int DetectSmbVersionMatch(ThreadVars *tv,
DetectEngineThreadCtx *det_ctx, Flow *f, Packet *p, void *data) {
 SmbVersionData *smb_data = (SmbVersionData *)data;
```

```
 if (p->smb_version == smb_data->version)
  return 1; // Match found
 return 0; // No match
}
```

If the packet's SMB version matches the one specified in the rule, the function returns `1`, indicating a match.

8.3.4 Output (output, output-lua, output-json-*)

The output management in Suricata is responsible for logging and exporting detected events in different formats. The files associated with this component implement mechanisms to send data to external systems or save them in structured formats for further analysis.

1. **output:** Contains generic functions to manage output, allowing configuration and sending of logs and alarms in various formats.
2. **output-lua:** Handles integration with Lua scripts, allowing advanced customization of output. It is especially useful for scenarios where you need to transform or filter data before exporting it.
3. **output-json:** Implements log generation in JSON format, the most used structured format in Suricata. It provides detailed and highly configurable output for integration with analysis and visualization tools.
4. **output-json-alert:** Exports alerts in JSON format. Each alert includes details such as timestamp, IP addresses, ports, protocol, and the rule that triggered the alert.
5. **output-json-dns:** Specific to DNS traffic, logs queries and responses in JSON format for in-depth DNS traffic analysis.
6. **output-json-tls:** Saves TLS traffic details, such as certificates, protocol versions, and cipher suites, in a structured format useful for encrypted traffic analysis.

These output files offer a high degree of flexibility, allowing Suricata to adapt to different network infrastructures and analysis requirements.

Example: `output-json-dns.c`

The `output-json-dns.c` file implements support for outputting DNS traffic in JSON format. This file allows you to log DNS queries and responses in a structured format, useful for DNS traffic analysis and integration with analytics tools such as ELK (Elasticsearch, Logstash, Kibana).

Key Features

1. Output Module Registration The file registers the JSON output module for DNS during the Suricata initialization phase. This is done via the `TmModuleRegister` function:

```
void TmModuleJsonDnsLogRegister(void) {
 TmModuleRegister(TM_DNS_LOG, "dns-log", DnsLogThreadInit,
 DnsLogThreadExit, DnsLogDispatch);
}
```

This function registers the module with the name `dns-log`, associating it with specific functions for initialization, release, and output management.

2. Creating JSON output The `DnsLogDispatch` function is responsible for creating and sending JSON logs. When processing a DNS event, this function constructs a JSON object representing the traffic details:

```
void DnsLogDispatch(ThreadVars *tv, void *data, Packet *p) {
 json_t *js = json_object();

 json_object_set_new(js, "timestamp", TmTimeToJson(p->ts));
 json_object_set_new(js, "src_ip", JsonIPAddress(p->src_ip));
 json_object_set_new(js, "dst_ip", JsonIPAddress(p->dst_ip));
 json_object_set_new(js, "query", json_string(dns_data->query));
 json_object_set_new(js, "response", json_string(dns_data->response));
 json_object_set_new(js, "type", json_string(dns_data->type));

 OutputLogJson(tv, js);
 json_decref(js);
}
```

In this example:

- The timestamp and source/destination IP addresses are added to the JSON object using the `TmTimeToJson` and `JsonIPAddress` functions.
- Specific details of the DNS query and response, such as the domain name and query type (e.g. `A`, `MX`, `PTR`), are included.

3. Sending the JSON log After the JSON object is created, the `OutputLogJson` function takes care of sending it to the configured destination, such as a file or a network pipeline:

```
void OutputLogJson(ThreadVars *tv, json_t *js) {
 char *json_str = json_dumps(js, JSON_COMPACT);
 SCLogInfo("DNS Log: %s", json_str);
 SCFree(json_str);
}
```

8.4 /rust/Makefile.am and lib.rs

The Makefile in the `rust` directory is responsible for managing the compilation of components written in Rust. In a project that combines multiple languages, such as C and Rust, this file defines the rules for including Rust modules in the general compilation flow orchestrated by `autotools`. Among its various functions is the specification of *feature flags* used to enable or disable specific features at build time.

This Makefile is closely tied to the `lib.rs` file which represents the main entry point for code written in Rust within Suricata. In a Rust project, `lib.rs` is conventionally used to define the main module of a library or a software component. In the case of Suricata, this file coordinates and organizes the Rust modules that implement specific features.

Integration with C via FFI

A key aspect of the `lib.rs` file is its integration with C code. Thanks to support for FFI (Foreign Function Interface), functions written in Rust can be exposed and used by the rest of the project. For example:

```
#[no_mangle]
pub extern "C" fn rust_function() -> i32 {
 420
}
```

In this example, the `rust_function` function is made available to be called from C code, allowing for close collaboration between the two languages.

8.5 /rules/Makefile.am

The `Makefile.am` file in the `rules` directory is responsible for managing the configuration, installation, and organization of detection rules. Rules are a key component of how Suricata works, used by the detection engine to identify suspicious or malicious activity on the network. This file defines the `dist_rule_DATA` variable that contains the `.rules` rule files.

8.6 suricata.yaml.in

The `suricata.yaml.in` file is the main template for generating the `suricata.yaml` configuration file, which Suricata uses to define the behavior of the IDS/IPS engine. This file contains a detailed and flexible structure that allows users to configure key components, such as traffic analysis, supported protocols, detection rules, output formats, and more.

8.6.1 File Structure

The `suricata.yaml.in` file structure is organized into hierarchical sections, each representing a specific aspect of the engine.

Main File Sections

The details of the most important sections are as follows:

1. `vars` This section defines network variables, such as IP addresses and ports, that are used in detection rules. Example:

```
vars:
 HOME_NET: "[192.168. 1.0/24]"
 EXTERNAL_NET: "!$HOME_NET"
```

`HOME_NET` is the internal network being monitored, while `EXTERNAL_NET` is everything that is not part of the internal network.

2. `interfaces` In this section, you configure the network interfaces that Suricata uses to capture traffic:

```
af-packet:
 - interface: eth0
  cluster-type: cluster_flow
  defrag: yes
```

The `eth0` interface is configured to use `af-packet`, a high-performance capture mode.

3. `logging` The `logging` section defines settings for system-generated logs:

```
logging:
 default-log-level: info
 outputs:
  - console:
   enabled: yes
  - file:
   enabled: yes
   filename: /var/log/suricata/suricata.log
```

In this example, logs are sent to both the console and a specified file.

4. `outputs` This section manages the format and destination of the generated output. An example for JSON format:

```
outputs:
 - eve-log:
  enabled: yes
  filetype: json
  filename: /var/log/suricata/eve.json
  types:
   - alert
   - dns
   - http
```

In this case `eve-log` is configured to save alert, DNS and HTTP events in JSON format.

5. `app-layer` The `app-layer` section allows you to enable or disable specific application layer protocols. This means that Suricata will analyze or ignore traffic associated with those protocols during runtime, but it will not remove the global structures defined in the source code.

```
app-layer:
 protocols:
  http:
   enabled: yes
    request-body-limit: 100kb
    response-body-limit: 100kb
  dns:
   enabled: yes
```

In this example, HTTP and DNS support is enabled, with size limits configured for the HTTP request body.

8.6.2 Generating the `suricata.yaml` file

The `suricata.yaml.in` file is a template that contains placeholders, such as `@VARIABLE@`, that are replaced with values defined during configuration via `./configure`. For example:

```
HOME_NET: "@HOME_NET@"
```

When generating the `suricata.yaml` file, `@HOME_NET@` is replaced with the actual address specified by the user.

Chapter 9

Changes Made

The changes shown in this chapter are aimed at optimizing Suricata 8.0.0 to adapt it to devices with limited resources or with specific monitoring needs. Although Suricata offers support for many network protocols, this flexibility comes with a high resource consumption, even for unused features. The optimization implemented consists of providing the ability to choose which protocols to enable already at compile time, so as to exclude any structures in memory.

The integration of the changes must offer a reduction in CPU and memory consumption, without compromising the effectiveness of threat detection and provide adaptability to different operational scenarios. The modified source code has been uploaded to this GitHub repository and can be deployed following this guide.
Additionally, a comprehensive list of all the defined flags and the files modified for each flag is available in this other repository.

To better illustrate the nature of the modifications, the following sections will detail the changes made to various parts of the source code using the Internet Key Exchange (IKE) protocol as an example.

9.1 Adding Configuration Flags in `configure.ac`

To make a protocol optional, the first step is to define a flag, to be used at compile time to request its enabling or removal, and the relative variables to be used in the file associated to the protocol. For this reason, the following code fragment has been added to the `configure.ac` file:

```
AC_ARG_ENABLE([ike],
 [AS_HELP_STRING([--enable-ike], [Enable IKE protocol])],
 [if test "x$enable_ike" = "xyes"; then
  ENABLE_IKE=1
  ENABLE_IKE_STRING="yes"
 else
  ENABLE_IKE=0
  ENABLE_IKE_STRING="no"
```

```
 fi],
 [ENABLE_IKE=1
 ENABLE_IKE_STRING="yes"])

AC_SUBST(ENABLE_IKE)
AC_SUBST(ENABLE_IKE_STRING)
AC_DEFINE_UNQUOTED([ENABLE_IKE],[$ENABLE_IKE],[Enables support for IKE])

AM_CONDITIONAL([ENABLE_IKE], [test "x$ENABLE_IKE" = "x1"])
```

This code introduces the `--enable-ike` configuration option, which enables or disables IKE support and is based on the following Autoconf statements:

- `AC_ARG_ENABLE`: defines a configurable option for the `./configure` command. This macro accepts:
  1. feature name.
  2. `AS_HELP_STRING` which contains the description that appears in the `./configure` command help.
- if directive: Checks whether the user specified `--enable-ike` during configuration and defines the global variable `ENABLE_IKE`. If not explicitly disabled (via `--disable-ike`) the feature is enabled by default.
- `AC_SUBST`: Macro that tells Automake to substitute the value of the specified variable (`ENABLE_IKE` or `ENABLE_IKE_STRING`) in generated files, such as Makefiles and `.yaml`.
- `AC_DEFINE_UNQUOTED`: Defines a macro in C code during compilation. Accepts:
  1. The name of the defined macro.
  2. The value of the `ENABLE_IKE` variable, which will be 1 or 0.
  3. A short description added as a comment in the generated file.
- `AM_CONDITIONAL`: Defines a condition for Automake, based on the value of a variable. Requires:
  1. The name of the condition.
  2. the condition, in this case true if `ENABLE_IKE` is equal to 1.

  This condition can be used in the `Makefile.am` file to include or exclude files.

9.2 Changes to `src/Makefile.am`

Once the variables has been defined in the `configure.ac` file, the `Makefile.am` file has been modified to include IKE-related source and header files only if the protocol is enabled:

```
if ENABLE_IKE
 noinst_HEADERS += app-layer-ike.h
 noinst_HEADERS += detect-ike-exch-type.h
 noinst_HEADERS += detect-ike-spi.h
 noinst_HEADERS += detect-ike-vendor.h
 noinst_HEADERS += detect-ike-chosen-sa.h
 noinst_HEADERS += detect-ike-key-exchange-payload-length.h
 noinst_HEADERS += detect-ike-nonce-payload-length.h
 noinst_HEADERS += detect-ike-nonce-payload.h
 noinst_HEADERS += detect-ike-key-exchange-payload.h
 noinst_HEADERS += output-json-ike.h
endif

if ENABLE_IKE
 libsuricata_c_a_SOURCES += app-layer-ike.c
 libsuricata_c_a_SOURCES += detect-ike-exch-type.c
 libsuricata_c_a_SOURCES += detect-ike-spi.c
 libsuricata_c_a_SOURCES += detect-ike-vendor.c
 libsuricata_c_a_SOURCES += detect-ike-chosen-sa.c
 libsuricata_c_a_SOURCES += detect-ike-key-exchange-payload-length.c
 libsuricata_c_a_SOURCES += detect-ike-nonce-payload-length.c
 libsuricata_c_a_SOURCES += detect-ike-nonce-payload.c
 libsuricata_c_a_SOURCES += detect-ike-key-exchange-payload.c
 libsuricata_c_a_SOURCES += output-json-ike.c
endif
```

In this case headers and sources of IKE parser, IKE options detectors and IKE output formatter.

9.3 Updates to `suricata.yaml.in`

`suricata.yaml.in` has been modified to include the following condition under the `outputs` and `app-layer` sections:

```
- ike:
  enabled: @ENABLE_IKE_STRING@
```

In this file, it is not possible to use "1" or "0" values to express a condition. This is why an additional variable was defined to store a string value instead. If the protocol is enabled, the `ENABLE_IKE_STRING` variable will hold the value "yes"; otherwise, it will be set to "no".

9.4 Changes to the `rules` folder

The `rules` directory contains files that define the detection rules that Suricata uses to analyze network traffic and generate alerts based on suspicious behavior or anomalies.

There are two major changes to this folder in the project:

1. The `decoder-events.rules` file, which originally contained rules for handling anomalies or errors detected during traffic processing, has been split into multiple files, each dedicated to a specific protocol. For example, the rules for DCERPC have been moved to the `decoder-events-dcerpc.rules` file, while those for GRE have been included in `decoder-events-gre.rules`.

2. In the `Makefile.am` file, the `dist_rule_DATA` variable is used to specify the `.rules` files that should be included in the distribution. The files have been split by protocol and made optional by using the `ENABLE_*` variables, previously defined in the `configure.ac` file. This approach allows you to include only the necessary files based on the protocols enabled during configuration. An example is the following:

   ```
   if ENABLE_IKE
    dist_rule_DATA += ipsec-events.rules
   endif
   ```

   Note that IKE rules are defined in the `ipsec-events.rules` file because IKE serves as a fundamental protocol for the implementation of IPsec.

9.5 Changes to `rust/Makefile.am`

As explained in the chapters before, Suricata is written using a mix of C and Rust languages, this is why, in the `rust/Makefile.am` file, the following code has been added to handle Rust features to enable them according to the variable defined in the `configure.ac`:

```
if ENABLE_IKE
 RUST_FEATURES += ike
endif
```

9.6 Updates to the `lib.rs` File

In the `lib.rs` file, a conditional module has been added that is enabled only if the `ike` feature has been enabled in the corresponding `Makefile.am`:

```
#[cfg(feature = "ike")]
pub mod ike;
```

9.7 Changes in Suricata C Code (`src`)

To dynamically enable or disable IKE-related code, `#if ENABLE_IKE` directives have been added. For example:

```
#if ENABLE_IKE
 RegisterIKEParsers();
#endif
```

These changes ensure that IKE-specific functions are only included in the program if IKE support is enabled.

9.8 Problems Encountered

During the project, some specific problems arose related to disabling certain protocols. These problems highlighted the importance of understanding the internal architecture of Suricata and its dependencies. The main obstacles encountered are discussed below.

9.8.1 HTTP and libhtp

The HTTP protocol handling in Suricata is delegated to an external library called `libhtp`. This library is responsible for parsing and normalizing HTTP data, providing an abstraction layer to simplify the processing of the protocol.

Disabling HTTP posed an additional challenge, as `libhtp` also had to be excluded from the build process. This means that the file `src/Makefile.am` had to be modified to remove references to the library and its associated files.

9.8.2 TCP and UDP

The TCP and UDP protocols, while theoretically disableable, are a fundamental part of network flow and how Suricata works. They play a key role in multiplexing and decoding source and destination ports, which are essential for analyzing network flows.

Disabling these protocols would drastically reduce Suricata's capabilities, making it impossible to identify communications between hosts. This would render the IDS ineffective for most use cases. Therefore, it was decided to keep these protocols enabled as part of the system.

Chapter 10

Change Impacts

The modifications made to the Suricata code to make certain protocols optional have had a notable impact on both the software's size and its performance. This chapter provides an in-depth analysis of these effects, emphasizing the benefits achieved as well as any limitations introduced.

The subsequent sections explore how these changes influenced the overall size of the software and examine their impact on the system's performance and flexibility.

All tests were conducted on a virtualized machine running `Ubuntu 24.10` with 8 cores and ≈ 9 GB of RAM.

10.1 Executable Size

The table 10.1 shows the results of the measurements of the Suricata executable in different configurations of Suricata itself:

Table 10.1. Size of Suricata's executable (in MB)

Configuration	**Dimension**
All protocols enabled	≈ 100 MB
Core protocols enabled (`network` flag)	≈ 55 MB
Only essential protocols (IPv4, TCP, UDP)	≈ 42 MB

10.2 Memory Usage

The memory usage has been measured using the following command:

```
sudo pmap -x $SURICATA_PID
```

This command is used to display the memory map of a process that is how memory is allocated and distributed within a running process. Table 10.2 shows the memory consumption measured during runtime:

Table 10.2. Suricata's memory usage (in GB)

Configuration	**Memory used**
All protocols enabled	≈ 1,91 GB
Core protocols enabled (`network` flag)	≈ 1,89 GB
Only essential protocols (IPv4, TCP, UDP)	≈ 1,04 GB

The possibility of excluding unused protocols contributed to a significant reduction in dynamic memory allocation.

To be noted that the major boost to the efficiency has been obtained removing the support of `HTTP` and the `libhtp` library.

10.3 Impact on log files

Each test has been made using the same sample, that is a capture of real network traffic on a busy private network's access point to the Internet. This log contains several packets, approximately 790k, of different protocols like IPv4 and IPv6, TCP and UDP, ARP, ICMP and many others.

We can note that removing the support to the protocols also means the generation of less events, the table 10.3 shows this result:

Table 10.3. Suricata's log file size (in MB)

Configuration	**Log size**
All protocols enabled	≈ 36.0MB
Core protocols enabled (`network` flag)	≈ 34.6MB
Only essential protocols (IPv4, TCP, UDP)	≈ 18.7MB

By reducing the number of supported protocols and eliminating their logs, the software's ability to analyze traffic in depth is reduced. This results in a reduction in the accuracy of threat detection. It is therefore essential to find the optimal balance between security and efficiency, ensuring high performance without excessively compromising monitoring and protection capabilities.

Part IV

OpenWrt: A Lightweight Network Firmware

Chapter 11

Introduction to OpenWrt

OpenWrt is an open source Linux-based operating system designed for network devices such as routers, access points, and other embedded platforms. Unlike pre-installed firmware from manufacturers, OpenWrt offers users a flexible and customizable platform that allows complete control over the functionality of the device.

Throughout this chapter, we will analyze why OpenWrt was chosen, its main features, and how it differs from a full Linux system. The goal is to highlight why OpenWrt is a popular choice for developers and networking enthusiasts.

11.1 From Evaluating Options to Choosing OpenWrt

Before selecting OpenWrt, other options were explored, including:

1. **VyOS:** An open-source Linux-based distribution designed for routing and network management. While VyOS is robust and highly configurable, it is more geared toward complex network infrastructures and less suited to resource-constrained embedded devices, this is easily notable looking at the minimum requirements of this OS. Additionally, its customization requires a level of abstraction that can prevent low-level optimization.

2. **OPNsense:** is an open-source distribution based on FreeBSD, designed primarily for firewalls, routers, and advanced network security. It offers a full-featured GUI for managing network features such as VPN, firewall, IDS/IPS, and more.

3. **Ubuntu:** As a full-featured Linux distribution, Ubuntu offers a wide range of features and a large community of support. However, its general structure and resource consumption make it poorly suited to devices with limited hardware capabilities, reducing operational efficiency compared to a leaner system like OpenWrt.

Of the options considered, OpenWrt allowed us to build a tailor-made system, capable of balancing efficiency, flexibility and scalability. Its intrinsic characteristics made it the ideal basis to address the challenges of the project.

11.2 What is OpenWrt and its main features

As Studied by Damasceno, Dantas, and Araujo [4], OpenWrt is a lightweight and versatile operating system, specifically designed for network devices and embedded environments. Unlike proprietary firmware provided by manufacturers, OpenWrt is open source and offers a wide range of features that make it a preferred solution for many networking scenarios.

11.2.1 Key Features of OpenWrt

1. **Modular Architecture**
   OpenWrt uses an optimized Linux kernel and a package-based system. This modular approach allows you to install only the components you need, minimizing resource consumption and increasing flexibility.

2. **Extensive Package Repository**
   OpenWrt has a rich package repository that includes advanced networking tools, lightweight servers, VPNs, firewalls, and more. Each package can be installed or removed as needed by the user.

3. **Extensive Hardware Support**
   OpenWrt supports a wide range of embedded devices, from low-cost routers to high-end networking systems. With community support, new devices are added all the time.

4. **Graphical Interface and CLI**
   The LuCi graphical interface simplifies configuration for novice users, while the CLI provides advanced control for more complex configurations.

5. **Continuous Updates and Security**
   The OpenWrt community ensures regular updates, feature enhancements, and timely security patches, reducing the risk of vulnerabilities in the firmware.

6. **Full Customization**
   Users can modify every aspect of OpenWrt, from the kernel to packages to specific configuration of networking features.

11.2.2 Differences from a full Linux system

Although OpenWrt is based on Linux, it differs from a full Linux system like Ubuntu or Kali in several key ways:

- Optimization for embedded devices: OpenWrt is designed to run on hardware with limited resources, such as low-power CPUs and low memory[20].

- Custom build system: The Buildroot framework allows you to create custom firmware images, including only the required components.

- Lacks some desktop features: To reduce size and resource consumption, OpenWrt excludes many graphics libraries and tools that are common in PC Linux systems.

These features make OpenWrt an ideal platform for developers, network administrators, and enthusiasts looking for a powerful, lightweight, and customizable solution for network devices.

Chapter 12

Advantages and Limitations of OpenWrt

OpenWrt stands out as a versatile and innovative platform in the panorama of firmware for network devices. Its features make it a privileged choice for scenarios that require efficiency, modularity and flexibility on embedded hardware. However, like any technology, it also presents challenges, especially regarding hardware compatibility and configuration complexity.

In this chapter, the main advantages of OpenWrt will be analyzed, highlighting how it manages to meet specific needs of advanced networking. Subsequently, the limitations of the platform will be discussed, offering a complete picture of its potential and the difficulties that may arise in its adoption and use.

The goal is to provide a clear understanding of the reasons why OpenWrt represents an effective solution for many scenarios, without neglecting the aspects that require attention or advanced technical skills.

12.1 Efficiency

OpenWrt's efficiency comes from its ability to optimize the use of hardware resources. For example, this OS uses dynamic memory allocation and release algorithms inherited from the Linux kernel, such as *kmalloc* and the *slab allocator* system, to optimize memory use in resource-constrained environments. *kmalloc* is a function that allows you to allocate contiguous blocks of memory in kernel space, ideal for small or medium-sized allocations. Internally, *kmalloc* often relies on the *slab allocator*, a subsystem designed to manage fixed-size objects, organizing them in dedicated caches to reduce fragmentation and overhead. The *slab allocator* reuses already allocated blocks of memory, thus improving overall efficiency and allocation speed.

In addition, OpenWrt benefits from other advanced features of the Linux kernel, such as the *OOM Killer* (Out-Of-Memory Killer), which kills low-priority processes in low-memory situations to preserve system stability. For extreme scenarios, support for swap partitions or files allows to temporarily extend virtual memory, reducing the risk of crashes.

Thanks to these mechanisms, OpenWrt can operate reliably even on devices with limited memory.

Furthermore, the size of the operating system can be customized to include only the strictly necessary modules, ensuring a very small footprint.

12.2 Modularity

One of the distinguishing features of OpenWrt is its modularity. The platform uses a package system that allows users to install only the components they need, minimizing the system footprint. Some examples include:

- Customizable packages: OpenWrt provides a large repository of packages, such as web servers, advanced firewalls, and monitoring tools, that can be added or removed as needed.
- LuCI interface: The web graphical user interface (LuCI) can be installed or uninstalled to optimize resource usage, offering a compromise between ease of use and light weight.

12.3 Scalability

Despite being designed for limited hardware, OpenWrt is highly scalable. It can be configured to run on a wide range of devices, from small home routers to powerful systems used in enterprise environments. Its scalability strengths include:

- Multi-architecture support: OpenWrt supports multiple hardware architectures, including MIPS, ARM, and x86.
- Advanced configurations: Advanced features, such as VLAN, VPN, and QoS, can be integrated without compromising system stability.
- Cluster and mesh networking: OpenWrt can be used to create scalable mesh networks and complex network infrastructures, such as Wi-Fi hotspot systems.

12.4 Challenges related to hardware compatibility and configuration

Despite its many advantages, OpenWrt does have some significant challenges, especially in terms of hardware compatibility and configuration. These aspects can be a stumbling block, especially for those who are new to the platform or operate in environments with a large variety of devices.

12.4.1 Hardware Compatibility

Hardware compatibility is one of the main pain points of OpenWrt. This firmware must adapt to a wide range of embedded devices[21], each with unique specifications and limitations.

The diversity of supported architectures, such as *MIPS*, *ARM*, and *x86*, provides considerable flexibility, but not all devices have official pre-built images.

Additionally, support for components such as Wi-Fi network cards depends on the availability of appropriate drivers. While many open source drivers are supported, others are only available in proprietary versions, limiting the options for the user. Added to this is the difficulty in dealing with older devices, for which manufacturers often stop releasing updates.

Another critical aspect is the installation procedures. For unsupported hardware, the user must resort to custom firmware builds or undocumented flashing techniques. This process, if not done correctly, can lead to the risk of rendering the device unusable, a condition known as "bricking".

12.4.2 Configuration Challenges

Even once installed, OpenWrt can be difficult to configure. While the LuCI web interface simplifies interaction for many users, some advanced options require manual management of configuration files.

For example, correctly configuring VLANs or Quality of Service (QoS) often requires detailed knowledge of networks. Files such as `/etc/config/network` or `/etc/config/firewall` must be carefully edited, and a mistake here can cause the device to fail.

Package management is another complex issue. Devices with limited flash memory force users to carefully choose which packages to install, balancing functional needs with space limitations. Additionally, updates can introduce dependency conflicts, especially when using custom builds of the firmware.

Finally, while OpenWrt offers extensive documentation, it is often technical and geared toward experienced users. Beginners may find it difficult to navigate the available information or understand the specific details of a given hardware or package.

12.4.3 Mitigating Challenges

Despite these difficulties, there are tools and resources that can help overcome many of the challenges of using OpenWrt. For example, the custom build system allows users to create optimized firmware images, avoiding the inclusion of unnecessary components and improving hardware support.

Furthermore, the OpenWrt community is extremely active and a great strength. Through forums, detailed guides, and contributions from experienced users, it is possible to find practical solutions to many issues.

Chapter 13

Technical Aspects of OpenWrt

Thanks to its ecosystem, composed of an SDK, a flexible toolchain and a highly configurable build system, OpenWrt allows developers to create custom firmware images to meet specific technical and application needs.

In this chapter, the main technical aspects of OpenWrt will be analyzed. We will start from the use of the SDK and the toolchain for compiling packages, then describe the build system based on Buildroot, highlighting the customization possibilities offered. Finally, the main features of the platform will be discussed, such as advanced routing, firewall, QoS management, VPN support and mesh networks. These technical tools and features consolidate OpenWrt as a versatile and scalable solution for a wide range of network applications.

13.1 SDK and Toolchain

13.1.1 What is the OpenWrt SDK

The OpenWrt **Software Development Kit (SDK)** is a set of pre-configured tools and resources that allow developers to create, modify, and compile software packages for embedded devices. The SDK is an isolated development environment, designed to simplify the process of creating packages without requiring full firmware compilation.

The OpenWrt SDK includes:

- A pre-compiled version of the **toolchain**, specific to the target architecture.
- Libraries and header files (`headers`) needed for development.
- A build system based on **Buildroot**, optimized to manage packages and dependencies.
- Scripts and tools for managing configurations and cross-compilation.

With the SDK, you can develop packages without having to build the entire firmware image. This saves time and resources, making the process more efficient. For example, a

user could download the ARM-specific SDK and compile a package like `tcpdump` with a few simple commands, without having to recompile the kernel or base system.

13.1.2 OpenWrt Toolchain

The OpenWrt **toolchain** is a set of tools used to compile software for embedded devices. The unique thing about the OpenWrt toolchain is that it is designed for **cross-compilation**, which is the process of generating executable binaries for a different architecture than the development system.

The toolchain includes:

- **GCC Compiler**: The GNU C/C++ compiler is configured to generate code for specific architectures, such as ARM, MIPS, or x86.
- **Binutils**: A collection of tools for managing binary files, including a linker and assembler.
- **C Library**: A lightweight C library, such as `musl libc`, designed for embedded environments to replace heavier libraries such as `glibc`.
- **Debugger**: Tools such as `gdb` for debugging compiled code.

With this configuration, the toolchain allows you to generate highly optimized binaries for devices with limited resources. Each target architecture has its own version of the toolchain, which includes optimizations for the specific instruction set of the hardware.

13.1.3 Differences between SDK and Toolchain

Although related, SDK and toolchain serve slightly different purposes:

- The **toolchain** is solely concerned with compilation, providing the compilers, linkers, and libraries needed to generate executable binaries for a specific architecture.
- The **SDK**, on the other hand, is a broader development environment, which includes the toolchain, but also adds configuration files, pre-compiled libraries, and scripts to facilitate the development of packages or applications.

13.1.4 Cross-Compilation Process

The process of cross-compiling with the SDK and the toolchain can be summarized in the following steps:

1. **Environment Setup**: Download the appropriate SDK for the target architecture and set the necessary environment variables, such as `STAGING_DIR`.
2. **Makefile Creation**: Every package in OpenWrt uses a `Makefile` file to define the sources, dependencies and build commands.

3. **Package Compilation**: Run the command
   `make package/<package-name>/compile`
   to generate the `.ipk` package.

4. **Installation on the device**: The generated package can be installed on the target device via the `opkg` package manager.

13.1.5 Practical Example of Use

To clarify the process of using the SDK and the toolchain, let's consider the example of creating a custom package called `hello-world`, which contains a simple program written in C language.

Package Source Code

The source code of our program, called `hello.c`, is a simple file that prints a message to the screen:

```
#include <stdio.h>

int main() {
 printf("Hello, OpenWrt!\n");
 return 0;
}
```

This file must be placed in a directory dedicated, for example `package/hello-world/src/hello.c`.

The Package `Makefile`

To create a package in OpenWrt, you need a package-specific `Makefile` file, which defines basic information such as the package name, the category it belongs to, dependencies, and the commands to build it. Here is an example `Makefile` for `hello-world`:

```
include $(TOPDIR)/rules.mk

# Package Definition
define Package/hello-world
SECTION:=examples
CATEGORY:=Examples
TITLE:=Hello World Example
endef

# Build Commands
define Build/Compile
$(TARGET_CC) $(TARGET_CFLAGS) -o $(PKG_BUILD_DIR)/hello \
$(PKG_BUILD_DIR)/src/hello.c
```

```
endef

# Installation Commands
define Package/hello-world/install
$(INSTALL_DIR) $(1)/usr/bin
$(INSTALL_BIN) $(PKG_BUILD_DIR)/hello $(1)/usr/bin/
endef

# Include the OpenWrt framework for packages
$(eval $(call BuildPackage,hello-world))
```

Explanation of `Makefile`

- `include $(TOPDIR)/rules.mk`: Includes the general OpenWrt framework for package management.
- **Section `Package/hello-world`**: - `SECTION`: Specifies the section the package belongs to (in this case, "Examples"). - `CATEGORY`: Defines the category the package belongs to (e.g. "Examples"). - `TITLE`: Provides a short description of the package.
- **`Build/Compile`** section: - Uses the target compiler (`$(TARGET_CC)`) and build flags (`$(TARGET_CFLAGS)`) provided by the toolchain to build the executable. - Compiles the `hello.c` source file located in the `src` directory, generating the `hello` binary in the build directory (`$(PKG_BUILD_DIR)`).
- **`Package/hello-world/install`** section: - `$(INSTALL_DIR)`: Creates the target directory on the target device. - `$(INSTALL_BIN)`: Copies the compiled binary (`hello`) to the `/usr/bin` directory of the target device.
- **`$(eval $(call BuildPackage,hello-world))`**: - This macro starts the package building process using the specified rules.

Package Compilation

To compile the package, follow these steps:

1. Place the `hello.c` source file in the `package/hello-world/src/` directory.
2. Place the `Makefile` in the `package/hello-world/` directory.
3. Run the following command to compile the package:

   ```
   make package/hello-world/compile V=s
   ```

4. At the end of the process, the generated package will be in the `bin/packages/` directory, ready to be installed on the target device.

Installation and Testing on the Target Device

After being moved to the target device, the generated package (`hello-world.ipk`) can be installed using the `opkg` package manager. Example:

```
opkg install /tmp/hello-world.ipk
```

Once installed, the program can be run directly:

```
/usr/bin/hello
```

The output will be:

```
Hello, OpenWrt!
```

13.2 OpenWrt Build System

The OpenWrt build system is the heart of the process of generating custom firmware images and software packages. Based on **Buildroot**, a flexible and modular framework, the build system allows developers to configure, compile and assemble firmware for a wide range of embedded devices, including only the necessary components to optimize resources.

13.2.1 Build System Structure

The OpenWrt build system is organized into several directories and key files, each with a specific purpose:

- `target/`: Contains configurations for the different hardware architectures (e.g. ARM, MIPS, x86) and supported devices.
- `package/`: Contains the available software packages, each with its own `Makefile` to define sources, dependencies and build commands.
- `feeds/`: Manages external repositories containing additional packages, which can be easily integrated into the build system.
- `rules.mk`: Main file that defines global rules and variables for the build process.
- `scripts/`: Contains scripts used to automate common tasks, such as updating dependencies or cleaning temporary files.

This modular structure allows you to easily extend the functionality of the build system, adding new targets or packages without modifying existing files.

13.2.2 Build Process

The build process in OpenWrt consists of three main steps:

1. Configuration

The configuration of the build system is done via the `make menuconfig` command, which opens an interactive text interface. Here, the user can:

- Select the hardware architecture and target device.
- Enable or disable specific software packages.
- Customize the Linux kernel, choosing the modules to include.

After the configuration, the system saves the choices made in a file called `.config`, which will be used in the following phases.

2. Compilation

The compilation phase is managed by the `make` command, which performs the following operations:

1. Compilation of the **toolchain** for the target architecture, including compiler, linker and base libraries.
2. Compilation of the **Linux kernel**, customized according to the options specified during configuration.
3. Compilation of the enabled **software packages**, using the rules defined in their respective `Makefile`.

During this phase, the system also generates intermediate files, such as compilation logs and configuration files, which can be used for debugging.

3. Firmware Image Generation

Once the build is complete, the system generates a firmware image, which includes:

- The configured Linux kernel.
- The root filesystem (`rootfs`) containing the selected packages.
- Initial configuration files for the target device.

The generated image can have different formats, such as `squashfs` (read-only) or `ext4` (read-write), depending on the needs of the device.

13.2.3 Advanced Configurations

OpenWrt offers several advanced options to customize the build process:

- **Custom Images**: Users can create firmware images that include only the strictly necessary packages, reducing the overall size.

- **Custom packages**: You can add user-defined packages by creating a `Makefile` in the `package/` directory.
- **Parallel build**: Using the `-j<n>` flag, where `<n>` is the number of threads, you can speed up the build process by taking advantage of multi-core CPUs.
- **Updating feeds**: Package repositories (`feeds`) can be updated and synchronized with the command:

```
./scripts/feeds update -a
./scripts/feeds install -a
```

13.2.4 Practical Example: Creating a Firmware Image

Suppose we want to create a firmware image for a router with ARM architecture:

1. Download the OpenWrt source code:

```
git clone https://git.openwrt.org/openwrt/openwrt.git
cd openwrt
```

2. Configure the build system:

```
make menuconfig
```

   Select the target architecture (e.g. `ARM`) and enable the desired packages.

3. Start the compilation:

```
make -j4
```

4. Once completed, the firmware image will be available in the directory `bin/targets/<architecture>/<device>/`.

13.3 OpenWrt Key Features

OpenWrt does more than just provide a lightweight operating system for embedded devices, it also integrates a wide range of advanced features[19] that make it an ideal choice for complex network applications. In this section, we will describe some of the main features offered by the platform, with a focus on the technical aspects related to routing, firewall, QoS management, VPN support and wireless networks.

13.3.1 Advanced Routing

Routing is one of the central features of OpenWrt, allowing network devices to determine the optimal path for data transfer between sources and destinations. Routing protocols can be divided into two main categories: **static** and **dynamic**.

Routing Protocols: Static and Dynamic

Static routing protocols require routes to be manually defined by the network administrator. This approach is simple and suitable for small networks or networks with a stable topology, but it does not scale to complex environments, where changes in the network may require frequent updates.

Dynamic routing protocols, on the other hand, allow network devices to exchange route information and automatically update themselves based on the current topology. This flexibility makes them ideal for larger, dynamic networks, where routes may change frequently due to failures or traffic variations.

Protocols Supported by OpenWrt

With the integration of routing daemons such as `Bird`, `Quagga` and `FRRouting`, OpenWrt supports a wide range of dynamic routing protocols, including:

- **OSPF** (Open Shortest Path First): OSPF is a dynamic routing protocol based on a *link-state* algorithm, where each router maintains a complete map of the network and calculates the shortest paths using Dijkstra's algorithm. It is ideal for complex internal networks (*intradomain routing*) and supports advanced features such as load balancing over multiple paths (*equal-cost multipath routing*).

- **BGP** (Border Gateway Protocol): BGP is the primary routing protocol used on the Internet to connect autonomous domains (*Autonomous Systems*, AS). Unlike OSPF, BGP relies on a *path vector* algorithm and considers custom criteria, such as network policies and route attributes, to determine the best path. It is especially useful for managing enterprise networks with multiple connections to ISPs.

- **Babel**: Babel is a lightweight dynamic protocol designed for heterogeneous and mesh networks. It uses a hybrid algorithm that combines *distance-vector* and *link-state* techniques to ensure fast convergence and adaptability to unstable networks. Due to its lightweight and simplicity, Babel is often used in embedded scenarios.

Load Balancing Across Multiple WAN Connections

Load balancing is a technique that allows you to distribute network traffic across multiple WAN connections, improving the overall speed and reliability of the network. OpenWrt implements this functionality through the `mwan3` package, which allows you to manage multiple WAN interfaces and define rules for balancing traffic.

The operation of `mwan3` is based on:

- **WAN interface monitoring:** Each WAN interface is monitored in real time to verify its availability.
- **Traffic distribution:** Connections are distributed among the active WAN interfaces based on user-defined rules, such as weight (priority) or round-robin.
- **Custom traffic policies:** You can route specific types of traffic (for example, VoIP or streaming) through a preferred WAN connection.

With load balancing, OpenWrt allows you to make the most of available network resources, ensuring greater efficiency and reducing bottlenecks.

Automatic Failover

Automatic Failover is a complementary feature to load balancing, designed to ensure continuity of service in the event of a WAN connection failure. If a primary connection goes down, OpenWrt automatically redirects traffic to a secondary connection without interrupting communications.

In OpenWrt, this feature is also implemented via `mwan3`, which uses a monitoring system based on `ping` or DNS queries to detect connectivity issues. When a connection fails:

1. The affected WAN interface is marked as unavailable.
2. Traffic is automatically routed to an available alternate connection.
3. When the primary connection comes back up, traffic can be restored to the original connection.

Automatic failover is especially useful in enterprise or mission-critical environments where continuity of service is critical.

13.3.2 Firewall and Security

A firewall is one of the fundamental tools for protecting networks, and in OpenWrt it is a crucial component to ensure security and traffic control. A firewall acts as a filter between networks, allowing or blocking traffic based on predefined rules.

Role of a Firewall

A firewall performs several key functions within a network:

- **Access Control:** Prevents unauthorized users or devices from accessing internal resources.
- **Cyber Attack Protection:** Detects and blocks intrusion attempts or malicious attacks such as *port scanning* and *Denial of Service* (DoS).
- **Traffic Shaping:** Filters traffic based on protocols, IP addresses, ports, and other criteria, ensuring that only the intended connections are allowed.

- **Internal Network Isolation:** Protects specific segments of the network by preventing unauthorized communication between internal devices.

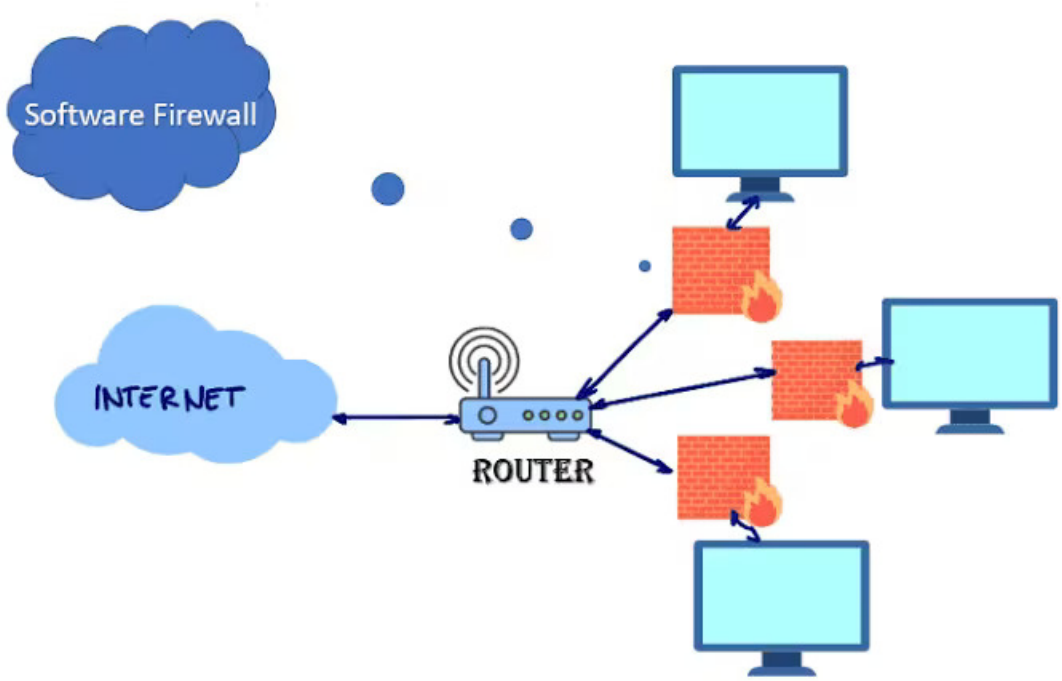

Figure 13.1. Use case of a firewall.

Based on `iptables` and `nftables`

OpenWrt's firewall system is traditionally based on `iptables`, a tool that allows you to configure and apply traffic filtering rules using a table structure. Each table contains a series of **chains**, which represent ordered sequences of rules. Each chain is associated with a specific phase of the packet flow through the device. The main chains are:

- `INPUT`, for packets destined for the device itself.
- `FORWARD`, for packets traversing the device, such as a router.
- `OUTPUT`, for packets originating from the device.

When a packet arrives at the device, it is analyzed by passing through the relevant chains based on its direction and purpose. Each rule in the chain can specify criteria such as source or destination IP address, protocol, and port, and define an action to take (for example, accept or reject the packet). For example, a rule in `iptables` to block TCP traffic to port 22 (SSH) might be:

```
iptables -A INPUT -p tcp --dport 22 -j DROP
```

This rule adds (`-A`) a rule to the `INPUT` chain, specifying that TCP packets destined for port 22 should be blocked (`-j DROP`).

In recent years, OpenWrt has introduced support for `nftables`, a more modern solution that is gradually replacing `iptables`[14]. Unlike its predecessor, `nftables` uses a single shared internal structure for all tables and chains, making the system more efficient and less affect by synchronization problems. `nftables` rules are written in a more concise and readable syntax. For example, the same rule to block SSH traffic with `nftables` would be:

```
nft add rule ip filter input tcp dport 22 drop
```

In this case, a rule is added to the `filter` table in the `input` chain, specifying to block (`drop`) TCP packets destined for port 22.

One of the key features of `nftables` is its ability to simplify rule management. While in `iptables` rules are distributed across separate tables, `nftables` allows you to combine multiple criteria and actions into a single rule, reducing complexity. Additionally, `nftables` is extensible and supports advanced features such as connection tracing and time-based filtering.

In OpenWrt, firewall configuration remains abstracted to the end user thanks to the `/etc/config/firewall` file, which uses a high-level format to define rules and policies. This file is automatically translated into `iptables` or `nftables` commands, depending on the backend used. This abstraction allows users to benefit from the advanced features of `nftables` without having to master its complex syntax.

Internal Network Isolation

One of the most powerful features offered by OpenWrt is the ability to isolate segments of the internal network. This feature is crucial for scenarios where you want to separate different types of traffic, such as:

- Create a separate network for guests, preventing them from accessing the main devices.
- Segment IoT traffic to protect the main network from potentially vulnerable devices.
- Configure VLANs (Virtual Local Area Networks) to assign each network segment its own isolated subnet.

In OpenWrt, isolation is implemented using VLANs combined with firewall rules. For example:

- VLANs are configured in the `/etc/config/network` file, assigning each VLAN a virtual interface.
- Firewall rules are configured to block traffic between VLANs, while maintaining isolation.

A practical example would be to configure one VLAN for guest Wi-Fi traffic and another for IoT devices. To ensure complete isolation, you would define two firewall rules, as follows:

```
config rule
option src 'guest'
option dest 'iot'
option target 'REJECT'

config rule
option src 'iot'
option dest 'guest'
option target 'REJECT'
```

These rules ensure that traffic between the `guest` and `iot` VLANs is blocked in both directions, maintaining complete isolation between devices on the two networks.

IDS/IPS Support

OpenWrt offers the ability to integrate intrusion detection and prevention systems (IDS/IPS) to further enhance security. Although **Snort** is included in the official packages, **Suricata** is not natively supported, but can be installed manually. These tools analyze traffic in real time and generate alerts for suspicious activity, with the possibility of directly blocking malicious traffic.

Configuring an IDS like Snort on OpenWrt requires:

- Installing the `snort` package via the `opkg` package manager.
- Configuring detection rules in the `snort.conf` file.
- Firewall integration to automatically block packets that trigger rules.

In the case of Suricata, installation requires manual cross-compilation, as it is not part of the official packages. Once installed, Suricata can be configured to analyze traffic on specific interfaces and generate logs in JSON format.

13.3.3 QoS Management and Traffic Shaping

QoS (*Quality of Service*) management and *traffic shaping* are fundamental techniques to ensure efficient use of network resources, especially in the presence of congested connections or devices competing for the same bandwidth.

QoS refers to the ability of a network to prioritize certain types of traffic, such as VoIP calls or video streaming, over less critical traffic such as file downloads or software updates. Traffic shaping, on the other hand, is the process of controlling the flow of data to prevent traffic from exceeding the maximum capacity of the network by regulating the rate at which packets are transmitted.

SQM (Smart Queue Management)

In OpenWrt, QoS management and traffic shaping are implemented primarily through `SQM (Smart Queue Management)`. SQM is an advanced framework that combines queue management algorithms to reduce latency, improve user experience, and ensure fairer bandwidth usage.

SQM is based on a concept known as *Active Queue Management* (AQM), where packets are dynamically inspected and organized before they are sent. This helps reduce excessive buffering, known as *bufferbloat*, a major cause of delays in congested networks. Bufferbloat occurs when routers or switches accumulate too many packets in their internal buffers, causing high delays and a degraded user experience.

With SQM, users can easily configure rules to limit the available bandwidth for each device or application, improving fairness and reducing overall latency.

CAKE and fq_codel Algorithms

The `CAKE` and `fq_codel` algorithms are two of the main mechanisms used by SQM to manage network traffic.

CAKE (Common Applications Kept Enhanced) `CAKE` is a queuing algorithm developed specifically for home and small business networks. It is designed to be easy to configure, reduce bufferbloat, and improve traffic fairness. Its key features include:

- **Fair Queueing:** Divides the available bandwidth equally among active flows, preventing a single device from monopolizing the network.
- **Bandwidth Shaping:** Allows you to set a maximum bandwidth limit, ensuring that traffic does not exceed the capacity of the connection.
- **DiffServ Integration:** Supports Differentiated Services (DiffServ) traffic classes, allowing you to prioritize certain types of packets.

A practical example of using `CAKE` is in a home network with multiple devices: thanks to fair queueing, each device gets a fair share of bandwidth, preventing a single download or stream from saturating the connection.

fq_codel (Fair Queue Controlled Delay) `fq_codel` is an AQM algorithm designed to reduce latency in congested networks by eliminating "old" packets from buffers and organizing traffic into distinct flows. Its main features include:

- **Latency Reduction:** Automatically identifies congested flows and applies drop policies to excess packets, keeping delays low.
- **Automatic Fairness:** Splits traffic into separate streams and ensures each stream receives a fair amount of bandwidth.
- **Simplicity:** fq_codel requires no complex configuration, making it an ideal choice for home and business networks.

For example, in a congested network during a video call and a large download, `fq_codel` ensures that the video call packets have a higher priority, maintaining quality of service.

SQM Configuration on OpenWrt

In OpenWrt, SQM configuration is simplified through the `luci-app-sqm` package, which provides an intuitive graphical interface. The user can define the main parameters, such as the maximum download and upload speed, and choose the preferred algorithm (`CAKE` or `fq_codel`). The associated configuration file is `/etc/config/sqm`.

An example configuration might include:

```
config queue 'eth1'
option interface 'eth1'
option download '100000'
option upload '20000'
option qdisc 'cake'
option script 'piece_of_cake.qos'
option enabled '1'
```

This configuration applies traffic shaping on the `eth1` interface, limiting the download speed to 100 Mbps and the upload speed to 20 Mbps using the `CAKE` algorithm.

13.3.4 VPN Support

A Virtual Private Network (VPN) is a technology that allows you to create an encrypted tunnel between two or more devices, ensuring the security and privacy of communications. VPNs are commonly used to securely access private network resources over the Internet, protecting the data transmitted from interception or manipulation. Through encryption, VPNs mask sensitive information such as IP addresses and browsing activity, making them an essential tool in public or untrusted networks.

Figure 13.2. Schema of a VPN.

In OpenWrt, setting up a VPN is made easy by the modularity of the platform, which supports numerous VPN protocols. Among these, the most used are OpenVPN, WireGuard and IPsec, each with specific characteristics that make it suitable for different scenarios.

OpenVPN

OpenVPN is an extremely versatile open-source solution for creating secure VPN connections. It is based on standard encryption protocols, such as TLS (Transport Layer Security), and supports various encryption algorithms such as AES (Advanced Encryption Standard). One of the main features of OpenVPN is its flexibility, which allows it to be configured in client-server or peer-to-peer mode.

OpenVPN uses TCP or UDP protocols to transmit data and offers the advantage of being able to easily pass through firewalls and NATs thanks to the use of port 443 (the same used by HTTPS traffic). However, this versatility can lead to more complexity in configuration compared to other protocols.

An example of configuration in OpenWrt is to create a `.ovpn` file containing credentials and connection parameters and upload it via the `luci-app-openvpn` package, which provides a graphical interface for easy management.

WireGuard

WireGuard is a modern VPN designed to be faster, simpler and more secure than traditional solutions. Unlike OpenVPN, WireGuard is implemented directly in the Linux kernel, ensuring superior performance and reduced latency. Its configuration is extremely

simple: it uses public and private key pairs to authenticate devices, eliminating the need for a complex certificate infrastructure.

WireGuard relies on modern cryptographic algorithms, such as Curve25519 for authentication and ChaCha20 for encryption, which ensure a high level of security. The lightweight nature of the protocol makes it ideal for embedded devices such as OpenWrt routers, where hardware resources may be limited.

In OpenWrt, WireGuard can be configured using the `luci-app-wireguard` package. An example configuration includes defining public and private keys and setting up VPN interfaces in the `/etc/config/network` file.

IPsec

IPsec (Internet Protocol Security) is a protocol widely used in enterprise environments to create secure connections between remote networks. Unlike OpenVPN and WireGuard, IPsec operates at the network layer (Layer 3 of the OSI model), making it suitable for configurations such as site-to-site (gateway to gateway) VPNs.

IPsec uses two main protocols to provide security:

- **AH (Authentication Header):** Ensures the integrity and authenticity of IP packets.
- **ESP (Encapsulating Security Payload):** Provides encryption, authentication, and protection against replay attacks.

IPsec is very robust and scalable, but its configuration can be complex due to the many options available, such as cipher type and authentication methods. In OpenWrt, the `strongSwan` package is commonly used to configure IPsec connections. This includes defining tunnels, pre-shared keys (PSK), or certificates to authenticate connections.

13.3.5 Wireless Network Support

One of the distinguishing features of OpenWrt is its ability to manage wireless networks in an advanced way, giving users unmatched flexibility in configuring and customizing Wi-Fi. OpenWrt allows you to optimize wireless networks for different scenarios, ensuring high performance, security and scalability.

Configuring Multiple SSIDs and VLANs

OpenWrt allows you to configure multiple SSIDs (*Service Set Identifier*), which are distinct Wi-Fi network names, on a single radio interface. This feature is particularly useful in contexts where you need to segment wireless traffic, such as:

- Create separate networks for guests, employees and IoT devices.
- Implement different access policies, ensuring greater security.

- Assign each SSID to a VLAN (*Virtual Local Area Network*), isolating devices on one network from each other. For example, a guest SSID can be assigned to a VLAN that blocks access to devices on the internal network.

Multiple SSIDs are configured using the `/etc/config/wireless` file, where each SSID can be associated with a separate logical interface. An example configuration is as follows:

```
config wifi-iface
option device 'radio0'
option network 'guest'
option mode 'ap'
option ssid 'GuestWiFi'
option encryption 'psk2'
option key 'guestpassword'
```

In this example, an SSID named `GuestWiFi` is created on the primary radio (`radio0`) and assigned to the VLAN `guest`.

Wireless Mesh Networks

OpenWrt supports wireless mesh networking, an ideal technology for extending Wi-Fi coverage in large spaces or areas that are difficult to wire. Mesh networks allow Wi-Fi nodes to communicate with each other dynamically, creating a scalable and resilient infrastructure.

With support for the `802.11s` standard, OpenWrt allows you to configure mesh networks natively. In a mesh configuration, each node acts both as an access point for wireless clients and as a forwarding point for traffic to other nodes in the network. This approach improves coverage and reduces bottlenecks, as traffic can be routed through alternate paths in the event of failures or congestion.

Advanced Protocol Support: `802.11r` and `802.11s`

OpenWrt provides support for advanced wireless protocols that improve the speed, mobility, and scalability of networks:

- `802.11r`: This protocol, known as *Fast Roaming*, allows wireless clients to quickly switch between multiple access points within the same network, minimizing downtime. It is especially useful in environments such as offices or campuses, where users frequently move between areas covered by different access points.

- `802.11s`: This standard defines a framework for creating wireless mesh networks, allowing multiple access points to dynamically collaborate to provide continuous Wi-Fi coverage. Mesh networks based on `802.11s` are self-configuring and self-healing, automatically adapting to changes in network topology.

Wireless Client Isolation

To ensure the security of shared Wi-Fi networks, OpenWrt includes features to isolate wireless clients. Isolation prevents devices connected to the same Wi-Fi network from communicating directly with each other, reducing the risk of attacks such as *ARP spoofing* or *packet sniffing*. This option is especially useful in public or guest networks.

Client isolation can be enabled for a given SSID by adding the following option to the `/etc/config/wireless` file:

```
option isolate '1'
```

With this configuration, devices connected to the same SSID will not be able to exchange packets, improving the security of the network.

13.3.6 Other Advanced Features

In addition to the features already described, OpenWrt includes a number of advanced tools and options that further expand its capabilities. These capabilities make it suitable for complex scenarios, where flexibility, scalability, and fine-grained control are essential.

Lightweight Containerization

OpenWrt supports containerization technologies such as `LXC` (Linux Containers) and, on more powerful hardware, `Docker`. Containerization allows you to run isolated applications in lightweight virtual environments, sharing the operating system kernel but maintaining separate user spaces.

`LXC` is particularly suitable for embedded devices, as it is designed to be lightweight and requires minimal resources. For example, you can create a `LXC` container to run a specific application, such as a DNS server or web server, without interfering with the main system.

`Docker`, on the other hand, offers greater flexibility to run modular and scalable applications, making it ideal for more powerful routers or OpenWrt-based devices used in enterprise environments. Containerization allows you to separate services, improve security, and simplify application management.

Captive Portal

A *captive portal* is a feature often used in public or commercial networks, such as hotels, airports, or coffee shops, to authenticate users before allowing access to the Internet. With OpenWrt, you can implement captive portals using packages such as `nodogsplash`, `CoovaChilli`, or `wifidog`.

A captive portal redirects users to an authentication page before granting access to the network, as shown in image 13.3. This page can be used to request credentials, collect usage policy acceptances, or integrate marketing features, such as data collection or serving advertisements.

Setting up a *captive portal* in OpenWrt involves:

- Creating a dedicated network for guest users, often isolated via VLANs.
- Installing and configuring the captive portal software.
- Integration with an authentication backend, such as a database or RADIUS server.

This feature is particularly useful in scenarios where you need to balance network access with security and control requirements.

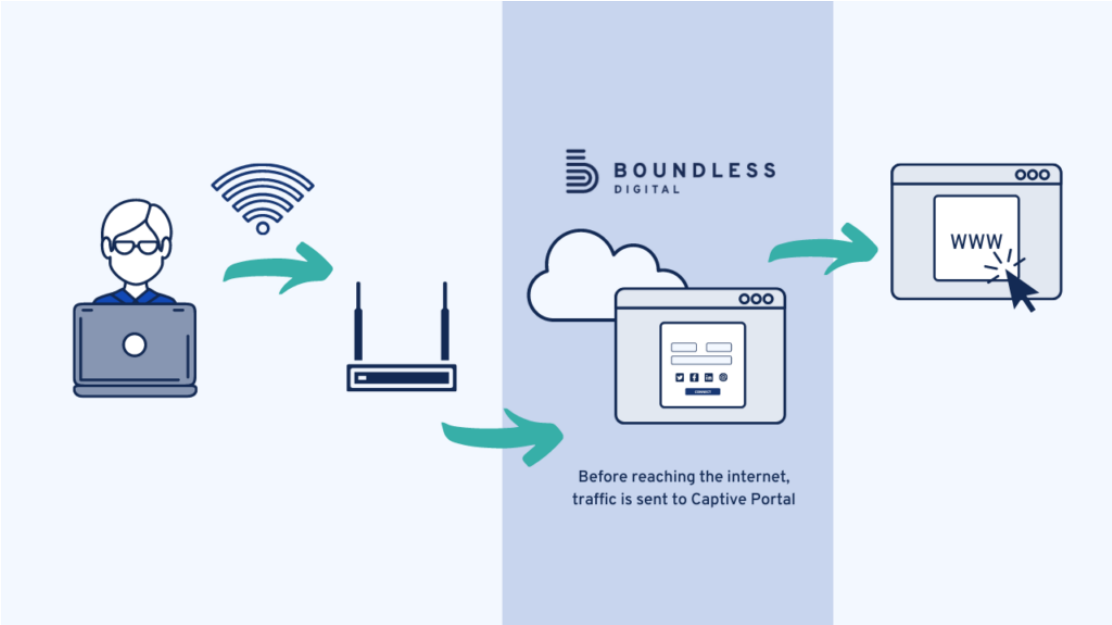

Figure 13.3. General schema of a captive portal.

Dynamic DNS

OpenWrt simplifies the setup of dynamic DNS services, allowing users to associate a static domain name with an IP address that can change over time. This is particularly useful for home networks or small businesses with Internet connections with dynamic IP addresses.

The OpenWrt `ddns-scripts` package allows you to set up a dynamic DNS service with providers such as `DynDNS`, `No-IP`, or others. A typical setup allows you to automatically update the public IP address associated with the domain whenever it changes, ensuring constant remote access.

An example `ddns-scripts` configuration might include:

```
config service 'myddns'
option service_name 'no-ip.com'
option domain 'mynetwork.ddns.net'
option username 'myusername'
```

```
option password 'mypassword'
option interface 'wan'
```

This configuration ensures that the Dynamic DNS service is always up to date, allowing remote access to devices or services on the internal network.

IPv6 Support

OpenWrt offers full support for IPv6, the network protocol designed to replace IPv4, due to its much larger address space. IPv6 introduces advanced features such as address autoconfiguration, hierarchical routing, and native support for IPsec security.

In OpenWrt, IPv6 can be configured for both WAN and LAN connections. Key features include:

- **Stateless Autoconfiguration (SLAAC):** Allows devices to automatically generate an IPv6 address based on the router's published prefix, simplifying network management.
- **DHCPv6 Support:** Allows dynamic configuration of IPv6 addresses and network options for devices in the LAN.
- **Advanced Routing:** OpenWrt supports IPv6 routing protocols such as OSPFv3 and BGP.

Part V

Integrating Suricata with OpenWrt: Compilation and Testing

Chapter 14

Porting Suricata to OpenWrt

Integrating Suricata with OpenWrt represents a significant technical challenge, due to the hardware limitations of embedded devices and the differences between a traditional Linux system and the OpenWrt environment. While Suricata is designed to run on complete systems with abundant CPU and memory resources, OpenWrt requires careful optimization to ensure efficient operation without compromising performance.

Initial efforts focused on determining whether Suricata could actually be ported to OpenWrt, initially using a virtual environment (VM). After successfully demonstrating the feasibility of this approach in the VM, it was decided to move forward with porting OpenWrt, along with Suricata, to real hardware.

This chapter describes the process of porting Suricata, both original and the optimized version, to OpenWrt 25.05.5, addressing dependency management, cross-compilation configuration, and adapting the code to be compatible with the OpenWrt environment. It will analyze the attempts made, the issues encountered, and the solutions adopted to ensure a stable and efficient integration.

14.1 The first attempt: compiling on OpenWrt

The first approach to run Suricata on OpenWrt was to compile it directly on the OpenWrt system already installed on a virtual machine in Virtual Box (link of the guide to do it on Virtual box). The goal was to see if a native installation could be achieved without having to resort to cross-compilation.

14.1.1 Initial setup and configuration

To start the build, we had to install a development environment on OpenWrt, including tools like `gcc`, `make` and basic dependencies. However, OpenWrt is designed to be a minimal system, with a small set of libraries and development tools. To fill these gaps, we tried to install the necessary packages via the `opkg` package manager.

14.1.2 Issues encountered

Despite adding the build tools, the build process was halted due to the lack of several key libraries required by Suricata, including `libpcre2-dev`, `libyaml-dev` and `libjansson-dev`.

Another challenge was the limited computational power of the OpenWrt device used for the compilation. Suricata is a complex software that requires an advanced compilation environment, and the limited processing power of the device would have made the process extremely slow and inefficient.

14.2 The Suricata package for OpenWrt

To integrate Suricata into OpenWrt, it was necessary to create a dedicated package with a specific **Makefile**, which defines the build process, dependencies and necessary configurations. This section analyzes the fundamental parts of the Makefile, explaining their role in the build process.

14.2.1 Package definition and code source

The first part of the Makefile establishes the package name, version and repository from which to get the source code:

```
PKG_NAME:=suricata
PKG_VERSION:=8.0.0
PKG_RELEASE:=1

PKG_SOURCE_PROTO:=git
PKG_SOURCE_URL:=https://github.com/guca11/SuricataIDS-onlyCode.git
PKG_MIRROR_HASH:=skip
```

14.2.2 Dependency Management and Build Configuration

Suricata requires several libraries to run properly. These dependencies are defined in the Makefile with:

```
PKG_BUILD_DEPENDS:=rust/host python3/host
DEPENDS:= +libexpat +jansson +libbpf ...
```

- **PKG_BUILD_DEPENDS** specifies the dependencies needed for the build, including support for **Rust** and **Python**.
- **DEPENDS** specifies the packages required for the run, such as `libpcap` for network packet capture, `libyaml` for configuration file management, and `libnss` for security management.

In addition, the Makefile includes several build options including:

```
CONFIGURE_ARGS += --target=$(RUSTC_TARGET_ARCH) \
--host=$(RUSTC_TARGET_ARCH) \
--build=$(RUSTC_HOST_ARCH) \
```

- **CONFIGURE_ARGS** contains options for the configuration process, including:
  - `-target, -host, -build`: Defines the build and target architectures.
  - `-disable-sctp, -disable-gre, -disable-nfs, ...`: Disables unnecessary protocols to reduce memory usage and improve performance.

14.2.3 Build and Installation Process

The Makefile defines the compilation and installation process through the following rules:

Configuration Phase

```
define Build/Configure
( \
$(CONFIGURE_VARS) cargo install --force --root $(STAGING_DIR)/host cbindgen ; \
cd $(PKG_BUILD_DIR) && $(CONFIGURE_VARS) ./scripts/bundle.sh ; \
cd $(PKG_BUILD_DIR) && $(CONFIGURE_VARS) ./autogen.sh &&
$(CONFIGURE_VARS) ./configure $(CONFIGURE_ARGS) ; \
)
$(call Build/Configure/Default)
endef
```

These commands run:

- `cargo install` to install **cbindgen**, a Rust tool for generating C bindings.
- `./scripts/bundle.sh` to prepare the source code for compilation.
- `./autogen.sh` and `./configure` to configure the build with the specified options.

Installation Phase

```
define Package/suricata/install
$(INSTALL_DIR) $(1)/usr/bin
$(INSTALL_BIN) $(PKG_INSTALL_DIR)/usr/bin/suricata $(1)/usr/bin/suricata
$(INSTALL_BIN) $(PKG_INSTALL_DIR)/usr/bin/suricatactl $(1)/usr/bin/suricatactl
$(INSTALL_BIN) $(PKG_INSTALL_DIR)/usr/bin/suricatasc $(1)/usr/bin/suricatasc
$(INSTALL_BIN) $(PKG_INSTALL_DIR)/usr/bin/suricata-update $(1)/usr/bin/suricata-update

$(INSTALL_DIR) $(1)/usr/lib
$(CP) -r $(PKG_INSTALL_DIR)/usr/lib/* $(1)/usr/lib/
```

This step:

- Copy Suricata binaries and its management tools (`suricatactl`, `suricata-update`) to the `/usr/bin` directory.
- Install the necessary libraries to the `/usr/lib` directory.
- Copy the essential configuration files to `/etc/suricata`.

14.3 The second attempt: cross-compilation with the SDK

After realizing that direct compilation on OpenWrt was impractical, a more structured approach was chosen: cross-compilation using the official OpenWrt SDK (Software Development Kit). This method would allow Suricata to be compiled on a more powerful machine, generating a package compatible with OpenWrt.

14.3.1 SDK setup and compilation environment

To start the cross-compilation process, the OpenWrt SDK for the desired target architecture was downloaded. The development environment was configured on a virtual machine with `Ubuntu 24.10`, installing the necessary dependencies and the SDK.

Subsequently, the Suricata package was added to the packages to be compiled.

14.3.2 C library and toolchain issues

One of the main obstacles that arose during the compilation was the fact that the OpenWrt SDK comes pre-configured with a limited set of libraries and development tools, excluding some compatibility extensions. When trying to compile with the SDK, several errors occurred due to missing functions in `musl`[18], including:

- `fopen64` – Used for handling large files, not defined in musl.
- `fstat64` – Function to get information about files, not supported in musl with the same interface as glibc.

14.4 Third attempt: Rebuilding OpenWrt from scratch

After encountering compatibility issues between Suricata and the OpenWrt SDK toolchain, it was decided to completely rebuild OpenWrt from scratch using the Buildroot system, with a custom configuration that included all the dependencies required by Suricata.

14.4.1 Build System Setup

To start the OpenWrt rebuild, the official source code of the project was downloaded and then the environment was configured with the command:

```
make menuconfig
```

Inside the configuration interface, the following options were selected to ensure compatibility with Suricata and optimize the system for use on a virtual machine:

- **BPF Toolchain** → `Build LLVM toolchain for eBPF`
  Enables support for eBPF (Extended Berkeley Packet Filter), useful for advanced traffic analysis and to improve network performance.
- **Libraries** → `libintl-full`
  Includes the full version of the `libintl` library, which is required for advanced localization support and string handling in programs.
- **Compile with Full language support**
  Ensures that the system supports multiple programming languages, which is necessary since Suricata is based on multiple languages.
- **Kernel build options** → `Enable XDP sockets support`
  Enables support for XDP (eXpress Data Path) sockets, a framework optimized for high-performance network packet handling.
- **Target Images** → `GZip images`
  Disables GZip compression of firmware images, reducing overhead when starting the VM because the image doesn't have to be decompressed.
- **Target Images** → `Build VirtualBox image files`
  Generates a VirtualBox compatible image.

Once the configuration was complete, we started the build with:

```
make -j$(nproc)
```

14.4.2 Building and Image Generation

The build took several hours, as Buildroot had to download and build all the packages from scratch, including the kernel, libraries, and system tools. At the end of the process, we generated a custom firmware image that was used to create a virtual machine.

14.4.3 Verifying Suricata Integration

After booting the VM, we verified that Suricata was up and running by running the executable with:

```
suricata
```

The output confirmed that the Suricata build was correctly compiled and ready to run in an OpenWrt environment.

14.5 Implementing OpenWrt on a Raspberry Pi 3

After successfully porting Suricata to OpenWrt for the `x86_64` architecture, the next goal was to adapt the solution to run on a real embedded device, specifically the **Raspberry Pi 3 Model B**. This device, based on the `ARM aarch64` architecture, offers a great test platform to evaluate the performance of Suricata in resource-constrained environments.

14.5.1 Raspberry Pi 3 Model B Hardware Specifications

The **Raspberry Pi 3 Model B** is an embedded system with the following features:

- **Processor**: `Broadcom BCM2837` ARM Cortex-A53 quad-core @ 1.2 GHz.
- **RAM**: 1 GB LPDDR2 SDRAM.
- **Storage**: MicroSD for OS and data.
- **Networking**: 10/100 Mbps Ethernet port, 2.4 GHz Wi-Fi 802.11 b/g/n.

The 64-bit **Cortex-A53** processor is an improvement over previous models, but the limited amount of RAM (1GB) and 100 Mbps Ethernet limit running advanced network monitoring applications like Suricata. Also, using the microSD card as the main storage can create bottlenecks in logging operations.

14.5.2 Build environment setup

To generate an OpenWrt image compatible with Raspberry Pi 3, the `Buildroot` system was used again.

The steps to follow are the same as previously seen with the difference of the following options in the configuration menu:

- **Target System** → `Broadcom BCM27xx` (to support the Raspberry Pi family).
- **Subtarget** → `BCM2710` (specific for Raspberry Pi 3).

Figure 14.1. The Raspberry Pi 3

14.5.3 Building and image generation

Once the system was configured, the build was started with:

```
make -j$(nproc) V=sc
```

After several hours, once the firmware image was generated, it was necessary to write it to a microSD card to boot the device. For this operation, **BalenaEtcher** was used, a simple and effective graphical software for writing images to storage media.

BalenaEtcher offers several advantages, including:

- **Intuitive graphical interface**, which reduces the risk of errors in selecting the target device.
- **Automatic integrity verification** of the written image, to avoid boot problems due to errors in copying.

After writing the image, the SD card was inserted into the Raspberry Pi 3 and the device successfully booted, loading OpenWrt.

14.5.4 `suricata-lua-sys` library and cross-compilation issue

One of the main bugs encountered when porting Suricata to Raspberry Pi 3 was related to the **suricata-lua-sys** library, a component that enables integration between Suricata and the `Lua` scripting language. This library is used to run Lua scripts directly within Suricata, allowing users to customize threat detection and log management in an advanced way. However, its configuration during the build process presented some issues, mainly related to toolchain management.

The original `Makefile` for `suricata-lua-sys` used the following commands to build:

```
CC= gcc -std=gnu99
AR= ar rcu
RANLIB= ranlib
```

This setup was problematic for cross-compiling, as it referenced the **host** machine's toolchains instead of those intended for the target architecture. As a result, the build process generated an executable that was only compatible with the architecture of the machine used for compilation (e.g., `x86_64`) and not with that of the Raspberry Pi 3 (`ARM aarch64`).

To fix this, it was necessary to modify the `Makefile` to explicitly use the target toolchain's compiler and archiver:

```
CC?= gcc
AR?= ar rcu
RANLIB?= ranlib
```

This way, if the CC, AR and RANLIB variables are already defined in the build environment, the assigned values will be used, ensuring that they are specific for the build target.

14.6 Auto-configuration of Suricata based on network traffic

One of the main problems of an IDS like Suricata is to find a balance between efficiency and accuracy. Analyzing all supported protocols can provide more complete monitoring, but at the cost of increased resource usage. To solve this problem, a **dynamic auto-configuration** system was developed, capable of adapting Suricata's behavior based on the protocols actually detected in network traffic.

The idea behind this solution is simple: start Suricata in a monitoring phase, collect the generated logs and extract information about the protocols used. If a protocol is never detected or rarely used, its analysis is superfluous and can be disabled to reduce the computational load. Conversely, if a protocol is present with a significant frequency, it is kept active. Once the relevant protocols are determined, the system automatically changes the configuration and recompiles Suricata to reflect the new settings.

14.6.1 Autoconfiguration script implementation

Suricata autoconfiguration is done using this Bash script that automates the process of detecting active protocols on the network and updating the software configuration based on the data collected.

Suricata scouting the network

The script starts Suricata running on the OpenWrt router, for a certain period of time. During this time, Suricata analyzes the network traffic and logs the detected events to the `eve.json` log file.

After a configurable time interval (`TIME`), Suricata is stopped and the log file is transferred to the local system for analysis.

Log analysis and configuration generation

The `eve.json` log file is analyzed by a Python script that counts the frequency of the observed protocols. If a protocol exceeds a predefined threshold (`THRESHOLD`), it is enabled in the Suricata configuration; otherwise, it is disabled.

After that, this Python script produces a set of flags in the `config_flags.txt` file, each corresponding to a protocol to enable or disable. For example, if the SSH protocol has been detected frequently, the resulting flag will be:

```
--enable-communication
```

Conversely, if a protocol such as DNP3 is rarely present in traffic, it will generate:

```
--disable-dnp3
```

Makefile editing and recompilation

Once the flags are generated, the script updates the `Makefile` of the Suricata OpenWrt package, inserting the new configuration options in the `CONFIGURE_ARGS` section. Afterwards, the Suricata package is recompiled with the new parameters and finally transferred to the remote device and installed.

14.6.2 Log parsing and settings generation

The parsing of the `eve.json` log file is handled by a script written in Python, which scrolls through the content line by line and identifies the protocols present. In particular, the `event_type` field of each event indicates the corresponding protocol. The script maintains a counter for each protocol and, at the end of the processing, compares these values with the defined threshold. For example, if HTTP traffic was detected many times, the script generates the `--enable-http` flag, while if the IKE protocol was absent, or detected less times than the threshold, the `--disable-ike` flag is generated.

A special case concerns protocols that do not have a direct entry in the `event_type` field, but can be detected within other events. For example, the script checks for the DCERPC protocol by analyzing the content of the SMB logs, since DCERPC can be transported via SMB. If detected, the script automatically enables DCERPC support.

The result of the analysis is saved to a temporary file, from which the Bash script reads the generated flags and adds them to the OpenWrt Makefile. The entire process is fully automated, avoiding the need for manual configuration.

Chapter 15

OpenWrt Performance with Suricata

Analyzing Suricata performance on OpenWrt is essential to evaluate the impact of the changes introduced and understand the real capabilities of the system in a resource-constrained environment. The goal of these tests is to measure CPU and memory consumption in realistic traffic scenarios, analyzing different Suricata configurations and their behavior under stress.

The tests were run using **Kali Linux 2024.4** as a traffic generation machine and an external network interface, the **AWUS1900** by Alfa Network, to overcome the limitations of the integrated network card. The choice of an external USB network card is motivated by the fact that many network interfaces integrated in the test devices have limitations in terms of packet management, especially in high-traffic scenarios or in the presence of DoS attacks.

These tests were performed using **Suricata in three different configurations** and they were run on both **Wired Ethernet** and **Wireless** connections to evaluate the performance differences between the two connection modes.

The obtained results allow to better understand the trade-off between **detection accuracy and resource consumption**, helping to determine the optimal configuration for embedded scenarios.

15.1 Resource Monitoring

To evaluate the resource consumption of Suricata on OpenWrt, we used the `top` command, a process monitoring utility that provides detailed information about CPU, memory, and running processes.

This command allows you to monitor system activity in real time, displaying running processes and their impact on hardware resources. It was used during testing to observe the behavior of Suricata in different traffic scenarios, checking the variations in CPU and RAM consumption.

An example of the output of the `top` command on OpenWrt is as follows:

```
Mem: 50456K used, 32156K free, 1120K buffers
CPU: 43.0% usr 5.0% sys 0.0% nic 50.0% idle
PID PPID USER STAT VSZ %CPU COMMAND
1354 1123 root R 243m 42.3% suricata
```

From the output you can get information like:

- **Memory Usage**: The `Mem` section shows the total amount of used and available memory.
- **CPU Usage**: The `CPU` line shows the percentage of CPU usage split between user (`usr`), system (`sys`), and idle (`idle`).
- **Active Processes**: Running processes with details about their virtual memory (`VSZ`) and CPU load (`%CPU`).

Furthermore, to evaluate the RAM consumption of Suricata, the command was used

```
cat /proc/$(pgrep suricata)/status | grep VmRSS
```

This command allows you to read the file `/proc/<PID>/status`, containing detailed information about the process, and then extract only the line related to the "Resident Set Size" (`VmRSS`), i.e. the amount of physical memory currently used by the process.

15.2 DoS Attacks

A **Denial of Service (DoS)** attack is a type of cyber attack that aims to make a system or network unusable by overloading it with requests or exhausting its available resources. This type of attack was simulated to evaluate Suricata's capabilities to detect malicious activity and the impact on the OpenWrt device's resources.

To simulate a DoS attack, the `tcpreplay` tool was used:

```
tcpreplay -i wlan1 --mbps=100 log.pcap
```

The `tcpreplay` command is used to resend a network packet capture file (`.pcap`) through a specified network interface:

- **-i wlan1**: Specifies the network interface to transmit packets through.
- **–mbps=100**: Sets the maximum packet transmission rate.
- **log.pcap**: Specifies the log file to replay.

15.2.1 SYN Flood Attacks

A **SYN Flood** attack is a type of Denial of Service (DoS) attack that exploits the TCP handshake process to exhaust the resources of the target system. The normal TCP handshake process consists of three phases:

1. The client sends a `SYN` packet to the server to initiate the connection.
2. The server responds with a `SYN-ACK` packet.
3. The client responds with a `ACK` packet, completing the handshake.

In SYN Flood attacks, the third step never completes, leaving the server with numerous pending connections, exhausting available resources and preventing new legitimate connections.

To simulate this type of attack, the `hping3` tool was used with the command:

```
hping3 --flood -S -p 80 192.168.1.1
```

Where:

- `-flood`: Send packets as fast as possible.
- `-S`: Send packets with the `SYN` flag set.
- `-p 80`: Attacks the server's port 80 (HTTP).
- `192.168.1.1`: Attack target (OpenWrt router).

15.3 System Performance

System performance tests were conducted using the log file previously used to evaluate Suricata performance modified with the `tcprewrite` command to change the destination IP. Tests were performed on two types of connections:

- **Ethernet Connection**, to measure performance in a wired environment with lower latency and higher stability.
- **Wireless Connection**, to evaluate how bandwidth limitations and increased latency affect System's resource consumption.

To evaluate the performance of the various configurations during the tests, the data transmission speed in Mbps and the number of packets per second (in Kpps) of the logs sent with `tcpreplay` were measured both via Ethernet and Wireless connection. The table 15.1 reports the value obtained for the different types of traffic.

Traffic Type	Ethernet		Wireless	
	Speed	**Packets**	**Speed**	**Packets**
Mix of protocols	94	26	29	8
DNS	85	92	11	12
UDP	91	45	12	6
HTTP	98	11	48	5

Table 15.1. Transmission speed in Mbps and packet rate in Kpps for Ethernet and Wireless connections.

For each Suricata configuration, measurements of the system's idle percentage, Suricata's RAM and CPU consumption were collected.

15.3.1 Snort Performance

Since Snort is an IDS already available on OpenWrt, it is useful to compare its performance with Suricata to evaluate its efficiency in environments with limited resources. Snort, while idle, consumes about 175.5 MB of RAM.

Traffic type	Ethernet			Wireless		
	% Idle	**RAM**	**% CPU**	**% Idle**	**RAM**	**% CPU**
Mix of protocols	57.7%	258.2 MB	26.8%	64%	221.9 MB	15%
DNS	58%	183.6 MB	28.9%	50%	183.6 MB	21.8%
UDP	54.1%	194.7 MB	27.8%	66.3%	191.9 MB	13.5%
HTTP	73.7%	217.9 MB	17.4%	73%	215.5 MB	9.9%
SYN Flood Attack	41.8%	500 MB	26.7%	62.8%	500 MB	18%

Table 15.2. System performance with Snort

15.3.2 Suricata Full Configuration

This configuration enables all protocols supported by Suricata, ensuring a complete analysis of network traffic, and it consumes, in idle, approximately 84.9 MB.

The following table shows the resource consumption under heavy load for different types of traffic:

Traffic type	Ethernet			Wireless		
	% Idle	RAM	% CPU	% Idle	RAM	% CPU
Mix of protocols	61.9%	102 MB	21.2%	75.9%	92.2 MB	6.7%
DNS	15.9%	87.4 MB	71.4%	46%	90.7 MB	27.3%
UDP	48.4%	94 MB	30.1%	74.5%	92.4 MB	5.5%
HTTP	81.4%	91.5 MB	11.2%	81.9%	91.1 MB	5.4%
SYN Flood Attack	29.2%	129.9 MB	41.8%	63.4%	129.9 MB	16.1%

Table 15.3. System performance in Full configuration

15.3.3 Network flag only configuration

In this configuration, Suricata only scans the main network protocols (IPv4, TCP, UDP, ICMPv4) and has an idle memory consumption of about 80.6 MB.

Traffic type	Ethernet			Wireless		
	% Idle	RAM	% CPU	% Idle	RAM	% CPU
Mix of protocols	65%	95.1 MB	20.2%	73.8%	88.3 MB	6.9%
DNS	16%	86.9 MB	71.6%	47.6%	87.1 MB	26%
UDP	53%	88 MB	25.5%	73.2%	87.7 MB	5.7%
HTTP	80.4%	87.5 MB	10.3%	79.2%	87.5 MB	4.3%
SYN Flood Attack	29.7%	126.9 MB	40.5 %	63.7%	126.9 MB	15.7%

Table 15.4. System performance with Suricata with Network flag only

15.3.4 Configuration without any optional protocols

This configuration excludes all optional protocols, leaving only the minimum modules for traffic analysis active. Suricata's idle memory consumption is about 79.3 MB.

Traffic type	Ethernet			Wireless		
	% Idle	RAM	% CPU	% Idle	RAM	% CPU
Mix of protocols	69.6%	92.7 MB	18.7%	71.3%	86.5 MB	7.3%
DNS	45.9%	82.4 MB	38.2%	65.1%	82.4 MB	7%
UDP	59%	83.9 MB	20.5%	74.5%	83.6 MB	5.1%
HTTP	81.3%	85.9 MB	10.4%	81%	85.5 MB	4.1%
SYN Flood Attack	31.8 %	124 MB	39.2%	65.8%	124 MB	14.3%

Table 15.5. System performance with Suricata without optional protocols

15.3.5 Analysis of results

Performance analysis of Snort and different Suricata configurations revealed significant differences in CPU usage, RAM usage, and system responsiveness.

In terms of resource consumption, Snort was more demanding than Suricata, with a higher average RAM consumption. However, Snort showed more stability in CPU usage in heavy traffic scenarios.

Comparing different Suricata configurations, it is clear that protocol selection impacts performance. The **Full Suricata** configuration, with full support for all protocols, showed relatively low RAM consumption (around 102 MB in Ethernet and 92 MB in Wireless), but with significantly higher CPU usage, especially for DNS traffic, where CPU peaks exceeded 70

The **network-flag-only Suricata** configuration reduced CPU consumption without significantly sacrificing detection capability. Memory consumption was slightly reduced compared to the full version, with a smaller impact on the system. However, the CPU gain was marginal compared to the minimal configuration.

In the **Minimal Suricata** version, which excluded all optional protocols, the RAM consumption was the lowest (about 86 MB in wireless mode) and the CPU percentage used was the lowest in almost all scenarios. This shows that customizing Suricata configurations allows for an efficient IDS even on devices with limited resources, adapting to the specific needs of the network.

An interesting aspect is the impact of HTTP traffic, which in all configurations recorded a lower CPU consumption than the other types of traffic. This can be attributed to the numbers of packets send in the same second. On the contrary, DNS and UDP traffic imposed a higher load, probably due to the high number of smaller packets, compared to HTTP, that need to be processed individually.

Finally, the analysis of **SYN Flood Attacks** showed that these attacks generate a significant increase in resource consumption, with an increase in CPU usage and a decrease in the percentage of system idle. This is to be expected, since a SYN Flood attack generates a high number of incomplete connections, increasing the work required to monitor them.

Overall, the tests show that Suricata's modularity offers a significant advantage, allowing to balance security and performance based on the specific requirements of the system and network.

Bibliography

[1] Asmaa Shaker Ashoor and Sharad Gore. Importance of intrusion detection system (ids). *International Journal of Scientific and Engineering Research*, 2(1):1–4, 2011.

[2] Akram Abd Eldjalil Boukebous, Mohamed Islem Fettache, Gueltoum Bendiab, and Stavros Shiaeles. A comparative analysis of snort 3 and suricata. In *2023 IEEE IAS Global Conference on Emerging Technologies (GlobConET)*, pages 1–6. IEEE, 2023.

[3] Waleed Bulajoul, Anne James, and Mandeep Pannu. Network intrusion detection systems in high-speed traffic in computer networks. In *2013 IEEE 10th International Conference on e-Business Engineering*, pages 168–175, 2013. doi: 10.1109/ICEBE.2013.26.

[4] Jeorgithon Damasceno, Jamilson Dantas, and Jean Araujo. Network edge router performance evaluation: An openwrt-based approach. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2022. doi: 10.23919/CISTI54924.2022.9820027.

[5] David Day and Benjamin Burns. A performance analysis of snort and suricata network intrusion detection and prevention engines. In *Fifth international conference on digital society, Gosier, Guadeloupe*, pages 187–192, 2011.

[6] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999. ISSN 1389-1286. doi: https://doi.org/10.1016/S1389-1286(98)00017-6. URL https://www.sciencedirect.com/science/article/pii/S1389128698000176.

[7] Suricata Docs. About dpdk, . URL https://docs.suricata.io/en/latest/capture-hardware/dpdk.html#dpdk.

[8] Suricata Docs. About suricata rules, . URL https://docs.suricata.io/en/latest/rules/index.html.

[9] Gustavo González-Granadillo, Susana González-Zarzosa, and Rodrigo Diaz. Security information and event management (siem): Analysis, trends, and usage in critical infrastructures. *Sensors*, 21(14), 2021. ISSN 1424-8220. doi: 10.3390/s21144759. URL https://www.mdpi.com/1424-8220/21/14/4759.

[10] The Tcpdump group. About libpcap. URL https://github.com/the-tcpdump-group/libpcap/blob/master/README.md.

[11] Qinwen Hu, Se-Young Yu, and Muhammad Rizwan Asghar. Analysing performance issues of open-source intrusion detection systems in high-speed networks. *Journal of Information Security and Applications*, 51:102426, 2020. ISSN 2214-2126. doi: https://doi.org/10.1016/j.jisa.2019.102426. URL https://www.sciencedirect.com/science/article/pii/S2214212619306003.

[12] Kire Jakimoski and Nidhi V Singhai. Improvement of hardware firewall's data rates by optimizing suricata performances. In *2019 27th Telecommunications Forum (TELFOR)*, pages 1–4, 2019. doi: 10.1109/TELFOR48224.2019.8971192.

[13] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2012.09.004. URL https://www.sciencedirect.com/science/article/pii/S1084804512001944.

[14] Praveen Likhar and Ravi Shankar Yadav. Impacts of replace venerable iptables and embrace nftables in a new futuristic linux firewall framework. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, pages 1735–1742, 2021. doi: 10.1109/ICCMC51019.2021.9418298.

[15] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3): 103–104, October 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL https://doi.org/10.1145/2692956.2663188.

[16] Florian Menges, Tobias Latzo, Manfred Vielberth, Sabine Sobola, Henrich C. Pöhls, Benjamin Taubmann, Johannes Köstler, Alexander Puchta, Felix Freiling, Hans P. Reiser, and Günther Pernul. Towards gdpr-compliant data processing in modern siem systems. *Computers and Security*, 103:102165, 2021. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2020.102165. URL https://www.sciencedirect.com/science/article/pii/S0167404820304387.

[17] Adabi Raihan Muhammad, Parman Sukarno, and Aulia Arif Wardana. Integrated security information and event management (siem) with intrusion detection system (ids) for live analysis based on machine learning. *Procedia Computer Science*, 217:1406–1415, 2023. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.12.339. URL https://www.sciencedirect.com/science/article/pii/S1877050922024243. 4th International Conference on Industry 4.0 and Smart Manufacturing.

[18] Musl. musl removed some functions. URL https://git.musl-libc.org/cgit/musl/commit/?id=246f1c811448f37a44b41cd8df8d0ef9736d95f4.

[19] OpenWrt. About openwrt features, . URL https://openwrt.org/docs/guide-user/start.

[20] OpenWrt. About openwrt minimum requirements, . URL https://openwrt.org/supported_devices/864_warning.

[21] OpenWrt. About openwrt supported devices, . URL https://openwrt.org/toh/start.

[22] Zeek Project. About zeek. URL https://docs.zeek.org/en/master/about.html.

[23] Syed Ali Raza Shah and Biju Issac. Performance comparison of intrusion detection systems and application of machine learning to snort system. *Future Generation Computer Systems*, 80:157–170, 2018. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.10.016. URL https://www.sciencedirect.com/science/article/pii/S0167739X17323178.

[24] Ana Paula Vazão, Leonel Santos, Rogério Luís de C. Costa, and Carlos Rabadão. Implementing and evaluating a gdpr-compliant open-source siem solution. *Journal of Information Security and Applications*, 75:103509, 2023. ISSN 2214-2126. doi: https://doi.org/10.1016/j.jisa.2023.103509. URL https://www.sciencedirect.com/science/article/pii/S2214212623000935.

[25] Abdul Waleed, Abdul Fareed Jamali, and Ammar Masood. Which open-source ids? snort, suricata or zeek. *Computer Networks*, 213:109116, 2022. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2022.109116. URL https://www.sciencedirect.com/science/article/pii/S1389128622002420.