



**University of
Zurich^{UZH}**

Security Management and Visualization in a Blockchain-based Collaborative Defense

*Christian Killer
Baden, Switzerland
Student ID: 12-738-118*

Supervisor: Bruno B. Rodrigues, Eder Scheid
Date of Submission: December 21, 2018

Zusammenfassung

Die stetig zunehmende Gefahr von Distributed Denial of Service (DDoS) Angriffen führt zu einem erhöhten Interesse an Gegenmassnahmen, sowohl in Forschung als auch Entwicklung. Ein dezentraler Ansatz um DDoS Angriffe einzudämmen basiert auf der Zusammenarbeit von mehreren autonomen Systemen. Das Blockchain Signaling System (BloSS) ist ein dezentrales, multi-domain, blockchain-basiertes und kooperatives DDoS Verteidigungssystem, wo jedes autonome System Teil in der Verteidigungsallianz teilnimmt. Die autonomen Systeme tauschen über die Ethereum Blockchain Angriffsinformationen aus. Die aktuelle Implementation von BloSS ist jedoch nicht interaktiv und die DDoS Abwehr wird automatisiert ausgeführt. In Realität würde jedoch ein menschlicher Analyst die Entscheidungen treffen, ob ein DDoS Angriff nun verteidigt oder unterbunden wird. Um das BloSS interaktiv zu evaluieren ist also ein Tool nötig, dass von einem Analysten einfach und effizient verwendet werden kann. Diese Arbeit beschreibt das Design und die Implementation einer grafischen Benutzeroberfläche für BloSS. Die Architektur dieser Benutzeroberfläche beinhaltet drei Hauptkomponenten: **bloss-core**, **bloss-node** und **bloss-dashboard**. Die **bloss-core** Komponente ist eine erweiterte und modifizierte Version der Arbeit von Mannhart [17] und Rodrigues et al. [25]. Das Hauptziel der Erweiterungen der **bloss-core** Komponente waren die Kommunikationsinterfaces mit **bloss-node**, welches eine neu erstellte Server Komponente ist, die sich um das State Management und weitere zentrale Aufgaben hat. Die **bloss-dashboard** Komponente ist eine webbasierte Single-Page Application (SPA) die in (beinahe) Echtzeit aktualisiert wird und dem Analysten ermöglicht mit dem BloSS zu interagieren.

Abstract

The increasing amount of Distributed Denial of Service (DDoS) attacks leads to a growing research and development interest in DDoS defense systems. One approach to fend off DDoS attacks is decentralized systems that require the collaboration among different Autonomous Systems (ASes). BloSS is a multi-domain, blockchain-based, cooperative DDoS defense system, where each autonomous system (AS) is taking part in the defense alliance. Each AS is able to exchange attack information about ongoing attacks via the Ethereum blockchain. The current implementation of BloSS is not interactive or visualized and DDoS mitigation is automated. However, in real world defense systems, a human cyber security analyst decides whether a DDoS threat should be mitigated or not. To enable the interactive evaluation of a cooperative defense, an interactive tool is necessary. This work outlines the design, implementation and evaluation of an interactive security management and visualization tool for BloSS. The architecture of this work consists of three components: **bloss-core**, **bloss-node** and **bloss-dashboard**. The component **bloss-core** is an extended and modified version of previous work by Mannhart [17] and Rodrigues et al. [25]. The main purpose of the extensions of **bloss-core** were implemented with the goal to enable communication with **bloss-node**, which is a server component taking care of state management and multiple communication tasks. The **bloss-dashboard** component is a web single-page application (SPA) that offers (near) real-time updates of the on-going processes and enables a human cyber security analyst to react on an individual threat level.

Acknowledgements

I would like to thank Bruno Rodrigues for his support and guidance during the thesis and Stephan Mannhart for his insights on the BloSS implementation details. I would also like to thank Raphael Matile for discussions on software architecture and engineering and another special thanks to Stefania Ventre for endless support, guidance and encouragement. A final thanks to my parents for their tireless support.

Contents

Zusammenfassung	i
Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Motivation	2
1.2 Description of Work	3
2 Related Work	5
2.1 DoS and DDoS	5
2.2 Defense Systems	6
2.3 Blockchain Signaling System	6
2.4 Cyber Situation Awareness	7
2.4.1 Network Security Situation Awareness Model	8
2.5 Security Visualization	8
2.5.1 DDoS Visualizations	9
2.5.2 Dashboards	9
2.6 Visualization Guidelines	10
2.6.1 Security Visualization Standard (SCeeL-VisT)	10
2.6.2 Security Visualization Effectiveness Measurement (SvEm) Theory .	11
2.6.3 Christopher Alexander's fifteen properties	12

3	Design	13
3.1	Requirements	13
3.2	Architecture	15
3.2.1	BloSS Dashboard	16
3.2.2	BloSS Node	17
3.2.3	BloSS Core	18
3.3	BloSS Workflows	19
3.3.1	Target AS perspective	19
3.3.2	Mitigator AS perspective	21
3.4	State Management	21
3.4.1	Mitigator Perspective	22
3.4.2	Target AS Perspective	23
4	Implementation	25
4.1	BloSS Hardware	25
4.2	BloSS Management	27
4.3	BloSS Core	27
4.3.1	Configuration	27
4.3.2	BloSS	28
4.3.3	Pollen Module	29
4.3.4	Stalk Module	29
4.4	Bloss Node	30
4.4.1	Configuration	31
4.4.2	Communication interfaces	32
4.4.3	Data management	32
4.4.4	State management	34
4.4.5	Service Management	35
4.5	BloSS Dashboard	36

<i>CONTENTS</i>	ix
4.5.1 Front-end stack	37
4.5.2 Components	37
4.5.3 Detail Modal Dialog	39
4.5.4 Responsive Design	40
5 Evaluation	43
5.1 Functional Requirements	43
5.1.1 UC1: Request Mitigation or Ignore Alarm (Target’s perspective) . .	44
5.1.2 UC2: Accept or Decline Mitigation Request (Mitigator’s perspective)	45
5.1.3 UC3: Start or stop services	46
5.2 Non-functional requirements	46
6 Conclusion	49
6.1 Future Work	50
Bibliography	53
Abbreviations	55
List of Figures	55
List of Tables	58
A Contents of the CD	61
B Use Case Figures	63

Chapter 1

Introduction

Cybersecurity has globally risen to one of the top priorities in both research and development [15]. One major topic in cybersecurity is Denial-of-service (DoS) attacks. These attacks aim to exhaust the target's resources through high-load traffic or by using malicious packets created to generate high computing loads [33]. As these attacks are typically performed on massive scales using distributed computing resources, they are commonly referred to as Distributed Denial of Service attacks (DDoS) [34]. Recent security incident trend statistics show a steady increase in the number of DDoS attacks worldwide [1]. Not only the number of attacks is growing, but also the number of long-duration attacks (*e.g.*, the most extended attack was longer than 12 days [1]).

One approach to defend against DDoS attacks is cooperative defense systems, where an alliance of entities collaborates. The entities have the capability to control Internet traffic and aim to create a globally distributed defense against DDoS attacks. The defense capacity of a single entity is extended by sharing the protection resources with all other entities. For instance, if a highly distributed and sophisticated attack happens, it is possible to request protection from other systems that may offer the ideal solution for the detection and mitigation of this attack. One example of a collaborative defense system is the Blockchain Signaling System (BloSS). BloSS is a multi-domain, blockchain-based, cooperative DDoS defense system, where each autonomous system (AS) is taking part in the defense alliance [25]. The exchange of attack information about ongoing attacks is handled via the Ethereum blockchain.

Organizations mainly rely on cybersecurity analysts to make critical decisions regarding threats and network security [27]. The exceptional perceptual capabilities of humans can be improved by providing cybersecurity analysts with appropriate visualizations and management tools [27] to process large data volumes and recognize trends and patterns visually [8]. This work aims to provide a security management tool visualizing the process of a DDoS defense scenario, applied to BloSS.

In the context of the Internet, where a growing number of stationary and portable devices communicate daily, the analysis of any data without an adequate form of visualization is a complex task. Before visualizing data, it is necessary to structure and categorize data so that its visualization makes sense to the analyst [16]. Because of the large amount of

data that is generated by the devices, organizing and selecting an adequate visualization is challenging. There are various types of visualizations for the same situation. Selecting the appropriate type of representation is not a straightforward decision. The selection introduces multi-disciplinary concepts and the decision-making process usually requires a low response time from the user [16].

Data visualization facilitates the communication of a large amount of information [18]. The processing of text is harder than the processing of images or pictures, which can be processed extremely well. Visualizations result in increased bandwidth between the information and the viewer [18]. In other words: *A picture is worth a thousand log records* [18]. Understanding the essence of something is more comfortable when it is aggregated in the form of visualization.

In order to apply visualization to the field of computer security, knowledge of different disciplines needs to be combined, namely security and visualization. The dichotomy of security visualization is that most people who are trying to visualize security data know the data and its meaning, but do not necessarily understand the main design principles of visualization [18]. Therefore it is essential to apply best-practices from visualization theory *e.g.*, the Visual Information Seeking Mantra of Ben Shneiderman [26]: Overview first, zoom and filter, then details-on-demand. It is necessary to grasp the situation and its possible outcomes to get an overview. For example, it is required to understand a situation before and during a specific event in order to understand the changing situation without knowing the exact details of this event. *Zoom and filter* relate to the detailed analysis of the event and the filtering of unrelated events that can introduce unwanted noise into the analysis. Details are only provided on demand [12].

This thesis investigates and proposes a solution to the problem of visualization and security management in the cooperative defense system BloSS.

1.1 Motivation

As DDoS become more frequent it is important to take countermeasures and deploy a defense system. There are centralized and decentralized defense system against DDoS. The disadvantage of a centralized defense approach is that the system becomes infeasible due to limited resources available to the autonomous system (AS) targeted by the attack. A decentralized defense system approaches the threat differently and can, therefore, mitigate the risk of becoming infeasible. The primary challenge of decentralized approaches is the difficulty in coordinating and communicating among ASes. Therefore, a human decision-maker (*e.g.* analyst) needs to decide whether a threat is serious and define a course of action. The analyst needs an appropriate overview of all the attack-relevant data and an appropriate security management tool is required. Fully automated systems that defend themselves and gain self-awareness without human analysts are still in the distant future. In conclusion, human decision-makers are still indispensable to make decisions [3]. Therefore it is important to visualize attacks for analysts so that they can make efficient decisions. Visualization also facilitates the ability to process large data volumes and present trends and patterns visually [8].

The BloSS is a collaborative, multi-domain and decentralized defense system that collaboratively approaches defense. Nevertheless, an analyst is still needed to make the decisions and choose the right behavior. In order that the analyst understands what happens with his system, a visualization of the DDoS attack and the possible decisions have to be displayed. The goal of this work is to create an interactive, easy-to-use Security Management tool for BloSS.

1.2 Description of Work

This thesis is divided into two main parts. The first part contains the research of related work and exploration of the current implementation of the Blockchain Signaling System (BloSS). The objective is to understand the operational processes and the communication among the different BloSS modules. It is necessary to consider the research and documentation of current approaches concerning security visualization and management, including cooperative defenses. In the second part of this thesis, the modification and extension of the current BloSS (hereafter referenced as **bloss-core**) are described and explained. Subsequently, the implementation of the security visualization tool **bloss-core** manageable by an analyst will be demonstrated. The source code of this work is available on github¹.

¹<https://github.com/blockchain-signaling-system>

Chapter 2

Related Work

Security is an important concern in any modern system available on the Internet. Consequently, there is a wide array of research in the most diverse fields of cybersecurity. The goal is to keep new and existing systems updated and secured.

The following chapter introduces the concept and prior work of the Blockchain Signaling System (BloSS). Furthermore, related work will be reviewed in the different fields adjoining the area of Security Management and Visualization in connection with a blockchain-based collaborative defense.

2.1 DoS and DDoS

One of the most straight-forward definitions of Denial of Service is provided by the International Telecommunications Union (ITU-T) recommendation X.800 [24]:

denial of service: *The prevention of authorized access to resources or the delaying of timecritical operations.*

Generally, DoS attacks cause direct or indirect resource exhaustion to the target through the generation of specific traffic [34]. The resource exhaustion causes an interruption on service availability, thus pose a threat to security and dependability [34].

In the real world, most of those attacks are performed on a massive scale using a large number of devices scattered across the globe. Therefore, DoS attacks are generally referred to as distributed denial of service attacks (DDoS) (*i.e.* each DDoS is a DoS) [34]. A distributed denial of service (DDoS) attack is a special variant of the generic DoS attack [24].

Attacker hosts are often a group of compromised computers which can be controlled by a *master* entity [24]. The attackers are called *bots* or *zombies* and a collection of such hosts is called a *botnet*. Typically, a DDoS attack is launched when a *botnet* is orchestrated by the *master* to direct traffic towards a single specified victim [24].

2.2 Defense Systems

Defense systems are much harder to implement than attack tools [34]. Traditionally, DDoS defense includes two main perspectives: source- and destination-based mitigation [33]. Source-based mitigation is when the defense system of an AS detects malicious outgoing traffic and blocks it. Destination-based mitigation, however, is based on incoming traffic analysis filtering malicious traffic before reaching the AS [33]. Since both mechanisms do not include the cooperation among different defense systems, they are considered to be centralized defense approaches [33].

DDoS attacks are orchestrated in a highly distributed manner, purely centralized defense approaches fall short in successfully defending networks. Hence, combining both source- and destination-based approaches with a degree of cooperation among defense systems can oppose the distributed threat.

The Internet Engineering Task Force (IETF) Distributed-Denial-of-Service Open Threat Signaling (DOTS) architecture [21] outlines standardization attempts for a decentralized defense systems. The DOTS protocol enables intra- and inter-organizational DDoS defense with multiple parties. The goal of the DOTS protocol is to aid with the coordination of attack responses using a client-server model [21]. Detecting an attack, the client requests mitigation from the DOTS server. Further, there is the differentiation of manual and automatic mitigation requests. Manual mitigation requests are manually demanded by the DOTS client operator, while automatic requests are executed with respect to pre-defined network conditions [21].

2.3 Blockchain Signaling System

Because DDoS defense systems are not adapted to resist large-scale attacks, research efforts by Rodrigues et al. [25] led to the inception of the Blockchain Signaling System (BloSS). Since existing DDoS signaling protocols often suffer from bottlenecks to coordinate a distributed defense, a blockchain-based system can offer new ways to reduce the complexity of signaling the attack information. Moreover, it also encourages cooperation and reduces the operational costs [25].

Prior work by Rodrigues et al. [25] successfully showcased the practical feasibility of BloSS with the help of a physical hardware system. The hardware system consists of three Software-Defined Network (SDN) based ASes [25]. The ASes are also interconnected to the Proof-of-Authority (PoA) Ethereum blockchain through a dedicated, congestion-free management network channel. Each of the ASes is connected to five hosts, each with static IP addresses. The hardware system showcased an automated DDoS defense process. As soon as one or multiple hosts is under attack, the AS of the attacked host reports this to the PoA blockchain. The AS of the attacking hosts then retrieves the attack report and accepts the DDoS mitigation request by blocking the traffic through SDN blackholing.

The initial efforts of Rodrigues et al. [25] nourished follow-up work by Gruhler [11] and Mannhart [17]. Gruhler designed and evaluated a reward system for cooperative DDoS

defense. Different existing reputation mechanisms and incentive-based mechanisms were evaluated and led to a prototype [10] of a reputation API inspired by the Work.nation project [31]. The proposed reputation system allows participants to identify constant lazy, selfish and honest behavior, by evaluating historical binary reputation values.

To advance the development of the BloSS system, Mannhart [17] modularized the existing codebase of Rodrigues et al. [25], increased stability and enabled easier future development. The **bloss-core** module further described in section 3.2.3 is based on prior work by Mannhart [17]. The implementation of an interactive dashboard for **bloss-core** led to multiple modifications described in Chapter 3.

2.4 Cyber Situation Awareness

The ever-increasing threat of cyber attacks emphasizes the importance of Situation Awareness (SA) in the context of modern networking infrastructure [28]. Generally, SA is best described as follows [4]:

Situation awareness is the perception of the elements of the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future.

Prior work declares that the idea of SA is not restricted to any particular domain but can directly be applied to the cyber domain [28]. There are different challenges to reach Cyber Situation Awareness (CSA) for a decision maker. It is key to identify and model the relevant activities of interest to understand what is necessary to reach CSA.

According to Barford et al. [3], CSA encompasses at least seven aspects:

1. Awareness of **current situation** or also called situation perception which also includes both situation recognition and identification. Situation identification can include identifying the type of attack, while recognition is acknowledging that an attack occurs.
2. Awareness of **impact of attack** or also called impact assessment which includes the assessment of the current impact and assessment of future impact.
3. Awareness of **evolution**: Be aware of how the situation evolves which includes tracking the situation.
4. Awareness of **adversary behavior**: A crucial factor of this facet is the attack trend and analysis of intent.
5. Awareness of **causality**: Be aware of how and why the current situation is caused. This aspect includes causality analysis, back-tracking and the use of forensics.
6. Awareness of **trustworthiness and quality**: Be aware of the quality and genuineness of the collected situation awareness information and the decisions derived from that information.
7. Awareness of **plausible futures**: This involves a multitude of technologies for projecting future possible actions/activities of an adversary.

These seven aspects are crucial when developing and designing a tool to use in cybersecurity. In the context of security visualization, CSA can be actively improved by leveraging visualizations to process large data volumes and present trends and patterns visually [8].

2.4.1 Network Security Situation Awareness Model

Another interesting framework concerning CSA is the *Network Security Situation Awareness Model* (NSSAM) [16]. The multi-level model proposes four phases [16]:

1. **security events generated**, various security devices generate scattered security events and log.
2. security incidents **formatting, filtering and sorting**
3. **quantitative calculations** of security situation index
4. **security situation visualization**, ultimately the security situation presented to the operator

According to the NSSAM, each phase increases the network situational awareness. [16].

2.5 Security Visualization

Organizations rely on cybersecurity analysts to make critical decisions regarding threats, vulnerabilities, and overall network security [27]. Fully automated systems that defend themselves and gain self-awareness without human operators are still in the distant future. Hence, prior work regards human decision makers as indispensable [3]. The unique perceptual capabilities of humans are combined with algorithmic support to enable a better understanding of complex data [27].

Thus, security visualizations associate and display security incidents to cybersecurity analysts by providing interactive visual experiences [8]. Visualization also facilitates the ability to process large data volumes and present trends and patterns visually [8]. Thus, security visualization has come up as an effective tool to better equip analysts to operate effectively in an evolving digital threat landscape [27] and ultimately achieve CSA.

Most research in cybersecurity focuses on data analysis and not the appropriate presentation of that data. Besides that, there are no widely established frameworks for evaluation available [19]. User-centered evaluation is not a priority for most works [20]. Generally, research in visualization either leads to adaption of existing visualization techniques to the cybersecurity domain, or it leads to novel ways to specifically visualize cybersecurity data [19].

Contributions in cybersecurity research range from theoretical work establishing the basic foundations in diverse fields until algorithms based on theoretical work to implement practical tools. [6].

2.5.1 DDoS Visualizations

DDoS visualizations are a special kind of security visualization. Visualizations of distributed denial of service (DDoS) attacks vary from representations for non-technical audiences [14] to visualizations specifically designed for cyber-security analysts [19].

Most visualizations in the context of DDoS aim to digest detailed attack data and transform the data into a visual format optimized for cybersecurity analysts. Of course, there is not a perfect visualization system due to the subjective nature of the human perception process [32].

2.5.2 Dashboards

Dashboards are commonly used to analyze data in (near) real-time. The focus for real-time monitoring is to understand the current state of a system, ongoing tasks or events of interest [18]. The following definitions help to classify dashboards and differentiate various types of dashboards.

According to [5], a dashboard can be described as:

A dashboard is a visual display of information needed to achieve one or more objectives which fit entirely on a single computer screen so it can be monitored at a glance.

Further, dashboards can be classified as operational, tactical or strategical according to [18].

- **Operational:** A dashboard to track core processes, metrics and status in real-time.
- **Tactical:** A dashboard to help monitoring processes of networks, states of machines and to help analyzing exception conditions and summarize data to analyze.
- **Strategic:** A dashboard to help monitoring the execution of strategic objectives. The aspect of real-time information is secondary and relaxed.

According to [13], there are various factors to consider when designing dashboards. First, it is important to not overload the dashboard with visual features and to make prominent visual features to show important messages. Therefore visual features should not be put in corners of the dashboard but rather in the center. [13]. Further, a dashboard should *not be an art show* but be designed within constraints, even though they still should be pleasant to view [13]. Hence, it is important to have constraints, guidelines, standards and best practices at hand when designing dashboards. Further, there should not be unnecessary decorations, or overused colors and other visual properties [18]. Moreover, a single screen should be used, while there should not be more than five elements on the screen to digest [18].

2.6 Visualization Guidelines

In order to develop visualizations to be more effective and appealing, it is necessary to measure how fitting the visualization is [7]. The correct interpretation of visualizations from users without a firm technical background is a frequent concern in the cybersecurity field. The accurate interpretation is crucial and time-sensitive in security visualizations.

The Security Visualization Standard (SCeeL-VisT) tries to tackle this problem with a systematic two-phase approach [7]. Another interesting theory is the Security Visualization Effectiveness Measurement (SvEm) Theory. The SvEm theory encapsulates a theoretical formula to evaluate the effectiveness of all processes of security visualizations. To contrast the theoretical approaches of SCeeL-VisT and SvEm, the third subsection discusses Alexander's fifteen properties (AFP) and covers visualizations from a totally different angle.

2.6.1 Security Visualization Standard (SCeeL-VisT)

The Security Visualization Standard has two main phases, the understanding phase and the visualization phase [7]. For a proper visualization, the requirements need to be accurately evaluated by an expert with domain knowledge.

1. Understanding what is in Hand (Dataset)
 - (a) **The Problem: Identification of Security Event**
 - Identify Security Event (e.g. Malware Attack, SQL Injection, etc.) & Data Type (Raw data: log files, Social media data, etc.)
 - Understand the nature of data (e.g. financial data, Health data. etc.)
 - (b) **The Purpose: Know the Visualization Type and Technique**
 - Understand the intension of security Visualization (e.g. show relationships, etc.)
 - Decision: Exploratory or Reporting Visualization
 - Decision: Security visualization Technique (e.g. Categorizing: timebase, Provenance-base attack-base)
 - (c) **Cyber Attack Landscape: Know the cyber-attack location (e.g. Network, Systems, Application layer, etc.)**
 - Know the point of attack (e.g. network attack, identify source and destination of attack, etc.)
 - Attribution of cyber-attack
2. Security Visualization Process
 - (a) **Visual Presentation Methodologies: How to present data visually**
 - (b) **Color & Shape Standard for Security: Decision on choice of colors**
 - Standardizing Main Color choices
 - Standardizing Main Shapes choices

The open and generic nature of SCeeL-VisT makes it a suitable, universal framework for cybersecurity visualization.

2.6.2 Security Visualization Effectiveness Measurement (SvEm) Theory

Through analysis of existing techniques and understanding their functionality, prior research introduced a new metric to measure the effectiveness in both a given security visualization and the experiences on security events [7]. This metric is called the Security Visualization Effectiveness Measurement (SvEm) Theory [7].

$$SvEm = \frac{(w * h) / S_{V_f} * d_n}{Cl * t_{me} * n_{clicks}} > 50\%(Distortion)$$

- **w * h**: Mobile Display Area (dimensions) of the surface is used to display a security visualization
- **S_{V_f}**: Security Visual Nodes (e.g. Infected-IP, Timestamp, etc.) which are known security attributes identified in a visualization
- **d_n**: n-dimensional view of security visualization, refers to how many visual dimensions are used to represent the visualization. If a higher number of dimensions n is used, that's an indicator for the depth of data being able to represent visually
- **Cl**: Cognitive Load (Identifiable Attributes (Quantity)—Prior Knowledge), which is based on how much prior knowledge a user has on the visualization that is presented
- **t_{me}**: Memory efficiency (Effort based on Working memory—Time-base), which is an attribute which measures how quickly one can recall a security-related attribute
- **n_{clicks}**: Number of clicks on visualization that have to be performed in order to view the required visualization

The overall aim of the SvEm theory is to reduce the duration that is spent on the visual processing of a visualization. The theoretical concept behind the SvEm theory is mainly based on the following two aspects [7]:

- **Visualization Distortion**: Effectiveness in Distortion is defined when a visualization has greater than 50% visual clarity
 - How clear and readable visualization is presented
 - Features and Attributes presented are visible
 - Visual patterns emerge into reality to observers.
- **User Response Time**: Duration in milliseconds (ms)
 - Analytical time of processing the visualization
 - Time of user- cognition to recall known memory

In order to apply the SvEm theory, the assumptions are summarized as follows [7]:

- If a user is able to understand a given security visualization within less than 5 s, then visualization has effective SvEm output.
- The user always needs some form of prior knowledge before engaging the given security visualization.

- The number of clicks (nclicks) refers to the number of clicks(navigating) on the mobile platform screen to the point where the first known security attribute has been identified.

Without evaluating and applying theories and standards, they remain abstract and far from practice. Hence, prior research applied an extended SvEm Theory (adding the SvEm Theory for Time Assessment) to mobile and web visualizations and evaluated the effectiveness across the entire security visualization process, including graphical representation and user interaction [8]. Prior work concluded that implementing such systematic approaches captured the viewer’s attention and increased the user’s attention span [8].

In conclusion, the SvEm theory does not always need to be applied precisely with numerical input, but rather it can offer a good guideline and properties to keep in mind when optimizing the effectiveness of visualizations.

2.6.3 Christopher Alexander’s fifteen properties

A rather holistic and *out-of-the-box* approach is the collection of Alexander’s fifteen properties (AFP) which are believed to be functionally and visually appealing [2].

The inspiration for the AFP is derived from extensive research in chemistry, physics, biology, and engineering. Hence, it is argued that these properties are suitable to improve analytical reasoning and to enable more robust evaluations [2].

Prior research formed twelve archetypes by applying the AFPs to the visualization domain [2]. Depending on the type of visualization method, those archetypes can help as guidelines for best-practices and can serve as an orientation for visualizations.

Chapter 3

Design

This chapter explains the design rationales of an interactive Security Management and Visualization tool for the Blockchain Signaling System (BloSS).

BloSS is a multi-domain, blockchain-based, cooperative DDoS defense system, where each autonomous system (AS) is taking part in the defense alliance. Each AS exchanges attack information about ongoing attacks via the Ethereum blockchain. As illustrated in figure 3.1, the attack information is shared with the AS of the subnet where the attacks originate from. These ASes either mitigate or decline the requests for mitigation.

Prior to this work, attack reporting and attack mitigation was carried out based on a constant configuration and could not be decided upon an individual threat level. Human decision-makers remain indispensable in the near future. Therefore, the design of an interactive management tool for a human analyst enables the interactive evaluation and showcases a cooperative defense [3]. The following section discusses the design decisions for the architecture of an interactive visualization tool for BloSS.

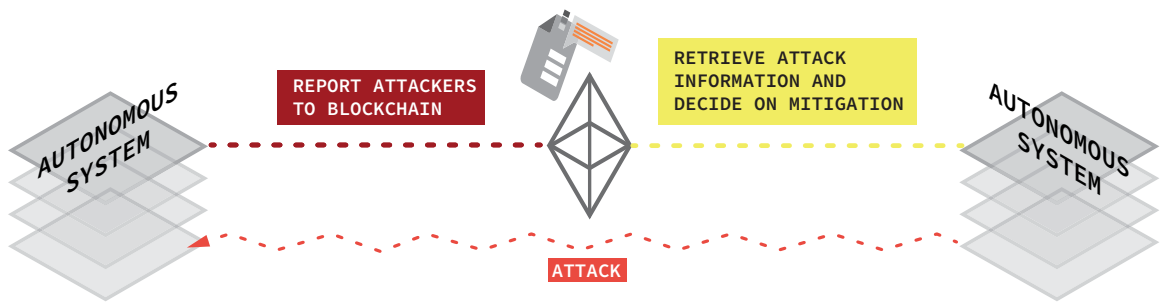


Figure 3.1: BloSS collaborative defense system

3.1 Requirements

This section describes the elicited requirements of an interactive Security Management tool for BloSS. The primary stakeholder is the AS operator, which ideally is a cyber

security analyst. The analyst is either in the position of a *mitigator* or a *target*. The terminology depends on the context of the DDoS attack.

The main functional requirements are as follows:

- **F_R1:** A target should be able to request mitigation or ignore an alarm
- **F_R2:** A mitigator should be able to accept or decline a mitigation request
- **F_R3:** An analyst should be able to check service statuses and start or stop them

The requirements **F_R1** to **F_R3** directly translate to the following use cases, graphically depicted in the use case diagram 3.2.

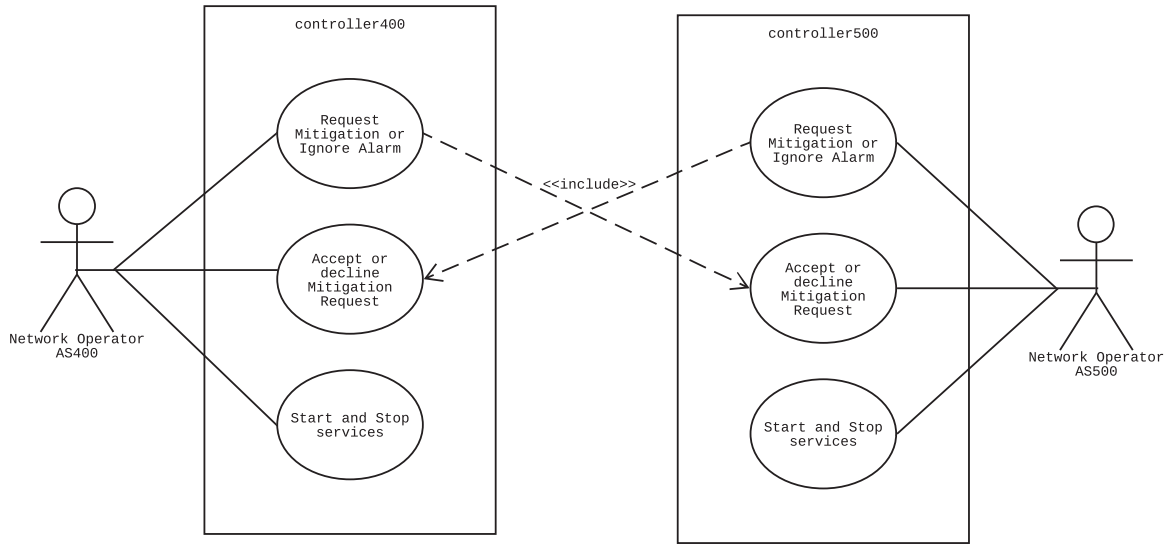


Figure 3.2: Use case diagram with example AS400 and AS500

The main non-functional requirement is:

- **NF_R1:** concerns *Usability*, the dashboard should be easy to manage from a web browser and also from mobile devices.

Usability requirements such as **NF_R1** are challenging to evaluate. For this work, the GNOME developer design principles are applied to optimize usability [29]. The design principles are straightforward and to the point: *the best applications provide an elegant solution to a specific task area.*[29].

The following section describes the architecture of a system that aims to fulfill the requirements elicited above.

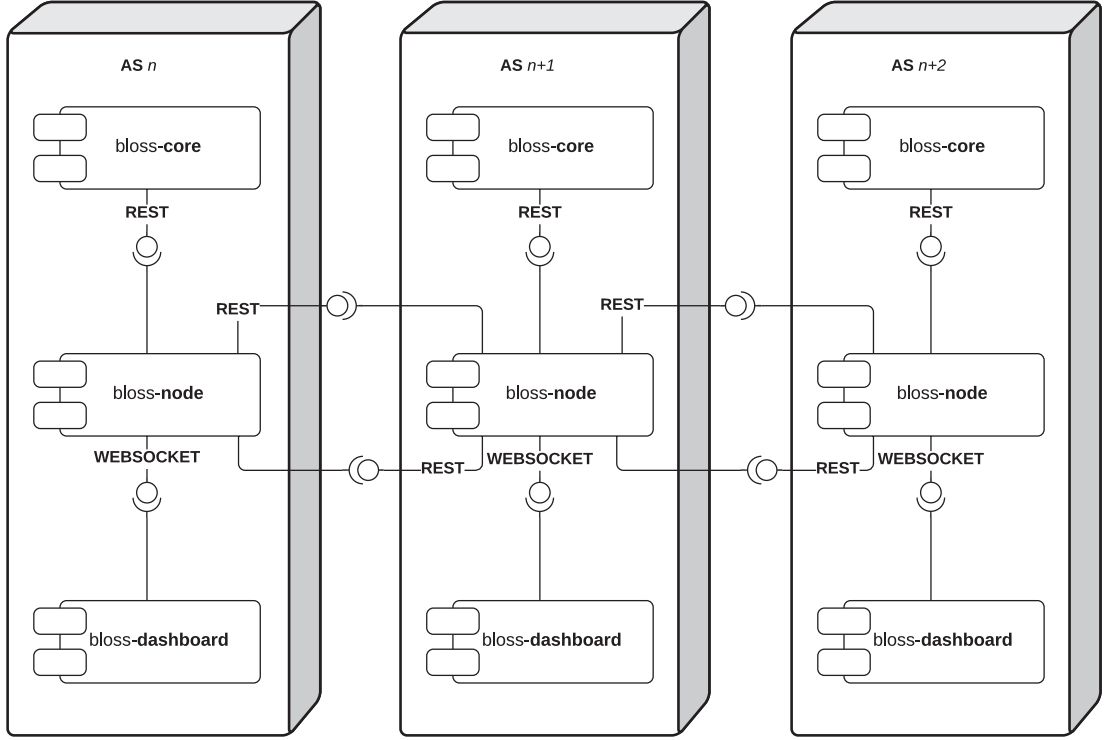


Figure 3.3: Component Diagram of the BloSS architecture

3.2 Architecture

This section discusses the chosen architecture for an administrative dashboard in the context of a collaborative defense. As indicated in Figure 3.3 there are three main components. The figure also depicts three identical ASes to illustrate the communication among different ASes.

bloss-core: The **bloss-core** component monitors and manages the network traffic of the underlying infrastructure and takes care of the communication with the Ethereum blockchain, the InfluxDB and also communicates with **bloss-node**. The RESTful interface is used to exchange information (*e.g.* **attack_reports**, blocking status, traffic breaches) with **bloss-node**. Design details of **bloss-core** are discussed in subsection 3.2.3.

bloss-node: Essentially, the **bloss-node** component acts as a relay server for communication between **bloss-core** and **bloss-dashboard**. To achieve (near) real-time communication, a WebSocket interface is used to exchange data with the front-end **bloss-dashboard**. Additionally, **bloss-node** handles state management of **attack_reports** and has a RESTful interface for direct communication with other instances of **bloss-node** running on other ASes. Design details of **bloss-node** are discussed in subsection 3.2.2.

bloss-dashboard: The **bloss-dashboard** component is a front-end dashboard displaying relevant information for the human analyst. The **bloss-dashboard** is implemented as a single-page application (SPA) and attached to **bloss-node** via WebSockets. Details of the User Interface (UI) of the **bloss-dashboard** are discussed in subsection 3.2.1.

The following subsections describe these components in more detail, highlighting the conceptual design choices and discuss the major tasks of each component.

3.2.1 BloSS Dashboard

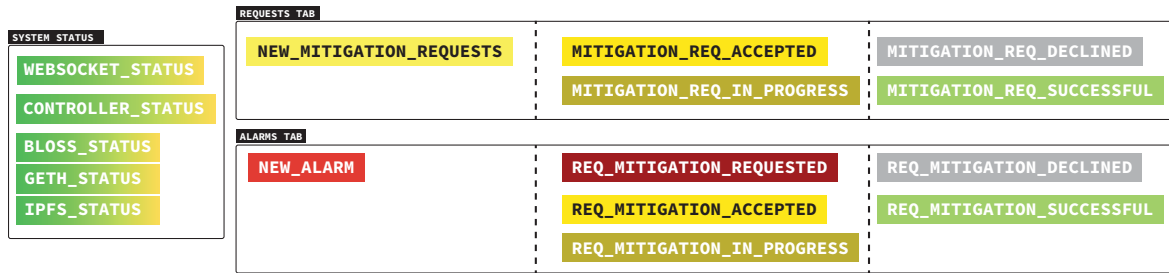


Figure 3.4: Schematic view of the user interface

The main task of the front-end SPA **bloss-dashboard** is the presentation of vital information about the status of relevant services and the status of individual **attack_reports**. Figure 3.4 depicts a schematic representation of the User Interface (UI) for **bloss-dashboard**. The following list describes the detailed content of the UI in Figure 3.4, while in Figure 3.5, the interaction with **bloss-node** is highlighted.

1. **System status:** displays the different service statuses.
 - (a) **WebSocket_Status** indicates if a WebSocket connection to **bloss-node** could successfully be initiated or is in progress.
 - (b) **Controller_Status** indicates if the AS host system is reachable. This status is redundant if the **bloss-node** instance is running on the same system as **bloss-core**. However, depending on the deployment setup, this status is important to locate hardware failures (*e.g.* if the AS host is not reachable at all, all services below are offline) and achieve loose coupling between components.
 - (c) **BloSS_Status** indicates whether the **bloss-core** service is actively running on the host AS.
 - (d) **Geth_Status** indicates whether the **geth**¹ client service is actively running on the host AS.
 - (e) **IPFS_Status** indicates whether the **ipfs**² service is actively running on the host AS.

¹<https://geth.ethereum.org/>

²<https://ipfs.io/>

2. **Requests & Alarms Tab:** both tabs should display three columns with progressing states from left to right. Thus, the left lane is dedicated to new elements. The middle lane contains all elements which are in progress and updates them accordingly. The right lane represents a log of all finished or declined elements. In more detail, the REQUESTS_TAB contains all incoming mitigation requests. Those mitigation requests are retrieved from the blockchain by **bloss-core** and relayed through **bloss-node**. The ALARMS_TAB contains all alarms triggered by the controller's own **bloss-core** instance as soon as `[THRESHOLD][BLOCKING_MBPS]`³ is breached. In figure 5.1 the aforementioned ALARMS_TAB is displayed with example `attack_reports` in all possible states.

3.2.2 BloSS Node

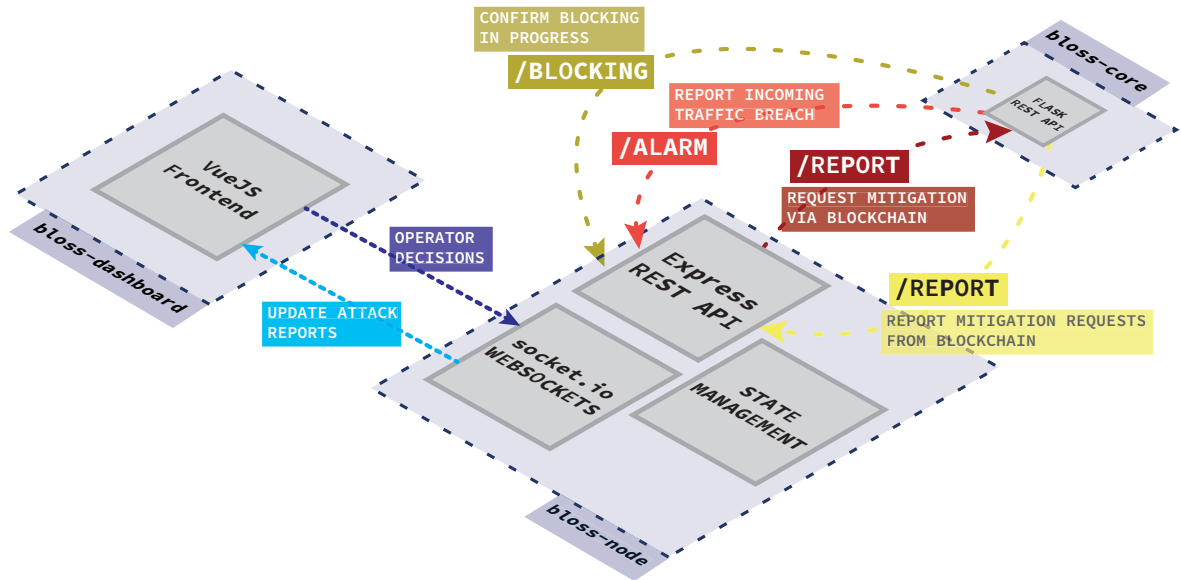


Figure 3.5: blossom-node

The **bloss-node** component is a Node.js⁴ server with multiple important tasks that facilitate the relaying of `attack_reports` received by **bloss-core**. While figure 3.5 shows the interaction with the other components focusing on **bloss-nodes** role in the interaction, the dominant tasks can be summarized with as follows:

- **State Management:** The different possible states of `attack_reports` trigger other sub-tasks depending on the current state they have.
- **Data transformation:** Data is sent around and needs to be properly processed and sometimes parsed to assure data integrity across component borders (*e.g* timestamps).

³https://github.com/blockchain-signaling-system/bloss-core/blob/master/default_config.ini#L52

⁴<https://nodejs.org/>

- **Interaction with other blossom-node:** via RESTful interfaces with other blossom-node instances. In order to directly react to `attack_reports`.
- **Interaction with blossom-core** via RESTful interfaces with host AS blossom-core instance. Blossom-node retrieves `attack_reports` from blossom-core or initiates the mitigation of mitigation requests by other ASes.
- **Interaction with blossom-dashboard** via WebSocket interface. To ensure (near) real-time monitoring, the front-end is immediately updated if any relevant state mutations were processed.
- **Managing services** on the host system (`geth`⁵, `ipfs`⁶ and `bloss-core`) by either starting, stopping or checking the status via SSH.

To summarize, the main purpose of the blossom-node server is to enable convenient communication and processing of `attack_reports` across lifecycles and component barriers.

3.2.3 BloSS Core

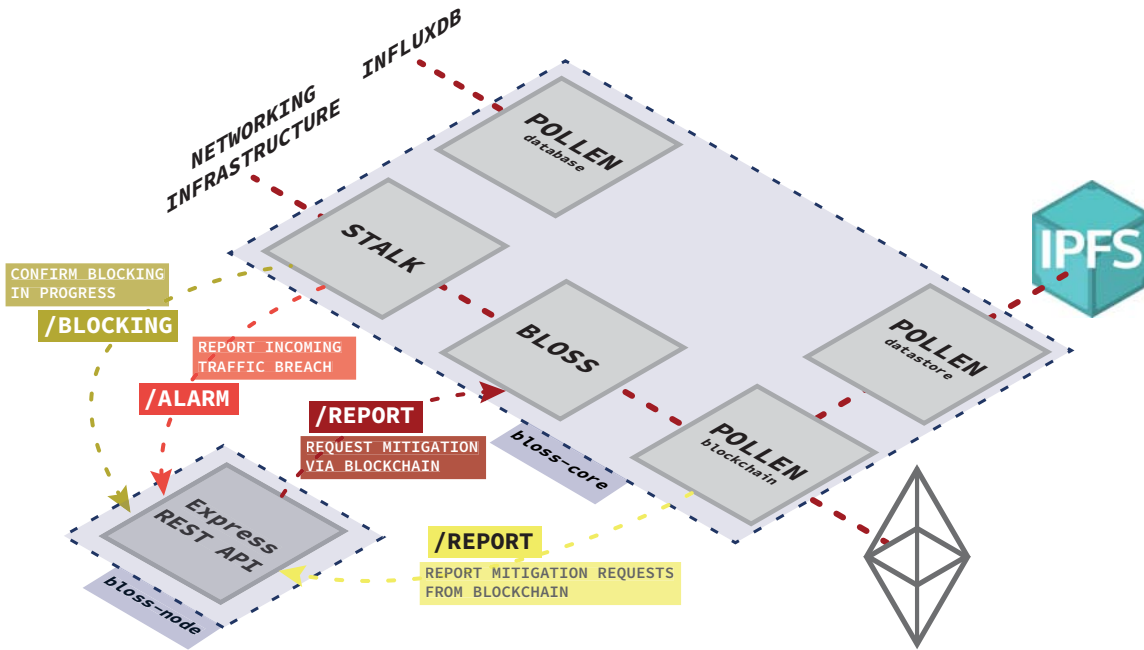


Figure 3.6: blossom-core

The blossom-core component is a modified version of the previous work by Mannhart [17] and Rodrigues et al. [25]. The modifications of blossom-core were implemented in order to enable communication with the previously described blossom-node. The main part of

⁵<https://geth.ethereum.org/>

⁶<https://ipfs.io/>

modifications enable the communication coming in and out of **bloss-core** via RESTful endpoints to **bloss-node**, as can be seen in Figure 3.6.

As the name suggests, the **bloss-core** component has essential tasks and three main modules called **STALK**, **BloSS** and **POLLEN**:

STALK monitors the network traffic of the underlying networking infrastructure

POLLEN encapsulates all communication-related tasks, handling communication with the database and the Ethereum blockchain

BloSS is the core component which receives mitigation requests and signals ongoing attacks through Pollen and stops attack traffic through Stalk

3.3 BloSS Workflows

The following sequence diagrams outline different possible sequences and conditional flows. One sequence diagram illustrates the perspective of a *target* and the other one describes the perspective of a potential *mitigator*. The start of the sequence diagram 3.7 is triggered by a DDoS attack by nodes from other subnets, operated by other ASes. In short, the AS of the attacked *target* will issue an **attack_report** to the blockchain, which in turn will be retrieved by the AS of the attacking nodes, henceforth called *mitigator*.

3.3.1 Target AS perspective

First, **bloss-core** monitors the traffic threshold and whether or not these pre-defined traffic thresholds are breached. For each breach, an **attack_report** is sent to **bloss-node** and is persisted with an initial status of **NEW_ALARM** if the same **attack_report** hash has not been persisted yet. The **attack_report** hashes are generated based on the **attack_report** objects within the **bloss-core**.

Next, the **attack_report** is relayed to the **bloss-dashboard** via WebSockets. In order to minimize load and spam to the **bloss-dashboard**, there is a simplistic Last-In-First-Out (LIFO) stack in place to only send **NEW_ALARMS** after a sustained amount of time. This simple method helps to ignore negligible and short bursts of traffic.

Whether the target decides to request mitigation or not, **bloss-node** updates the status of the respective **attack_report**. If the target decides to request help, the **attack_report** is then sent to the RESTful interface **/report** of **bloss-core**. This triggers the submission of the **attack_report** to the Ethereum blockchain. The AS of the attackers' origin will retrieve the **attack_reports** and decide whether or not to accept the requests. From here on out the process from the target's perspective is finished. The second sequence diagram 3.7 describes the process from the mitigator's perspective.

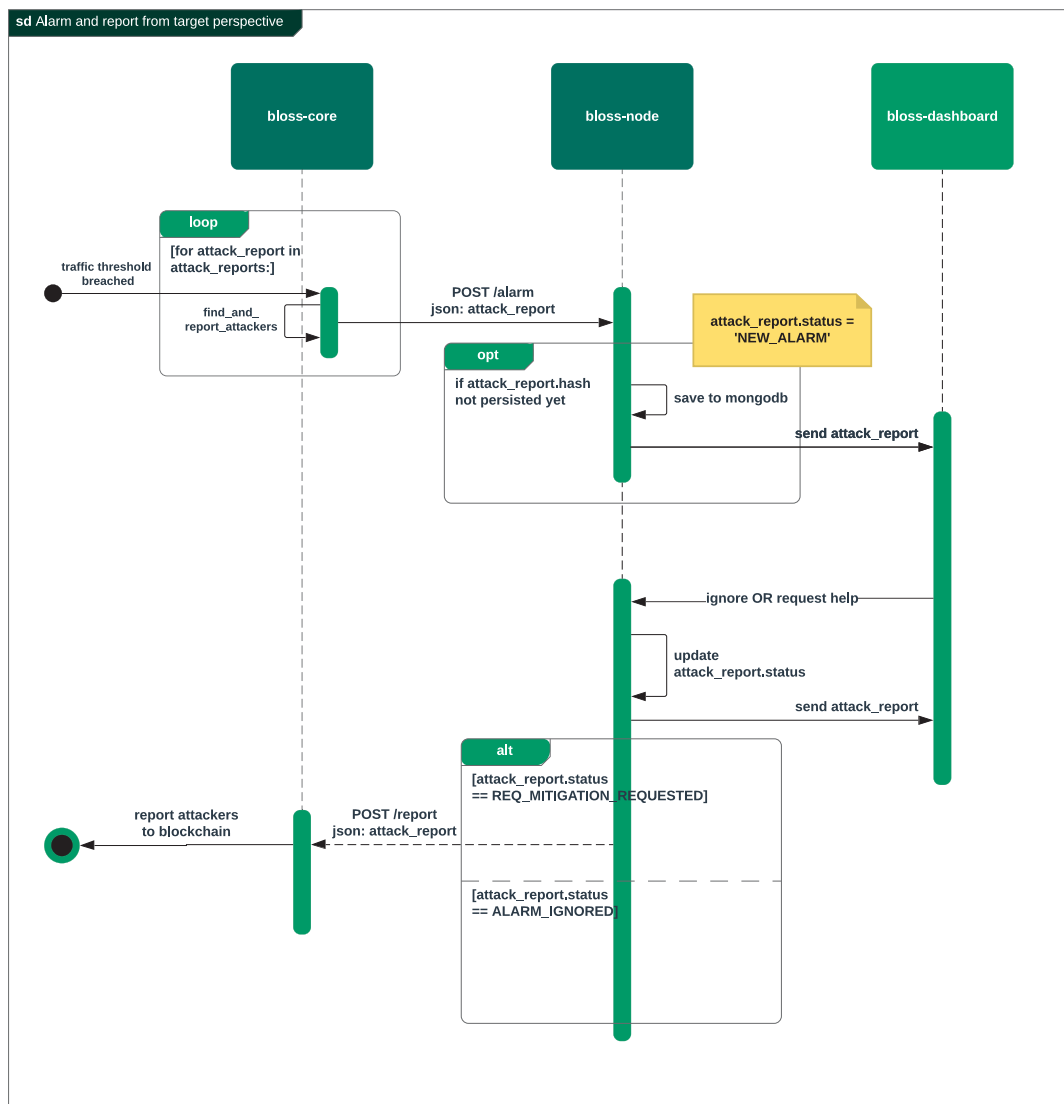


Figure 3.7: Sequence Diagram for Request Mitigation (Target perspective)

3.3.2 Mitigator AS perspective

The mitigator AS periodically retrieves attack reports from the Ethereum blockchain. As soon as an `attack_report` is retrieved, an `attack_report` is sent to `bloss-node` and is persisted with an initial status of `NEW_MITIGATION_REQ` if the same `attack_report` hash has not been persisted yet.

Next, the `attack_report` is relayed to the `bloss-dashboard` via WebSockets. Now, the analyst is displayed with all relevant details and can decide to accept or decline a mitigation request. Upon accepting the mitigation request, the response is first sent from `bloss-dashboard` to `bloss-node` and the respective `attack_report` is updated. If the mitigation request was accepted, it is immediately posted to the RESTful interface `/mitigate` of `bloss-core`.

3.4 State Management

Combining meaningful naming and appropriate coloring of states enhances the user experience for the analyst. It is crucial to standardize the use of colors in security visualization [8] to enhance rapid information processing. As already indicated in Figure 3.4, the colors are chosen according to their meaning in the process context.

The state management, therein the naming and coloring of states is another crucial part of security visualization. In this work, the naming of the states indicates the perspective of the AS. For **incoming mitigation requests**, the state identifiers start with `MITIGATION_REQ_*`, **outgoing requests for mitigation** start with `REQ_MITIGATION_*`. In the following listing, `*` stands for both, incoming and outgoing mitigation requests.

There are three overall categories for states and their identifying colors:

1. States in red hues highlight highest importance and primary priority to the analyst, hence it is used to display ● `NEW_ALARM` because they should be processed as soon as possible. Hence, if the target AS analyst decides to request help, ● `REQUEST_MITIGATION_REQUESTED` still needs to be monitored closely since the reaction of the mitigator AS has to be waited for.
2. States in yellow hues are also a primary priority, but not as urgent as states in red, since the mitigator is not target, but origin of an attack. ● `NEW_MITIGATION_REQUEST` elements are displayed in the brightest yellow hue. Further, the states ● `*_ACCEPTED` and ● `*_IN_PROGRESS` are also yellow since they need monitoring and are not finished yet. Errors could still occur.
3. States in green and grey hues are secondary priority since they do not need any user interaction and were both already inspected by the analyst. Therefore, ● `*_SUCCESSFUL` and ● `*_DECLINED` do not draw too much immediate user attention with their coloring.

The following state diagrams apply the syntax that was defined above. Both figures 3.9 and 3.10 directly represent the state machines that were implemented in this work.

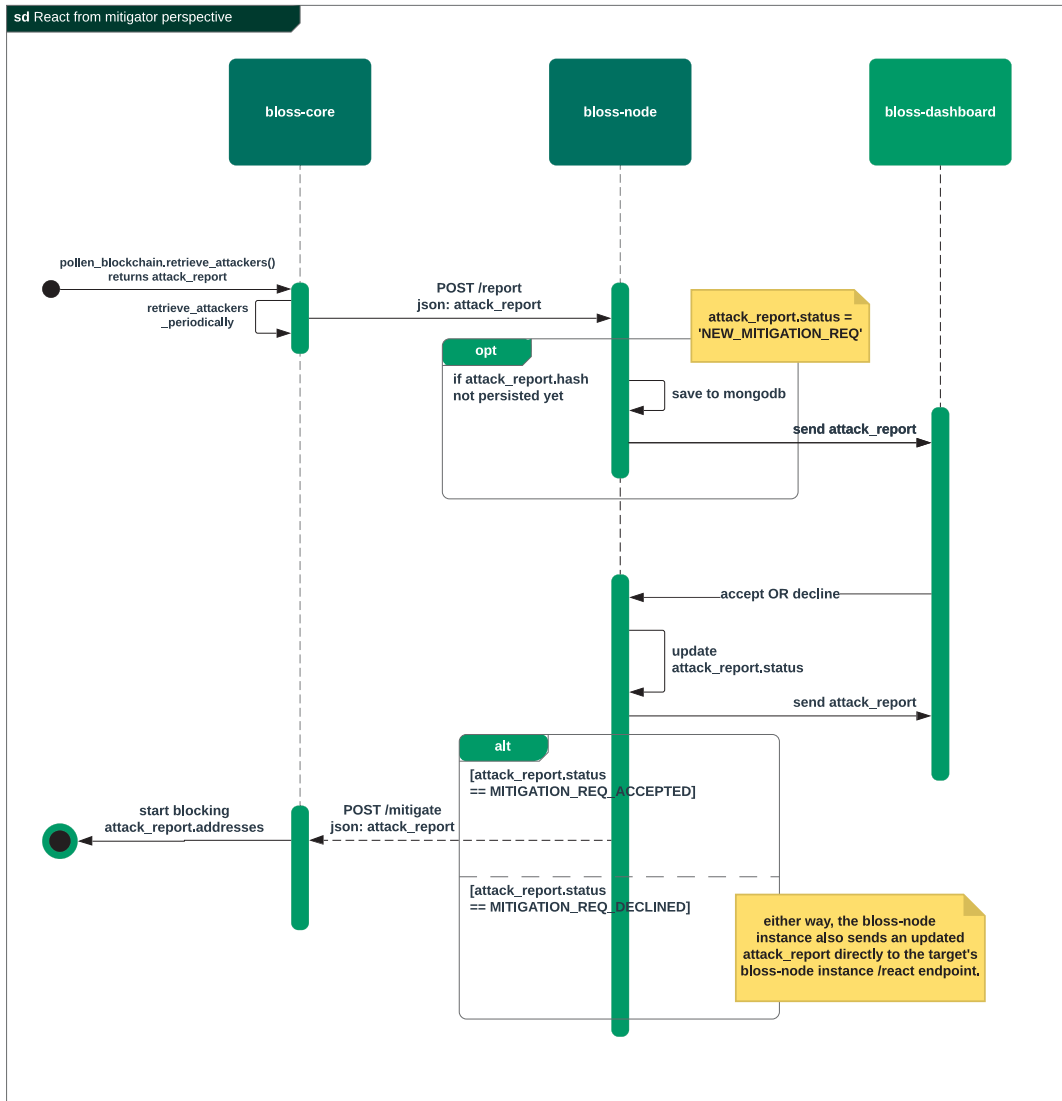


Figure 3.8: Sequence Diagram for Mitigation Requests (Mitigator perspective)

3.4.1 Mitigator Perspective

The following state diagram in figure 3.9 represents all possible states for an `attack_report` from the perspective of a mitigator AS. In other words, the attack is not targeting hosts within the AS subnet, but it is originating from it, hence the target is requesting mitigation from the AS.

The lifecycle of an `attack_report` starts with the state ● `NEW_MITIGATION_REQUEST`. The analyst of the mitigator AS has to decide whether or not the mitigation should be accepted or not. If the analyst declines, the `attack_report` is changing state to ● `MITIGATION_REQ_DECLINED` and the lifecycle of the `attack_report` ends. If the analyst

accepts the mitigation request, the state changes to an intermediary state of ● **MITIGATION_REQ_ACCEPTED**. As soon as the AS starts blocking the traffic of the reported attackers, hence mitigating the attack, the state changes to ● **MITIGATION_REQ_IN_PROGRESS**. After the blocking time expires, the mitigation successfully ends and the lifecycle of the **attack_report** changes to the state of ● **MITIGATION_REQ_SUCCESSFUL**.

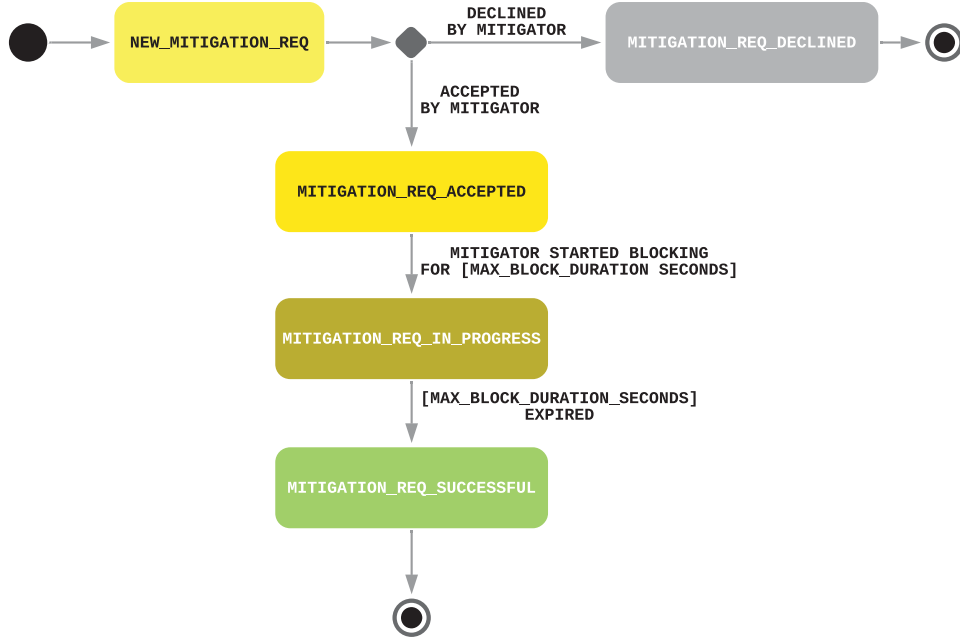


Figure 3.9: State Diagram for Mitigation Requests (Mitigator perspective)

3.4.2 Target AS Perspective

Figure 3.10 displays all possible states for an **attack_report** from the perspective of a target AS. The target AS is the victim of a DDoS attack and a new alarm is generated by the AS.

The lifecycle of an **attack_report** starts with the state ● **NEW_ALARM** as soon as a traffic breach occurs. The analyst of the target AS has to decide whether or not the attack should be mitigated or not (hence ignored). If the analyst ignores the alarm, the **attack_report** is changing state to ● **ALARM_IGNORED** and the lifecycle of the **attack_report** ends. If the operator decides that the attack is worth mitigating, he can request mitigation and the state changes to ● **REQUEST_MITIGATION_REQUESTED**. This means that the **attack_report** is posted to the blockchain and retrieved by the mitigator ASes. It is important to note that at this point, an **attack_report** with the state of ● **NEW_MITIGATION_REQUEST** is created on the mitigator's side (as soon as it was retrieved from the blockchain), hence starting the state machine in Figure 3.9. Going further in the current process from the target's perspective, the mitigator now decides whether or not to accept the request for

mitigation. If the request is declined, the state changes to ● REQ_MITIGATION_DECLINED and the lifecycle of the `attack_report` ends. If the request is accepted, the state changes to ● REQ_MITIGATION_ACCEPTED, meaning that the mitigator AS will block any harmful traffic. Now the states are congruent with the states of the mitigator in Figure 3.9. Therefore, from the moment that the mitigator starts blocking traffic, the state changes to ● REQ_MITIGATION_IN_PROGRESS. After the blocking time expires, the mitigation successfully ends and the lifecycle of the `attack_report` changes to the state of ● REQ_MITIGATION_SUCCESSFUL.

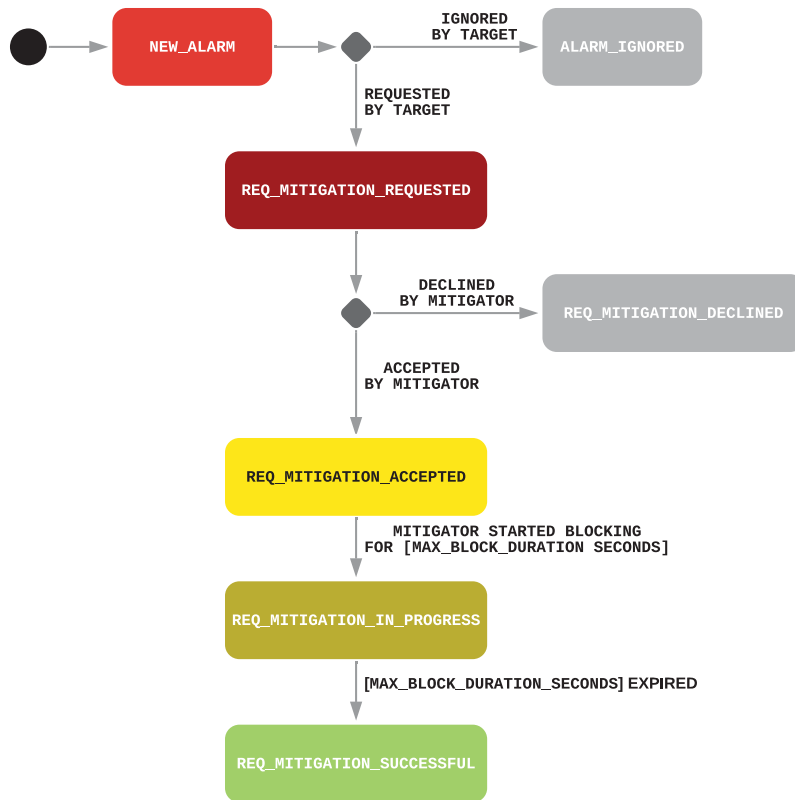


Figure 3.10: State Diagram for Request Mitigation (Target perspective)

Chapter 4

Implementation

This chapter describes the technical implementation of this work. The following sections outline the different software frameworks that were used, down to the hardware components that BloSS was tested on. Subsequently, the section titles starting with `bloss-*` correspond to the git repository names¹.

4.1 BloSS Hardware

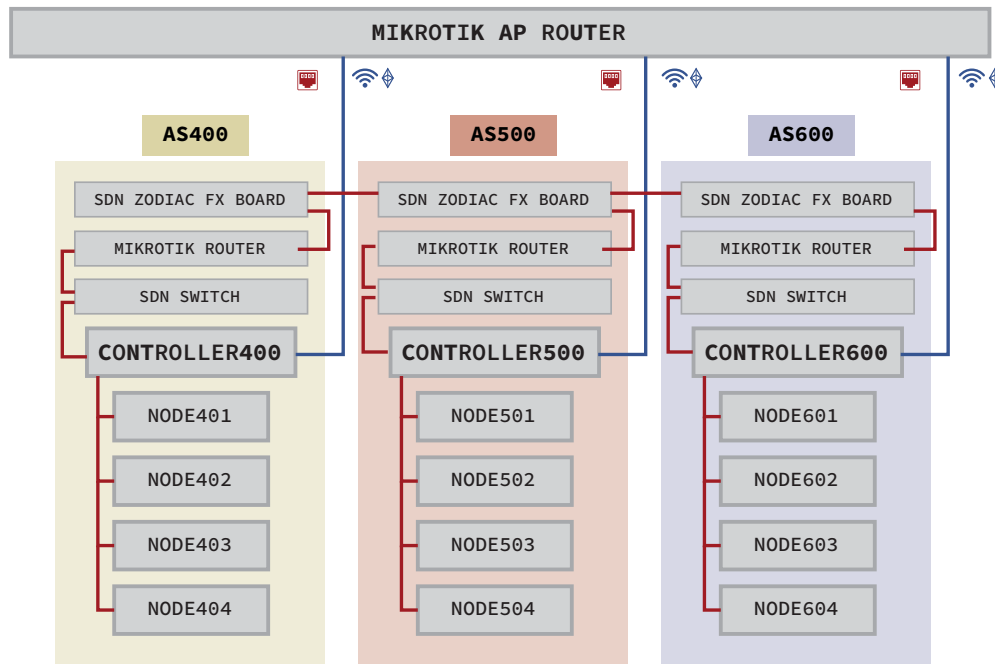


Figure 4.1: Schematic representation of hardware system architecture

¹<https://github.com/blockchain-signaling-system>

The development and evaluation of the components outlined in the previous Design chapter 3 was conducted on a dedicated hardware system. A picture of the system can be seen in Figure 4.2. Also, Figure 4.1 shows a schematic abstraction of the architecture.

A dedicated hardware system is an effective hands-on way to demonstrate the end-to-end process and concept of a collaborative defense. Therefore, it allows observing realistic networking congestion based on hardware- and software-related shortcomings instead of relying on virtualized networking hardware that might behave more predictable but less realistic.

As illustrated on 4.1, there are three identical ASes in the hardware system. The top MikroTik Router has three main tasks:

- Entry point for cluster management**, provide access through a wired or wireless network interface and has a global view of the hardware.
- Congestion-free communication channel for Ethereum blockchain** traffic, so the reporting and mitigation process is not impaired if an attack occurs.

There are two main networks set up: a primary network used to route traffic among nodes and where DDoS attacks are carried out on (Red lines on Figure 4.1). Another secondary WiFi network is dedicated to managing traffic (Blue lines on Figure 4.1). The controllers run **bloss-core**, as well as the **go-ethereum** and IPFS client. The controllers are both connected to the wired primary network, as well as to the secondary wireless network. The SDN networking is based on the Zodiac FX SDN Switches from Northbound Networks [22]. The 12 host nodes below the controllers are Raspberry Pi Model B with 1GB of RAM and a Broadcom ARMv8 CPU with 1200MHz.



Figure 4.2: Photo of hardware system to demonstrate BloSS

4.2 BloSS Management

Remotely managing a multiple nodes is most efficient by utilizing shell scripts to gradually eliminate recurring tasks. Depending on the host operating system, some scripts require the installation of additional software packages, such as `ansible`² and `pssh`³. The scripts are available in the `bloss-management` repository⁴.

- **Services:** In order to make the starting, stopping and checking of statuses easier, all relevant services have been added as `systemd` services that can be managed via `systemctl` and logs are managed by `journalctl`. To check the services statuses, a short script loops through all `CONTROLLERS` and queries the service statuses. Also, `gentle-startup.sh` and `gentle-shutdown.sh` take care of all the services to gently start and stop them to minimize the risk of corrupting any data.
- **Attack scripts:** To simulate a DDoS attack and test BloSS under realistic circumstances, the `iperf`[30] network bandwidth measurement tool was used. There are numerous scripts to automate the testing of attack scenarios which help to evaluate the various components under more realistic circumstances.
- **Configuration management:** Changing configurations from the `bloss-core` component on multiple remote hosts can be swiftly optimized by using a script to upload local config files to the distributed ASes `upload_configs.sh`.

4.3 BloSS Core

The component `bloss-core` is an extended and modified version of previous work by Mannhart [17] and Rodrigues et al. [25]. The main purpose of the extensions of `bloss-core` were implemented with the goal to enable communication with `bloss-node`. The most significant modifications are discussed in this section. The code of `bloss-core` is available on github⁵. The following subsections are named after the modules of `bloss-core` and discuss the relevant code excerpts.

4.3.1 Configuration

The configuration of `bloss-core` is managed via a simple `INI` file containing categorized parameters regarding detailed network setups and logging levels. Excerpt 1 shows the category `ENDPOINT` where `NODE` is added according to the configuration of the `bloss-node` instance. The value is configurable depending on the network setup and the `.env` configuration that `bloss-node` is executed with (Refer to 4.4.1).

²<https://www.ansible.com/>

³<https://linux.die.net/man/1/pssh>

⁴<https://github.com/blockchain-signaling-system/bloss-management>

⁵<https://github.com/blockchain-signaling-system/bloss-core>

Listing 1: Excerpt of blossom-core/default.config.ini

```

1  [ENDPOINT]
2  BLOSS = http://localhost:6000
3  STALK = http://localhost:6001
4  NODE = http://localhost:3001

```

4.3.2 BloSS

The BloSS module is the smallest module of blossom-core and consists of an API and a manager class. Listing 2 highlights the periodic retrieval of attackers by the manager class. As soon as an `attack_report` is returned from `_pollen_blockchain`, it is posted to the REST endpoint `/report` of the blossom-node server, so it can be relayed to the front-end blossom-dashboard.

Listing 2: Excerpt of blossom-core/bloss/manager.py

```

32  def _retrieve_attackers_periodically(self):
33      while True:
34          try:
35              attack_report = self._pollen_blockchain.retrieve_attackers()
36              if attack_report:
37                  requests.post(self._config['ENDPOINT'] ['NODE']
38                               + "/api/v1.0/report",
39                               json=json.dumps(
40                                   json.loads(
41                                       str(attack_report)
42                                   )
43                               )
44                  )
45              self._logger.info("Successfully retrieved {} attackers "
46                               "targeting {}".
47                               .format(len(attack_report.addresses),
48                                       attack_report.target))
49              time.sleep(self._config['INTERVAL'] ['RETRIEVE_SECONDS'])
50          except Exception as e:
51              time.sleep(self._config['INTERVAL'] ['RETRIEVE_SECONDS'])

```

Prior to this work, the BloSS REST API from listing 3 was accepting the `attack_reports` directly from the STALK module, in a completely automated fashion. The new BloSS REST API now accepts `attack_reports` sent from blossom-node. After that, the `attack_report` is posted to the blockchain by using an instance of the `PollenBlockchain` class.

Listing 3: Excerpt of blossom-core/bloss/api.py

```

19  @app.route('/api/v1.0/report', methods=['POST'])
20  def report():
21      if not request.json:

```

```

22     abort(400, message="No attack reports provided")
23     json_data = None
24     attack_reports = []
25     try:
26         json_data = request.json
27         attack_report = attack_reporting
28             .parse_attack_report_from_node(json_data)
29         attack_reports.append(attack_report)
30     except Exception as e:
31         logger.info(e)
32     try:
33         pollen_blockchain.report_attackers(attack_reports)
34     except:
35         return "Failed to report attackers to blockchain", 500
36     return "Successfully reported attackers to blockchain", 201

```

4.3.3 Pollen Module

The POLLEN module is responsible for data-exchanges. It includes specialized subclasses to access the blockchain (`PollenBlockchain`), IPFS storage (`PollenDatastore` and `PollenEncryption`) and another one for InfluxDB management (`PollenDatabase`). An additional helper class handles parsing and processing of `attack_reports` (`AttackReporting`). The excerpt in listing 4 shows the necessary method to parse `attack_reports` formatted as JavaScript Object Notation (JSON) that are sent from `bloss-node`. All other parts of the POLLEN module were not modified in this work.

Listing 4: Excerpt of `bloss-core/pollen/attack_reporting.py`

```

149     def parse_attack_report_from_node(self, message):
150         target = message['target']
151         action = message['action']
152         timestamp = message['timestamp']
153         subnetwork = str(message['subnetwork'])
154         addresses = set(message['addresses'])
155         hash = message['hash']
156         return AttackReport(target=target,
157                             action=action,
158                             timestamp=timestamp,
159                             subnetwork=subnetwork,
160                             addresses=addresses,
161                             hash=hash)

```

4.3.4 Stalk Module

The STALK module consists of two classes directly communicating with the networking hardware: The `controller` and the `simple_router`. While the `controller` analyzes

traffic flows and detects ongoing attacks, the `simple_router` route packets between different ASes through the SDN switches.

In excerpt 5, the controller of `bloss-core` sends an `attack_report` to the REST endpoint `/alarm` of `bloss-node`, as soon an ongoing attack is detected.

Listing 5: Excerpt of `bloss-core/stalk/controller.py`

```

143     def find_and_report_attackers(self, datapath_id):
144         attack_reports = self._hosts.detect_ongoing_attacks(datapath_id)
145         if len(attack_reports) > 0:
146             json_reports = []
147             count = 0
148             for report in attack_reports:
149                 self._logger.info("Sent alarm for attack report {}".format(report))
150                 json_reports.append(json.loads(str(report)))
151                 count += len(report.addresses)
152                 requests.post(self._config['ENDPOINT']['NODE'] + "/api/v1.0/alarm",
153                             json=json.dumps(json.loads(str(report))))
154                 self._database.update_reported_addresses(datapath_id, count)

```

As soon as an `attack_report` has been accepted by a mitigator, it is posted to the `/report` REST endpoint of `bloss-core` seen in listing 3. The mitigation process is started on `bloss-core` and as soon as the blocking starts, another REST call is executed to the `/blocking` endpoint of `bloss-node` to update the state of the `attack_report` (refer to excerpt 6), which in turn also updates the front-end SPA `bloss-dashboard`.

Listing 6: Excerpt of `bloss-core/stalk/controller.py`

```

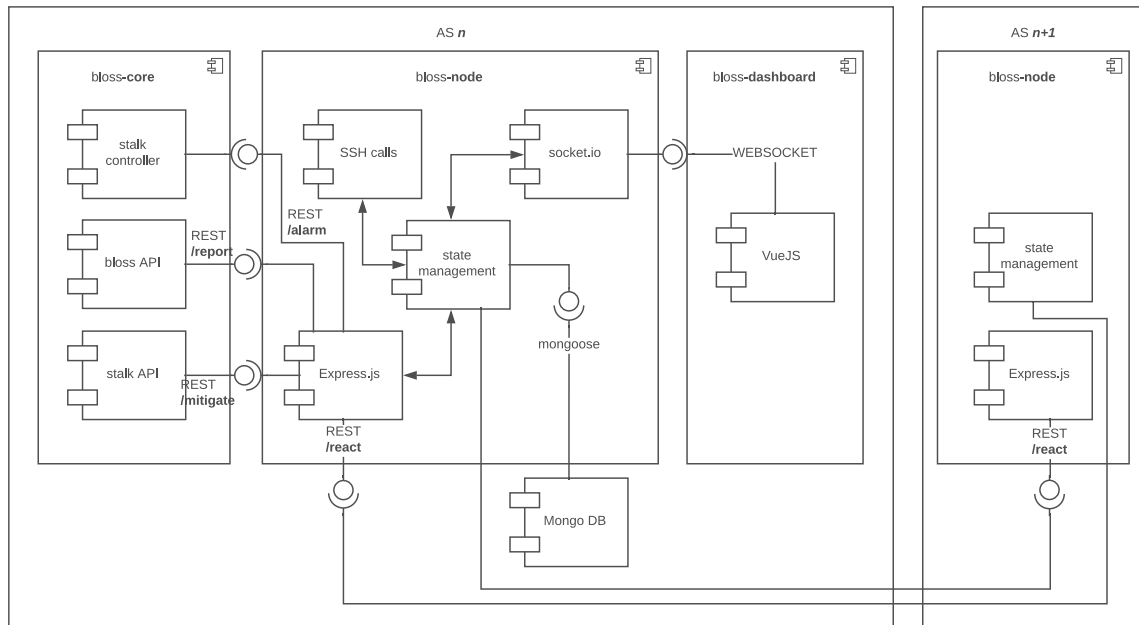
139         requests.post(
140             self._config['ENDPOINT']['NODE'] + '/api/v1.0/blocking',
141             json=json.dumps(hash_to_block))

```

4.4 Bloss Node

This section discusses the `bloss-node` component, which is implemented as a Node.js⁶ server. The asynchronous, event driven JavaScript run-time of Node is pertinent for the design of scalable network applications. In figure 4.3, the architecture of `bloss-node` is shown, highlighting the different communication interfaces, discussed in the next subsection.

⁶<https://nodejs.org/>

Figure 4.3: Component diagram of **blossom-node**

4.4.1 Configuration

To make **blossom-node** easily configurable, **dotenv**⁷ is used to manage **.env** files. An example **.env** file is shown in listing 7. Hence, the deployment to test **blossom-node** with multiple ASes and dashboards could be automated based on different **.env** files.

Listing 7: Lines 1 to 15 Excerpt of **blossom-node/.example.env**

```

1  CONTROLLER=CONTROLLER400
2  NODE_ENV=development
3  MAX_BLOCKING_DURATION_SECONDS=30
4  WS_PORT=3001
5  SSH_PORT=22
6  SSH_USER=ssh_user
7  SSH_KEY=/Users/killer/.ssh/pubkey
8  SSH_PW=password
9  STALK_PORT=:6001
10 BLOSS_PORT=:6000
11 C400_CONTROLLER_IP=172.10.15.17
12 C400_WS_PORT=4001
13 C400_SUBNET=192.168.40.0/24
14 C400_MONGOD=mongodb://localhost/c400reports
15 NODE401=192.168.40.2

```

⁷<https://www.npmjs.com/package/dotenv>

4.4.2 Communication interfaces

The `bloss-node` component provides two main communication interfaces:

- **WebSockets:** The communication with `bloss-dashboard` is managed through the WebSockets implementation of `socket.io`⁸. By using WebSockets, it is possible to update the `bloss-dashboard` front-end SPA. For instance, sending an updated `data` object can be achieved with `io.emit`. The following list outlines the usage of different WebSocket channels for different purposes.
 - **/alarmChannel:** Concerns `attack_reports` that are classified as Request Mitigation, since initially an alarm was triggered by `bloss-core` because of a traffic breach. All follow-up state changes of `attack_reports` are also pushed through this WebSocket channel (*cf.* 3.10 for all possible states).
 - **/reportChannel:** Concerns all `attack_reports` that are classified as Mitigation Request, since the report is retrieved by `bloss-core` from the blockchain. All follow-up state changes of `attack_reports` are pushed through this WebSocket channel (*cf.* 3.9 for all possible states).
 - **/statusChannel:** Concerns all statuses of services (`bloss,ipfs,geth`) that are periodically checked and then updated by sending the statuses through the `statusChannel` topic to the `bloss-dashboard`.
 - **/isControllerAvailable:** Only has one boolean value, hence the different naming. Either the controller is available or not.
- **RESTful interface:** The communication with `bloss-core` or other `bloss-node` instances is handled via RESTful interfaces, implemented with the `Express` framework⁹.
 - **/alarm** Receives an `attack_report` from `bloss-core` whenever a traffic breach occurs.
 - **/blocking** Requested as soon as an the mitigation of the blacklisted IP addresses start. This occurs when an `attack_report` was sent from `bloss-node` to `bloss-core` `/mitigate` and the mitigation process has started.

4.4.3 Data management

All `attack_reports` are persisted in a local or remote instance of MongoDB¹⁰. The MongoDB is accessed with Mongoose¹¹, which provides a straight-forward, schema-based solution to model application data. The Mongoose schema for `attack_reports` is shown in listing 8. Each Mongoose schema maps to a MongoDB collection and defines the shape of the documents within that collection. All `attack_reports` start in the same

⁸<https://socket.io/>

⁹<https://expressjs.com/>

¹⁰<https://mongodb.com/>

¹¹<https://mongoosejs.com/>

state ● `NEW_ALARM`. Therefore, the default values for new timestamps are set as default to `Date.now`, except the `timestamp` property, which is the timestamp parsed from the incoming JSON of `bloss-core`. Having multiple timestamp attributes allows the analysis of timing differences and other issues. For instance, if `timestamp_accepted` is equal to `created` the `/textttattack_report` has not been accepted yet.

- **hash** refers to the hash of the `attack_report`.
- **target** refers to the attack target and contains the IP address of the target.
- **timestamp** refers to the timestamp parsed from the incoming `attack_report`.
- **timestamp_requested** this timestamp is updated with the current time as soon as a mitigation was requested and the `attack_report` state changes to ● `★_REQUESTED`.
- **timestamp_accepted** this timestamp is updated with the current time as soon as the `attack_report` state changes to ● `★_ACCEPTED`.
- **timestamp_declined** timestamp is updated with the current time as soon as the `attack_report` state changes to ● `★_DECLINED`
- **timestamp_in_progress** refers to the timestamp which is updated with the current time as soon as the `attack_report` state changes to ● `★_IN_PROGRESS`
- **timestamp_successful** timestamp that is updated based on the current time as soon as the `attack_report` state changes to ● `★_SUCCESSFUL`
- **created** this timestamp is never changed
- **action** refers to the action the mitigation should perform (*e.g.*, blackhole traffic).
- **subnetwork** contains a subnetwork identified (*e.g.*, 192.168.60.0/24).
- **addresses** contains an array of strings of attacker IP addresses.
- **status** contains a valid status as a string.

Listing 8: Lines 207 to 221 Excerpt of `bloss-node/server.js`

```

207 const reportSchema = new mongoose.Schema({
208   hash: String,
209   target: String,
210   timestamp: { type: Date, default: Date.now },
211   timestamp_requested: { type: Date, default: Date.now },
212   timestamp_accepted: { type: Date, default: Date.now },
213   timestamp_declined: { type: Date, default: Date.now },
214   timestamp_in_progress: { type: Date, default: Date.now },
215   timestamp_successful: { type: Date, default: Date.now },
216   created: { type: Date, default: Date.now },
217   action: String,
218   subnetwork: String,
219   addresses: [String],
220   status: { type: String, default: RequestMitigation.NEW_ALARM },
221 });
222 const Report = mongoose.model('Report', reportSchema);

```

4.4.4 State management

The rationales regarding the state management of this work was discussed in prior section 3.4. The following listing 9 shows how the state is changed with the use of Promises¹². It outlines what happens when a `NEW_ALARM` is created. The same, or similar operations happen whenever a state change is triggered by the front-end `bloss-dashboard`, since the front-end also needs to persist the state changes via mongoose in an asynchronous fashion. Hence, the `bloss-node` serves as a relay server, state manager and also as a single source of truth (SSOT) for the `bloss-dashboard`.

Listing 9: Excerpt of `bloss-node/server.js`

```

1  app.post('/api/v1.0/alarm', (req, res) => {
2    try {
3      // Step 1: declare promise to check if duplicate exists
4      var checkDuplicatePromise = () => {
5        return new Promise((resolve, reject) => {
6          Report.find({ hash: attack_report.hash }, function (err, data) {
7            err
8              ? reject(err)
9              : resolve(data);
10           });
11         });
12       };
13
14      // Step 1.1: Declare Promise to persist attack report
15      var persistAttackReportPromise = () => {
16        return new Promise((resolve, reject) => {
17          var ts_from_moment = moment(moment()
18            .format('YYYY:MM:DD-HH:mm:ss'), 'YYYY:MM:DD-HH:mm:ss').toDate();
19
20          const report = new Report({
21            hash: attack_report.hash,
22            target: attack_report.target,
23            timestamp: ts_from_moment,
24            action: attack_report.action,
25            subnetwork: attack_report.subnetwork,
26            addresses: attack_report.addresses,
27            status: RequestMitigation.NEW_ALARM
28          });
29
30          report.save(function (err, data) {
31            if (err) {
32              console.error("Error while saving", err);
33            } else {
34              // Instead of sending via io.emit directly, add to stack
35              if (attack_report.subnetwork === process.env.C400_SUBNET) {
36                alarmStackC400.push(data);
37              }
38              if (attack_report.subnetwork === process.env.C500_SUBNET) {
39                alarmStackC500.push(data);
40              }
41              if (attack_report.subnetwork === process.env.C600_SUBNET) {

```

¹²<https://www.promisejs.org/>

```

42         alarmStackC600.push(data);
43     }
44     res.json({ message: 'Report persisted', data: data });
45 }
46 });
47 });
48 };
49
50 // Step 2: Declare async promise handler
51 var callCheckDuplicatePromise = async () => {
52     var result = await (checkDuplicatePromise());
53     // anything here is executed after result is resolved
54     if (result.length > 0) {
55         // There is already a report with this hash...
56         console.info('Duplicate hash found');
57     } else {
58         var persist = await (persistAttackReportPromise());
59     }
60 };
61 // Step 3: Execute the call
62 callCheckDuplicatePromise().then(function (result) {
63     res.json({ message: 'Alarm already persisted' });
64 });
65 } catch (e) {
66     next(e)
67 }
68 });

```

4.4.5 Service Management

Listing 10 shows the code that makes SSH calls to the AS running `bloss-core` and the other services (`geth`, `ipfs`). Since the front-end SPA `bloss-dashboard` could not directly make these SSH calls from a browser environment, `bloss-node` takes care of it.

Listing 10: Lines 172 to 194 Excerpt of `bloss-node/server.js`

```

172 /**
173  * Controls services on the controllers via SSH
174  */
175 socket.on('serviceControlRequest', function (data) {
176     console.info("serviceControlRequest received");
177     if (global.controllerAvailability) {
178         switch (data.cmd) {
179             case 'start':
180                 console.info(SSH_prefix + "Starting " + data.service);
181                 execSSH("sudo systemctl start " + data.service, data.service);
182                 break;
183             case 'stop':
184                 console.info(SSH_prefix + "Stopping " + data.service);
185                 execSSH("sudo systemctl stop " + data.service, data.service);
186                 break;

```

```

187         default:
188             console.error(SSH_prefix + "Something went wrong" + data);
189         }
190     } else {
191         console.error(SSH_prefix + "Controller not available, can't execSSH");
192     }
193 });
194 });

```

4.5 BloSS Dashboard

The following section describes the various implementation aspects of the `bloss-dashboard` component. As a front-end framework, the JavaScript framework Vue.js¹³ is used. As a starting point, a boilerplate of Vue CLI 3¹⁴ is generated.

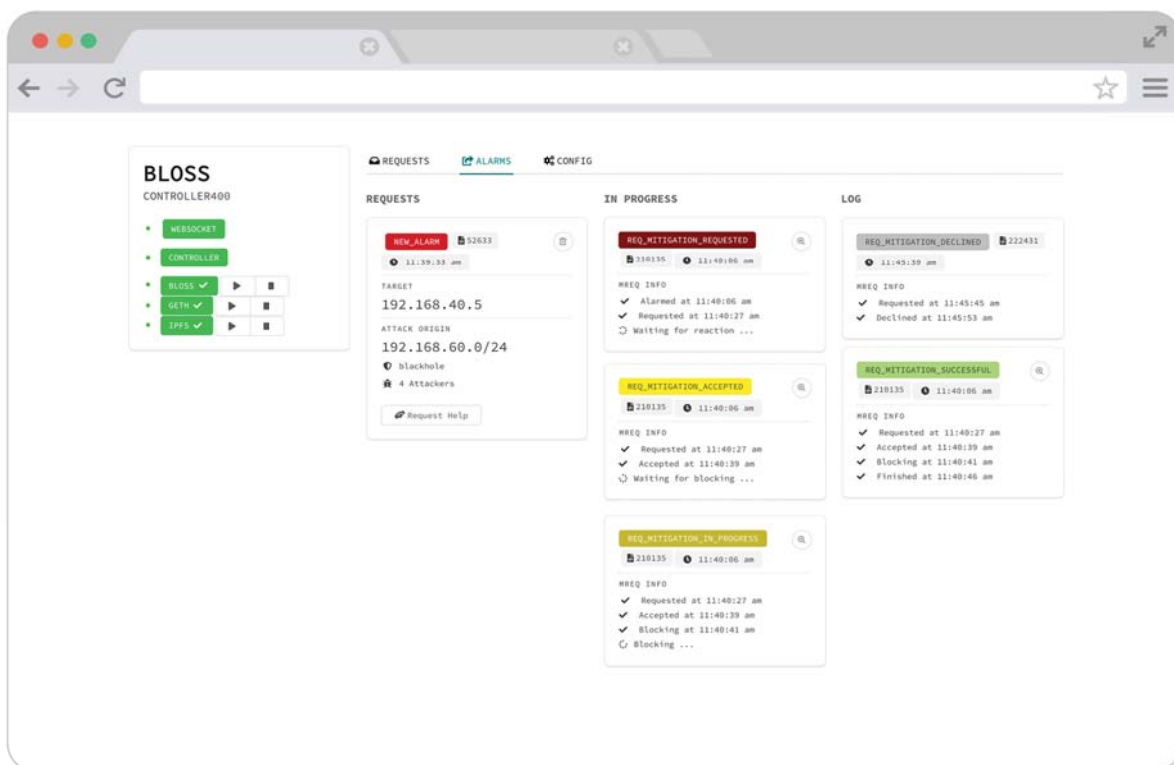


Figure 4.4: `bloss-dashboard` with all possible states of an attack_report

¹³<https://vuejs.org>

¹⁴<https://cli.vuejs.org/>

4.5.1 Front-end stack

Buefy¹⁵ is a UI library for Vue.js based on Bulma¹⁶, which is a lightweight, open source CSS framework based on Flexbox. Buefy was primarily used to set up the layout column grid and some dynamic components were integrated. The tabs pane seen in Figure 4.4 at the top is from the `element-ui` package, accompanied by fontawesome icons from the `fontawesome`¹⁷ package.

Within the individual tabs, the main layout grid is created with Buefy's `columns` and `column` classes to achieve flexibility and a responsive design out-of-the-box. The column design follows the same principles outlined in 3.2.1 and is shown in Figure 3.4. Within each column, a Vue.js for-loop iterates through a `computed` array shown in Listing 11. The computed properties are filtered according to different states. Hence, within the Requests tab, each column loops through computed (filtered) arrays containing instances of `MitigationRequest.vue`. In the Alarms tab, each column loops through computed (filtered) array containing instances of `AttackAlarm.vue`.

Listing 11: Excerpt of `bloss-dashboard/src/views/Home.vue`

```

737 },
738 computed: {
739   newMitigationRequests: function() {
740     return this.mitigationRequests.filter(function(mitigationRequest) {
741       return mitigationRequest.status == constants.NEW_MITIGATION_REQ;
742     });
743   },
744   acceptedAndProgressMitigationRequests: function() {
745     return this.mitigationRequests.filter(function(mitigationRequest) {
746       return (
747         mitigationRequest.status == constants.MITIGATION_REQ_ACCEPTED ||
748         mitigationRequest.status == constants.MITIGATION_REQ_IN_PROGRESS
749       );
750     });
751   },
752   successfulAndDeclinedMitigationRequests: function() {
753     return this.mitigationRequests.filter(function(mitigationRequest) {
754       return (
755         mitigationRequest.status == constants.MITIGATION_REQ_SUCCESSFUL ||
756         mitigationRequest.status == constants.MITIGATION_REQ_DECLINED
757       );
758     });

```

4.5.2 Components

Vue.js components are reusable Vue instances. Since all `attack_reports` should have the same structure, two custom components adhering to this structure, were implemented.

¹⁵<https://buefy.github.io>

¹⁶<https://bulma.io>

¹⁷<https://www.npmjs.com/package/@fortawesome/vue-fontawesome>



Figure 4.5: abstraction of `MitigationRequest.vue` and `AttackAlarm.vue`

The `MitigationRequest.vue` models all states of incoming reports and are displayed in the Requests tab. Figure 4.5 shows an abstraction of the `MitigationRequest.vue` and the `AttackAlarm.vue` component UI, including the fontawesome¹⁸ icons that were selected for their purposes. Depending on the current state of the individual component instance, the layout is changed accordingly.

Listing 12 shows an extract of `MitigationRequest.vue`. With the help of conditional rendering, one component can model every possible state correctly without code duplication. The `v-show` directive allows the straightforward implementation of conditional rendering. For instance, line 51 in listing 12 uses `v-show` to dynamically render the `div`, which is only visible if the current `status` of the mitigation request instance equals `'MITIGATION_REQ_ACCEPTED'`. Using the same approach, the UI can be extended dynamically.

Listing 12: Excerpt of `bloss-dashboard/src/components/MitigationRequest.vue`

```

50 <!-- BEGIN this.status === MITIGATION_REQ_ACCEPTED -->
51 <div v-show="(this.status==='MITIGATION_REQ_ACCEPTED')">
52   <p class="heading">MREQ INFO</p>
53   <span>
54     <font-awesome-icon style="margin-right:.75em" icon="check"/>
55     Accepted at {{moment(this.timestamp_accepted).format('hh:mm:ss a')}}
56     <br>
57     <font-awesome-icon
58       style="margin-right:.75em"
59       v-show="(this.status==='MITIGATION_REQ_ACCEPTED') "
60       icon="spinner"
61       pulse
62     />Waiting for blocking
63   </span>
64 </div>
65 <!-- END this.status === MITIGATION_REQ_ACCEPTED -->

```

¹⁸<https://www.npmjs.com/package/@fortawesome/vue-fontawesome>

The main advantage of these reusable Vue components is the combined usage with dynamic rendering (*e.g.*, by using a `v-for` loop). The code excerpt in listing 13 shows a dynamic `v-for` loop that iterates over all elements in the computed array `newMitigationRequests` (compare listing 11). Props are custom attributes registered on a component. By using the `v-bind` directive, values are passed to the `props` of the component and become properties of the component instance.

Listing 13: Excerpt of `bloss-dashboard/src/views/Home.vue`

```

521     <div class="columns">
522       <div class="column">
523         <p style="margin-top:0.5em" class="col-title">REQUESTS</p>
524         <mitigation-request
525           @showRequestDetailsEvent="showDetails(mit._id)"
526           @declineMREQEvent="declineMREQ(mit._id)"
527           @acceptMREQEvent="acceptMREQ(mit._id)"
528           v-for="mit in newMitigationRequests"
529           v-bind:key="mit.key"
530           v-bind:hash="mit.hash"
531           v-bind:target="mit.target"
532           v-bind:timestamp="mit.timestamp"
533           v-bind:timestamp_requested="mit.timestamp_requested"
534           v-bind:timestamp_accepted="mit.timestamp_accepted"
535           v-bind:timestamp_declined="mit.timestamp_declined"
536           v-bind:timestamp_in_progress="mit.timestamp_in_progress"
537           v-bind:timestamp_successful="mit.timestamp_successful"
538           v-bind:action="mit.action"
539           v-bind:subnetwork="mit.subnetwork"
540           v-bind:addresses="mit.addresses"
541           v-bind:status="mit.status"
542         ></mitigation-request>
543         <div v-if="newMitigationRequests.length === 0">
544           <el-alert title=" No new MREQ" type="info" :closable="false">
545             </el-alert>
546         </div>
547       </div>

```

4.5.3 Detail Modal Dialog

Details are only provided on demand, as the Visual Information Seeking Mantra suggests [26]: Overview first, zoom and filter, then details-on-demand. Details are only provided on demand [12]. As figure 4.6 shows, details are displayed in a Modal Dialog that can be easily closed by clicking outside of the dialog.

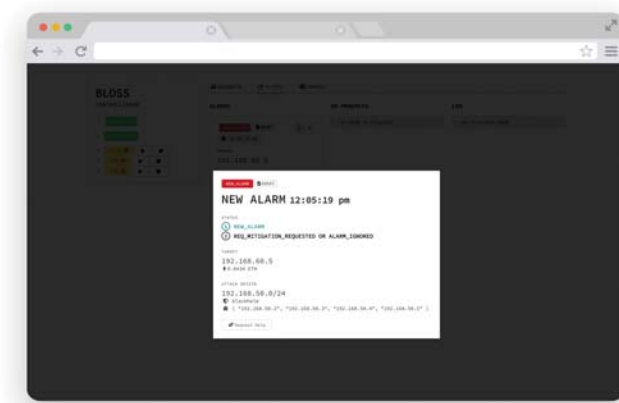


Figure 4.6: Screenshot of detail modal dialog

4.5.4 Responsive Design

As shown in figure 4.7 and 4.8, the `bloss-dashboard` is fully responsive down to tablets and smartphone screens.

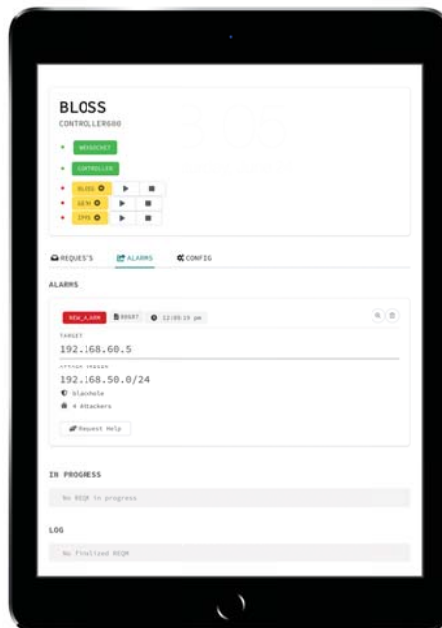


Figure 4.7: Screenshot of `bloss-dashboard` overview

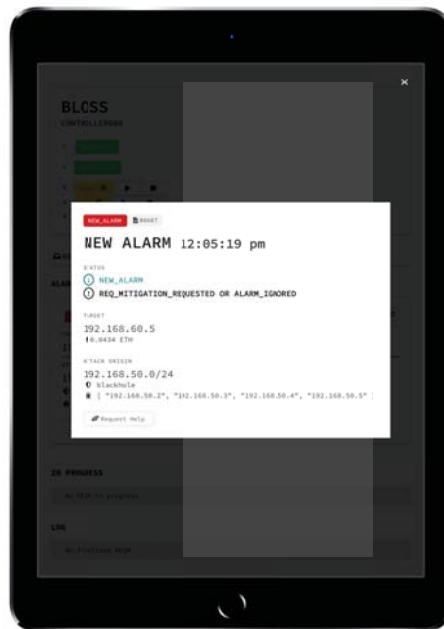


Figure 4.8: Screenshot of blossom-dashboard detail modal dialog

Chapter 5

Evaluation

The goal of this chapter is to evaluate the implemented security management tool for BloSS and identify whether the requirements of section 3.1 are satisfied. The functional requirements **F_R1** and **F_R2** are covered by the use cases described in section 5.1. The non-functional requirements of *Usability* and *Performance* are discussed in section 5.2

5.1 Functional Requirements

In the following Use Case subsections, the following scenario is assumed: there are three ASes running the architecture proposed in Figure 3.3. Multiple nodes from the AS **controller600** subnet (192.168.60.0/24) simulate a volumetric network traffic DDoS attack on a single node (192.168.40.5) from the subnet 192.168.40.0/24.

As depicted in figure 3.2, the main use cases are the following:

- **UC1: Request Mitigation or Ignore Alarm** - A new alarm is displayed and evaluated by the operator. Either the alarm is classified as a valid threat and if mitigation should be requested, or if the alarm can be ignored.
- **UC2: Accept or Decline Mitigation Request** - A new Mitigation Request is displayed and evaluated by the operator. Either the request is accepted or declined.
- **UC3: Start and stop services** - **bloss**, **geth** and **ipfs** can be started or stopped.

The use cases directly relate to the main functional requirements, *i.e.* **UC1** relates to **F_R1**, **UC2** relates to **F_R2** and **UC3** relates to **F_R3**.

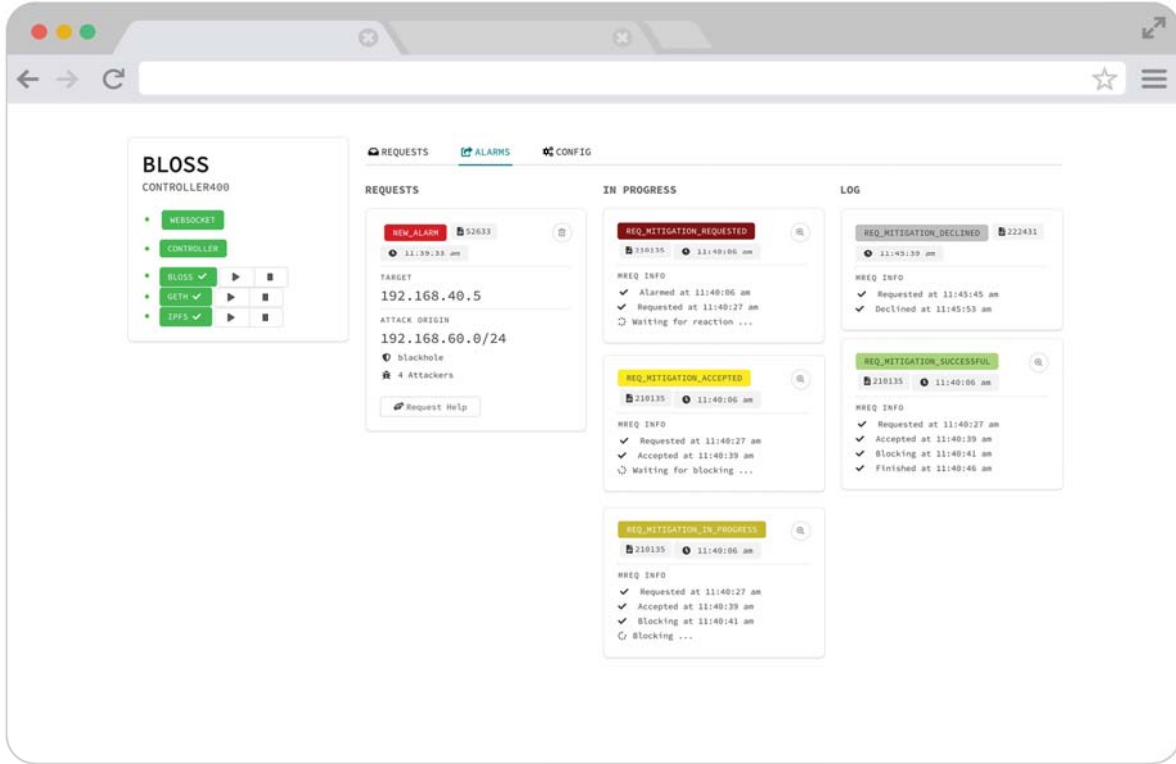


Figure 5.1: blossom-dashboard UI running on controller400 with all possible states of an example attack_report in the ALARMS_TAB

5.1.1 UC1: Request Mitigation or Ignore Alarm (Target's perspective)

Preconditions: User Interface loaded, all services active.

Actors: operator of controller400, operator of controller600, node404 [192.168.40.5], four attackers from subnet [192.168.60.0/24].

Steps:

- **Step 1:** blossom-core triggers a traffic alarm as soon as [THRESHOLD] [BLOCKING_MBPS]¹ is breached.
- **Step 2:** Display toast with message ● NEW_ALARM (Figure B.3).
- **Step 3:** Switch tab, display attack_report with status ● NEW_ALARM is displayed (Figure B.4).
- **Step 4: A:** Target operator ignores attack_report and the status of the attack_report changes to ● ALARM_IGNORED (Figure B.5).
- **Step 4: B:** Target network operator requests help for attack_report which triggers a POST call to blossom-core which then posts it to blockchain. The state changes to ● REQUEST_MITIGATION_REQUESTED (Figure B.6).

¹https://github.com/blockchain-signaling-system/bloss-core/blob/master/default_config.ini#L52

- **Step 5: A** Mitigator network operator declines the request for mitigation and the status of the attack_report changes to ● REQ_MITIGATION_DECLINED (Figure B.7).
- **Step 5: B** Mitigator network operator accepts the request for mitigation and the status of the attack_report changes to ● REQ_MITIGATION_ACCEPTED (Figure B.8).
- **Step 6:** As soon as the mitigator starts blocking, the attack_report is also ● REQ_MITIGATION_IN_PROGRESS (Figure B.9).
- **Step 7:** After the expiration of the maximum block duration, the attack_report is completed and thus ends in the status ● REQ_MITIGATION_SUCCESSFUL.

5.1.2 UC2: Accept or Decline Mitigation Request (Mitigator's perspective)

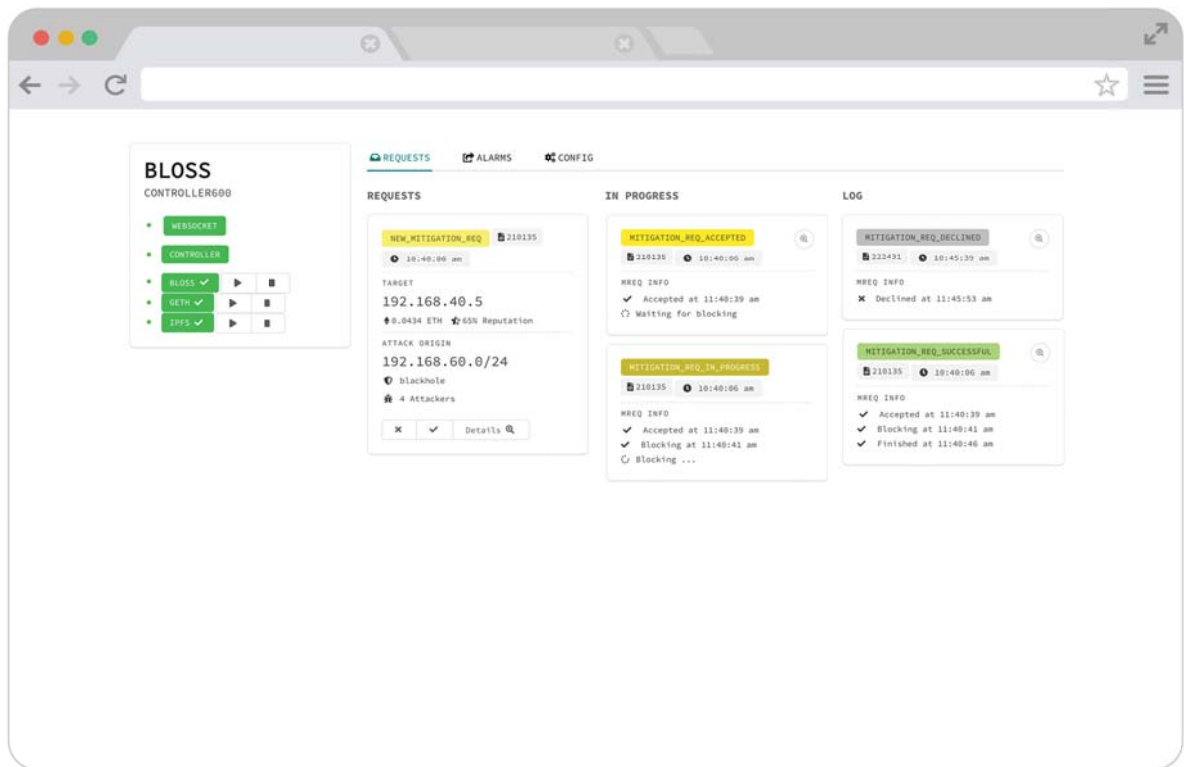


Figure 5.2: blossom-dashboard UI running on controller600 with all possible states of an example attack_report in the REQUESTS_TAB

Preconditions: User Interface loaded, all services active.

Actors: operator of controller400, operator of controller600, node404 [192.168.40.5], four attackers from subnet [192.168.60.0/24].

Steps:

- **Step 1:** blossom-core² periodically retrieves attack_reports from the blockchain and

²<https://github.com/blockchain-signaling-system/bloss-core/blob/master/bloss/manager.py#L32>

triggers a new POST call to **bloss-node**, which relays it to **bloss-dashboard** and displays it to the operator with status ● NEW_MITIGATION_REQUEST (Figure B.11).

- **Step 2A:** The operator decides to accept the request and the attack_report's status changes to ● MITIGATION_REQ_DECLINED (Figure B.12).
- **Step 2B:** The mitigator network operator decides to accept the request and the attack_report's status changes to ● MITIGATION_REQ_ACCEPTED (Figure B.13).
- **Step 3:** As soon as **bloss-core** starts to mitigate traffic from attacking hosts, the attack_report's status changes to ● MITIGATION_REQ_IN_PROGRESS (Figure B.14).
- **Step 3** After the expiration of the maximum block duration, the attack_report is completed and thus ends in the status ● MITIGATION_REQ_SUCCESSFUL (Figure B.15).

5.1.3 UC3: Start or stop services

This use case is a simple one-click interaction and not requiring a complete use case description. As soon as the WebSocket connection to **bloss-node** is successful and the controller is available, the status indicators show the service statuses in the left status box (Refer to 5.2). The start and stop icon can be clicked to initiate a change in the service status. Therefore, the third use case is also covered.

5.2 Non-functional requirements

The non-functional requirement **NF_R1** concerns the *Usability* of the dashboard. To evaluate the Usability of the UI designed in this work, the GNOME developer design principles [29] and the Material Design Guidelines [9] are considered. Additionally, the Visual Information Seeking Mantra [26]: Overview first, zoom and filter, then details-on-demand is taken into consideration.

- **Keep user interface complexity to a minimum [29]:** The complexity of the UI is minimal and based on a known layout from software development: the Kanban board [23]. The UI has the same logic as the Kanban board. This leads to a low entry barrier to understanding the UI.
- **Require as little work from the user as possible [29]:** Every task that can be performed on the UI (*e.g.*, Accepting a Mitigation Request and checking the detail modal dialog) does not need more than two or a maximum of three clicks.
- **Predictable[9]:** The UI is *predictable* as the main layout does not fluctuate and the empty states (*e.g.*, No new MREQ) indicate where in the layout new elements will appear.
- **Consistent[9]:** The UI is *consistent*, *e.g.*, the Kanban layout is used in both tabs and the basic scheme for boxes is generalized (*cf* Figure 4.5).

- **Responsive[9]:** The UI is fully *responsive* and able to scale up or down according to the dimensions of the users preferred screen size. This enables flexible usage in different scenarios.

Chapter 6

Conclusion

The global rise of DDoS attacks and cybersecurity threats emphasize the importance of reliable defense systems. As DDoS become more frequent, it is imperative to take countermeasures and deploy a defense system. One of the approaches to defend against DDoS attacks is a cooperative defense system, where an alliance of entities collaborates. The entities can control Internet traffic and aim to create a globally distributed defense against DDoS attacks, whereas the defense capacity of a single entity extends with the alliance. The BloSS is a collaborative, multi-domain and decentralized defense system that approaches defense in a collaborative way, where each AS is taking part in the defense alliance. The exchange of attack information about ongoing attacks is exchanged via the Ethereum blockchain. The primary challenge of decentralized approaches is the difficulty in co-operating and communicating among ASes. Human decision-makers are indispensable to make crucial decisions regarding cybersecurity threats. A human decision-maker (*e.g.*, an analyst) needs to decide whether a threat is severe and define a course of action. The analyst requires an appropriate overview of all the attack-relevant data in an appropriate security management tool.

In conclusion, the cyber security analyst is the primary stakeholder for an interactive Security Management dashboard for BloSS. The main functional requirements illustrate the use cases that an analyst can request mitigation (or ignore the alarm) and accept (or decline) a mitigation request from other ASes. A predictable, consistent, responsive UI and (near) real-time performance detail the primary non-functional requirements.

This work focused on the design, implementation and evaluation of an interactive security management and visualization tool for BloSS. The designed architecture of this work included three components: **bloss-core**, **bloss-node** and **bloss-dashboard**. The component **bloss-core** interacts with the underlying networking infrastructure and was extended to enable communication with the **bloss-node** component. The **bloss-node** component manages the individual states of threats, takes care of recurring tasks and offers communication interfaces. The **bloss-dashboard** is a predictable, consistent, easy-to-understand interactive UI in the context of the collaborative defense system BloSS.

The implemented architecture of this work offers a UI that allows a user to react on an individual DDoS threat (`attack_report`) level and monitor the threats, but also manage individual system services.

6.1 Future Work

The following list outlines a selection of different aspects that allow room for improvement and future work.

- **State management:** The implementation of the state management on `bloss-node` offers room for improvement. Some parts of the state management could also be handled via Smart Contracts and implemented in Solidity and would need further extensions and modifications of `bloss-core`. (*e.g.*, currently the declining of attack-reports is handled off-chain and directly via REST calls. This could be implemented in the Smart Contract on-chain)
- **Reputation and Payment:** As outlined in prior work [11], a reputation system with rating and payment functionality would further enhance the evaluation of BloSS.
- **Hardware system:** Depending on the further research goals, either the current hardware system is extended virtually or physically to allow for more complex use cases to be evaluated.
- **Smart Contract Deployment:** The deployment of Smart Contracts is partially manual and the initial setup of BloSS should be further automated.
- **Configurability:** As indicated in the current UI, the Config tab would offer an optimal space to place more configuration possibilities (*e.g.*, change `config.ini` params of `bloss-core` directly from the UI).

Bibliography

- [1] Alexander Khalimonenko, Oleg Kupreev, E. B. (2018). Ddos attacks in q1 2018. <https://securelist.com/ddos-report-in-q1-2018/85373/>. Online, accessed 2018-09-11.
- [2] Alshaikh, Z., Alarifi, A., and Alsaleh, M. (2013). Christopher alexander’s fifteen properties: Toward developing evaluation metrics for security visualizations. In *2013 IEEE International Conference on Intelligence and Security Informatics*, pages 295–300.
- [3] Barford, P., Dacier, M., Dietterich, T., Fredrikson, M., Giffin, J., Jajodia, S., Jha, S., Li, J., Liu, P., Ning, P., Ou, X., Song, D., Strater, L., Swarup, V., Tadda, G., Wang, C., and Yen, J. (2010). Cyber sa: Situational awareness for cyber defense.
- [4] Endsley, M. (1995). Endsley, m.r.: Toward a theory of situation awareness in dynamic systems. *human factors journal* 37(1), 32-64. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37:32–64.
- [5] Few, S. (2013). *Information Dashboard Design: Displaying Data for At-a-glance Monitoring*. Analytics Press.
- [6] Franklin, L., Pirrung, M. A., Blaha, L. M., Dowling, M. V., and Feng, M. (2017). Toward a visualization-supported workflow for cyber alert management using threat models and human-centered design.
- [7] Garae, J. and Ko, R. K. L. (2017). *Visualization and Data Provenance Trends in Decision Support for Cybersecurity*, pages 243–270. Springer International Publishing, Cham.
- [8] Garae, J., Ko, R. K. L., and Apperley, M. (2018). A full-scale security visualization effectiveness measurement and presentation approach. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/Big-DataSE)*, pages 639–650.
- [9] Google Material Design (2018). Material design - understanding layout. <https://material.io/design/layout/understanding-layout.html#>. Online, accessed 2018-09-04.
- [10] Gruhler, A. Prototype, reputation and reward scheme for a cooperative, multi-domain ddos defense. <https://github.com/in0rdr/prototype>. Online, accessed 2018-09-11.

- [11] Gruhler, A. (2018). A Reputation and Reward Scheme for a Cooperative, Multi-domain DDoS Defense. Master's thesis, University of Zurich, Department of Informatics, Communication Systems Group, Zurich, Switzerland.
- [12] Hao, L., Healey, C., and E. Hutchinson, S. (2013). Flexible web visualization for alert-based network security analytics.
- [13] Jacobs, J. and Rudis, B. (2014). *Data-Driven Security: Analysis, Visualization and Dashboards*. Wiley.
- [14] Lab., A. K. Cyberthreat real-time map. <https://cybermap.kaspersky.com>. Online, accessed 2018-09-11.
- [15] Lee, K.-C., Hsieh, C.-H., Wei, L.-J., Mao, C.-H., Dai, J.-H., and Kuang, Y.-T. (2017). Sec-buzzer: cyber security emerging topic mining with open threat intelligence retrieval and timeline event annotation. *Soft Computing*, 21(11):2883–2896.
- [16] Li, X., Wang, Q., Yang, L., and Luo, X. (2011). Network security situation awareness method based on visualization. In *2011 Third International Conference on Multimedia Information Networking and Security*, pages 411–415.
- [17] Mannhart, S. (2018). Mitigation as a Service in a Cooperative Network Defense. Master's thesis, University of Zurich, Department of Informatics, Communication Systems Group, Zurich, Switzerland.
- [18] Marty, R. (2008). *Applied Security Visualization*. Addison-Wesley Professional, 1 edition.
- [19] McKenna, S., Staheli, D., Fulcher, C., and Meyer, M. (2016). Bubblenet: A cyber security dashboard for visualizing patterns. *Computer Graphics Forum (EuroVis '16)*, 35(3):281–290.
- [20] Mckenna, S., Staheli, D., and Meyer, M. (2015). Unlocking user-centered design methods for building cyber security visualizations. In *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8.
- [21] Mortensen, A., Andreassen, F., Reddy, T., Gray, C., Compton, R., and Teague, N. (2018). Distributed-Denial-of-Service Open Threat Signaling (DOTS) Architecture. Internet-Draft draft-ietf-dots-architecture-06, Internet Engineering Task Force. Work in Progress.
- [22] Networks, N. (2016). Zodiac fx user guide. http://forums.northboundnetworks.com/downloads/zodiac_fx/guides/ZodiacFX_UserGuide_0216.pdf. Online, accessed 2018-09-04.
- [23] Oza, N., Fagerholm, F., and Månich, J. (2013). How does kanban impact communication and collaboration in software engineering teams? In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 125–128.

- [24] Raghavan, S. and Dawson, E. (2011). *An investigation into the detection and mitigation of denial of service (dos) attacks: critical information infrastructure protection*. Springer Science & Business Media.
- [25] Rodrigues, B., Bocek, T., and Stiller, B. (2017). Enabling a Cooperative, Multi-domain DDoS Defense by a Blockchain Signaling System (BloSS). In *Demonstration Track*, pages 1–3, Singapore, Singapore. IEEE.
- [26] Shneiderman, B. (1997). A grander goal: A thousand-fold increase in human capabilities. *Educom review*, 32:4 – 10.
- [27] Staheli, D., Yu, T., Crouser, R. J., Damodaran, S., Nam, K., O’Gwynn, D., McKenna, S., and Harrison, L. (2014). Visualization evaluation for cyber security: Trends and future directions. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, VizSec ’14, pages 49–56, New York, NY, USA. ACM.
- [28] Tadda, G. P. and Salerno, J. S. (2010). *Overview of Cyber Situation Awareness*, pages 15–35. Springer US, Boston, MA.
- [29] The GNOME Project (2018). Gnome developer - design principles. <https://developer.gnome.org/hig/stable/design-principles.html.en>. Online, accessed 2018-09-04.
- [30] Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. (2005). iperf: Tcp/udp bandwidth measurement tool.
- [31] Work.Nation. Work.nation: Decentralized skill attestations using uport, ethereum and ipfs. <https://github.com/worknation/work.nation>. Online, accessed 2018-09-11.
- [32] Yelizarov, A. and Gamayunov, D. (2009). Visualization of complex attacks and state of attacked network. In *2009 6th International Workshop on Visualization for Cyber Security*, pages 1–9.
- [33] Zargar, S. T., Joshi, J., and Tipper, D. (2013). A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys Tutorials*, 15(4):pp. 2046–2069.
- [34] Zlomislíć, V., Fertilj, K., and Sruk, V. (2017). Denial of service attacks, defences and research challenges. *Cluster Computing*, 20(1):661–671.

Abbreviations

AFP	Alexander's fifteen properties
AS	Autonomous System
BloSS	Blockchain Signaling System
CSA	Cyber Situation Awareness
SA	Situation Awareness
SPA	Single-Page Application
DDoS	Distributed Denial of Service
SDN	Software-Defined Network
UI	User Interface

List of Figures

3.1	BloSS collaborative defense system	13
3.2	Use case diagram with example AS400 and AS500	14
3.3	Component Diagram of the BloSS architecture	15
3.4	Schematic view of the user interface	16
3.5	<code>bloss-node</code>	17
3.6	<code>bloss-core</code>	18
3.7	Sequence Diagram for Request Mitigation (Target perspective)	20
3.8	Sequence Diagram for Mitigation Requests (Mitigator perspective)	22
3.9	State Diagram for Mitigation Requests (Mitigator perspective)	23
3.10	State Diagram for Request Mitigation (Target perspective)	24
4.1	Schematic representation of hardware system architecture	25
4.2	Photo of hardware system to demonstrate BloSS	26
4.3	Component diagram of <code>bloss-node</code>	31
4.4	<code>bloss-dashboard</code> with all possible states of an <code>attack_report</code>	36
4.5	abstraction of <code>MitigationRequest.vue</code> and <code>AttackAlarm.vue</code>	38
4.6	Screenshot of detail modal dialog	40
4.7	Screenshot of <code>bloss-dashboard</code> overview	40
4.8	Screenshot of <code>bloss-dashboard</code> detail modal dialog	41
5.1	<code>bloss-dashboard</code> UI running on controller400 with all possible states of an example <code>attack_report</code> in the <code>ALARMS_TAB</code>	44

5.2	bloss-dashboard UI running on controller600 with all possible states of an example attack_report in the REQUESTS_TAB	45
B.1	Controller 400 view loaded, waiting for bloss-node to retrieve statuses . .	63
B.2	Controller 400 view loaded, all services active and running	64
B.3	UC1: Step 2: Display new alarm toast	64
B.4	UC1: Step 3: New alarm	65
B.5	UC1: Step 4A: Alarm ignored	65
B.6	UC1: Step 4B: Request Mitigation Requested	66
B.7	UC1: Step 5A: Request Mitigation declined	66
B.8	UC1: Step 5B: Request Mitigation accepted	67
B.9	UC1: Step 6: Request Mitigation in progress (blocking)	67
B.10	UC1: Step 7: Request Mitigation successful	68
B.11	UC2: Step 1: Display new Mitigation Request	68
B.12	UC2: Step 2A: Mitigation Request declined	69
B.13	UC2: Step 2B: Mitigation Request accepted	69
B.14	UC2: Step 3: Mitigation Request in progress	70
B.15	UC2: Step 4: Mitigation Request successful	70

List of Tables

Appendix A

Contents of the CD

- **abstract.txt** - abstract in English
- **blockchain-signaling-system** - contains sub-folders containing relevant repositories
- **intermediate-presentation.pdf** - contains the slides for the intermediate presentation.
- **thesis.pdf** - this thesis report as PDF
- **tex** - contains L^AT_EX sources of this thesis, including all figures
- **zusammenfassung.txt** - abstract in German.

Appendix B

Use Case Figures

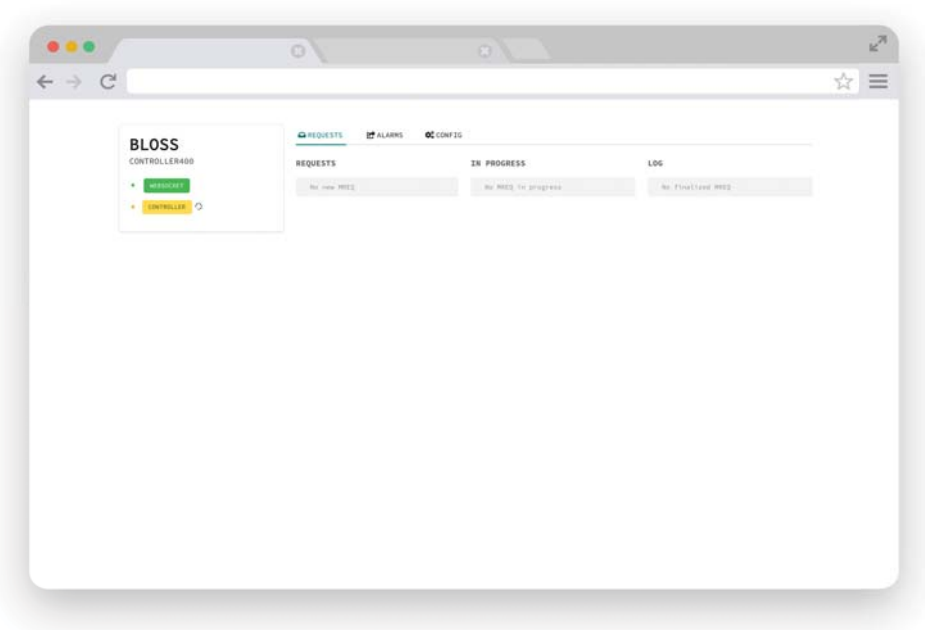


Figure B.1: Controller 400 view loaded, waiting for `bloss-node` to retrieve statuses

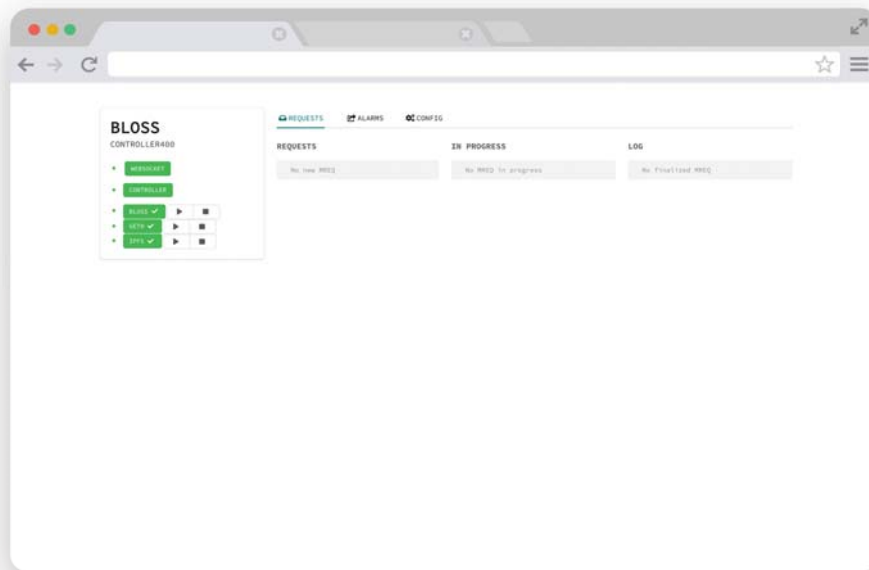


Figure B.2: Controller 400 view loaded, all services active and running

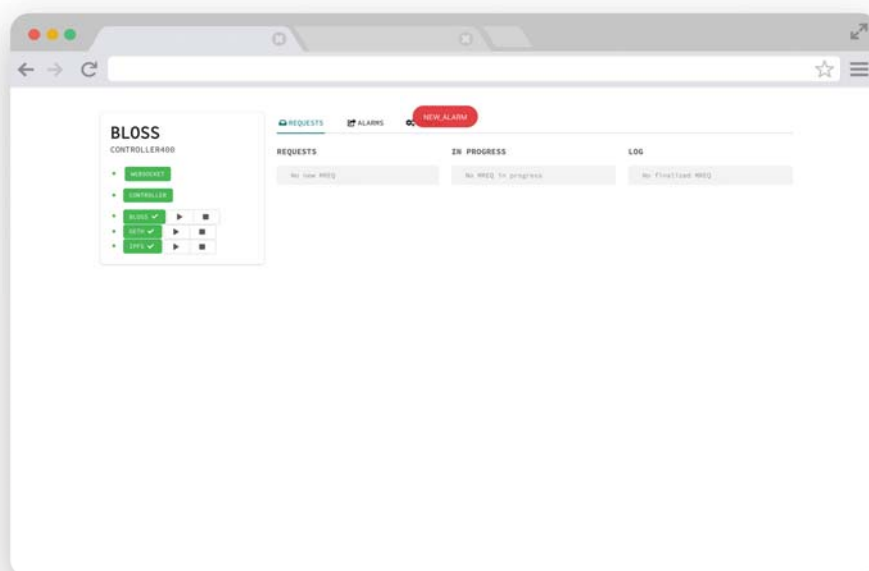


Figure B.3: UC1: Step 2: Display new alarm toast

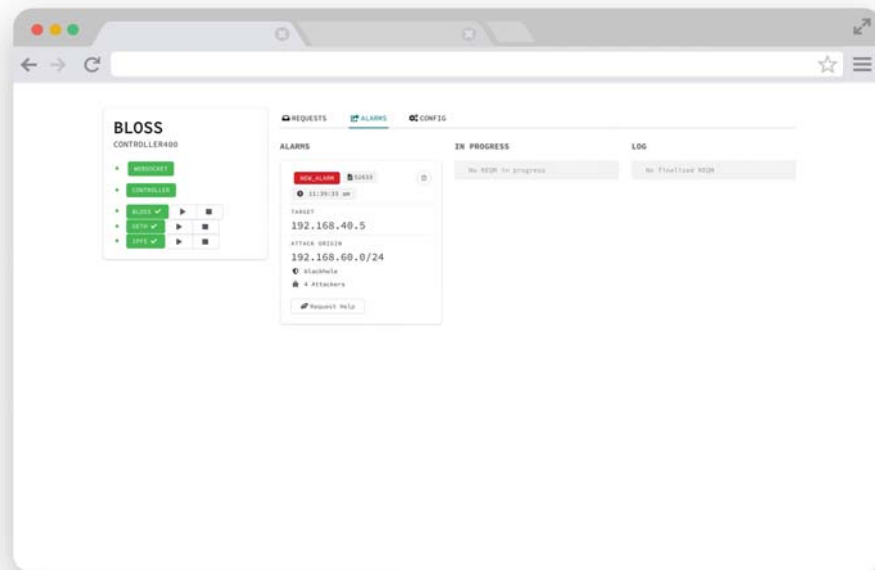


Figure B.4: UC1: Step 3: New alarm

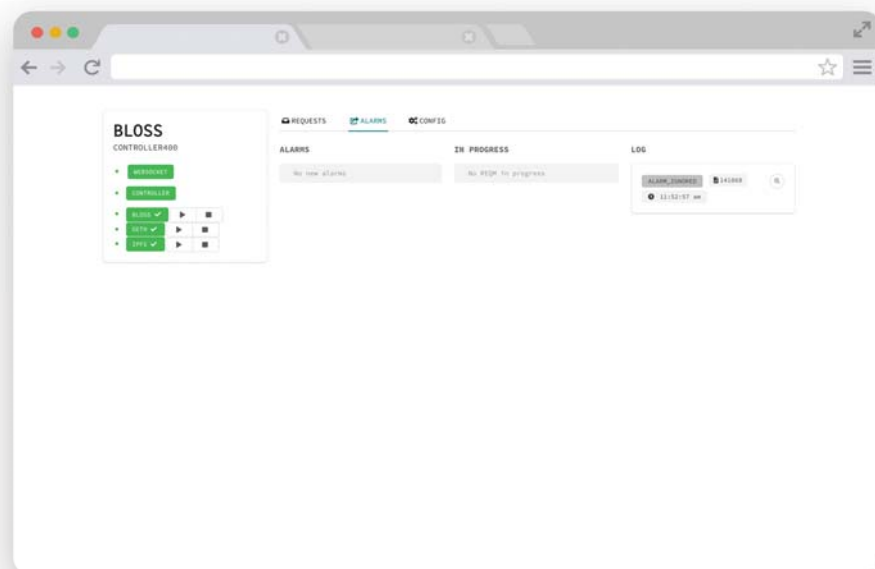


Figure B.5: UC1: Step 4A: Alarm ignored

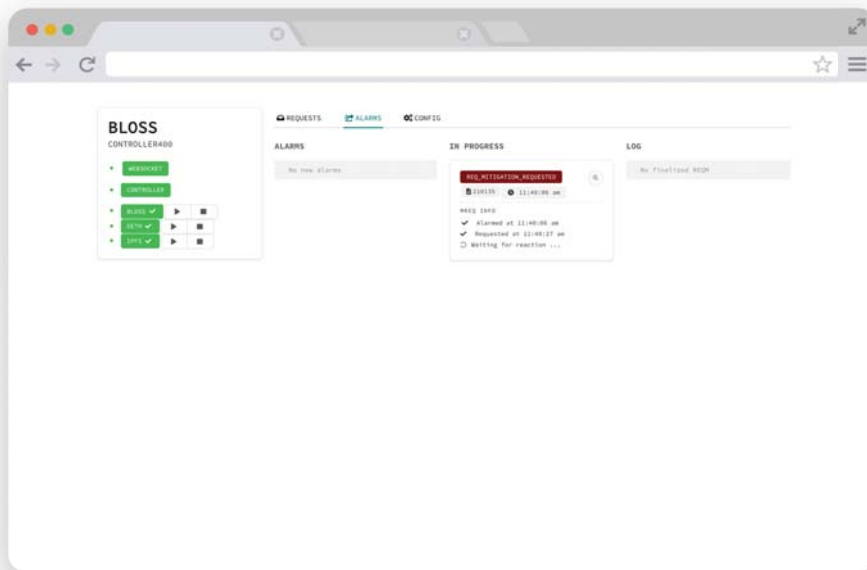


Figure B.6: UC1: Step 4B: Request Mitigation Requested

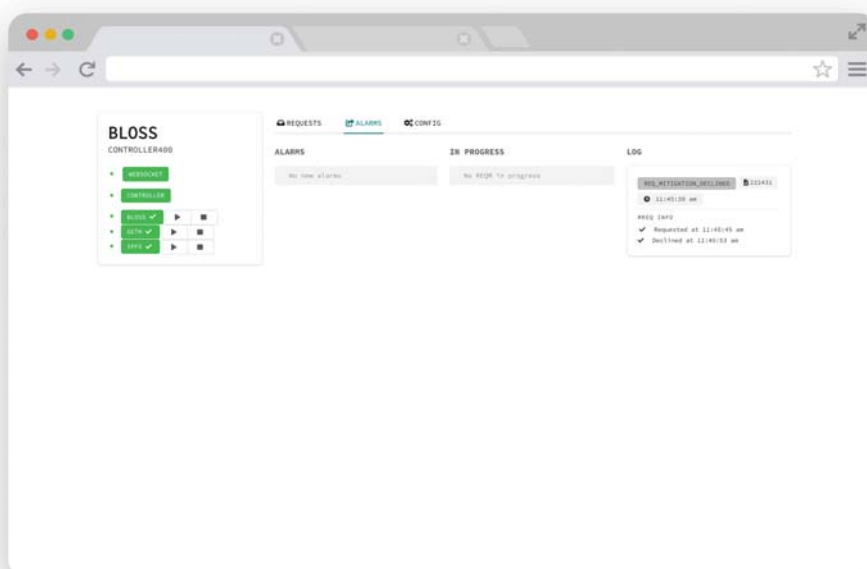


Figure B.7: UC1: Step 5A: Request Mitigation declined

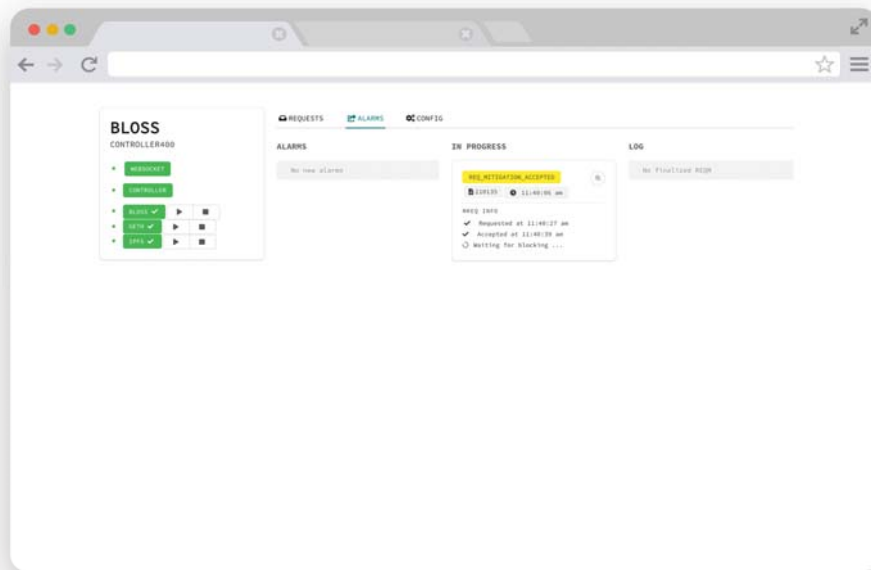


Figure B.8: UC1: Step 5B: Request Mitigation accepted

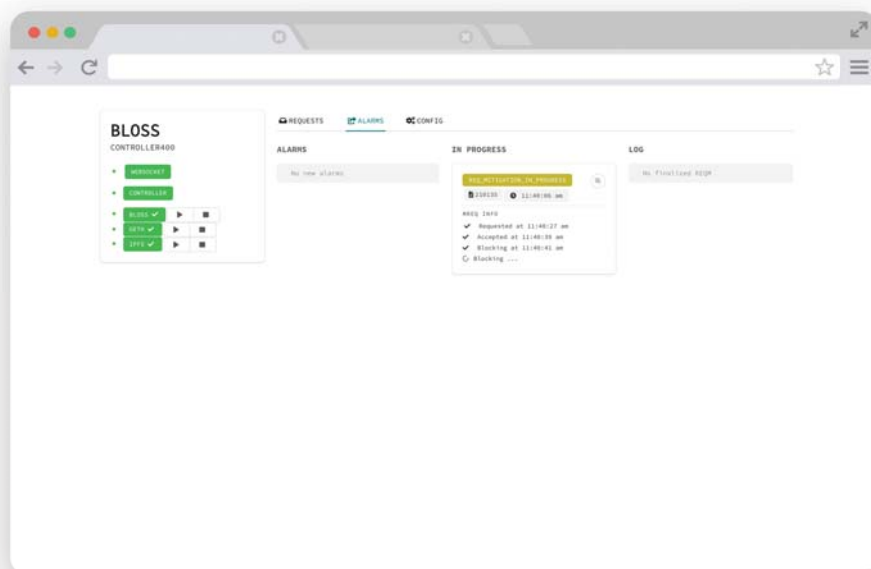


Figure B.9: UC1: Step 6: Request Mitigation in progress (blocking)

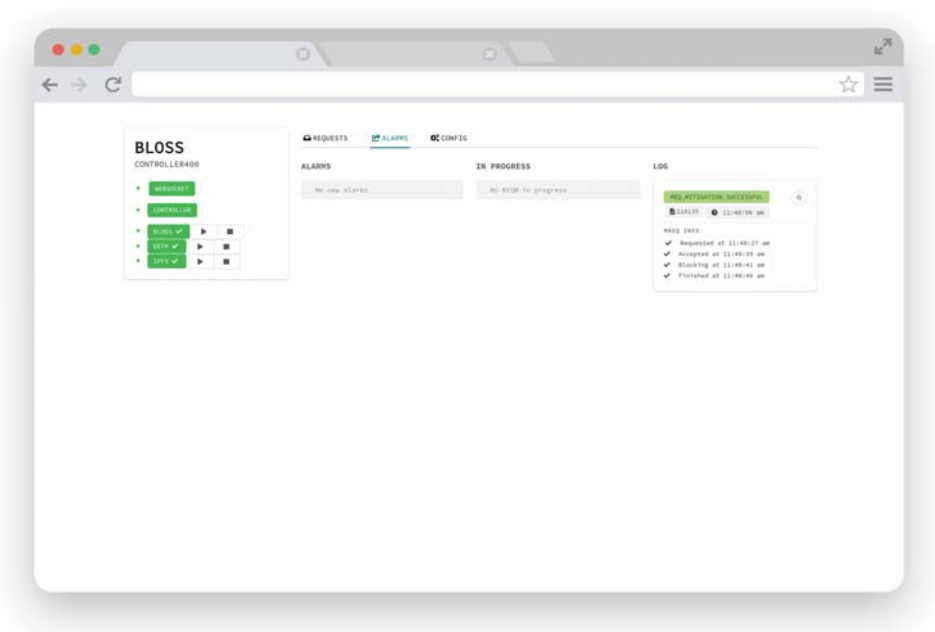


Figure B.10: UC1: Step 7: Request Mitigation successful

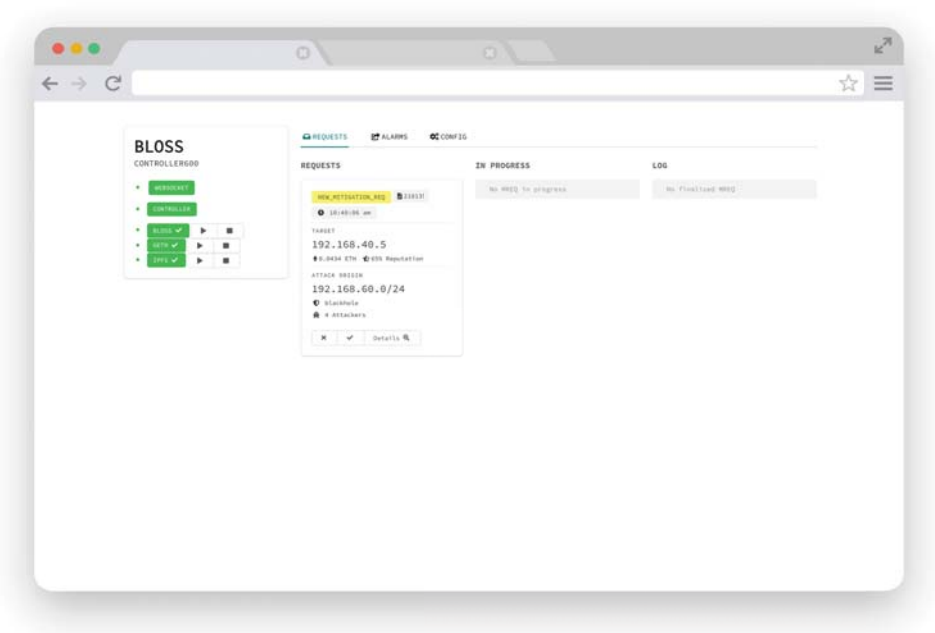


Figure B.11: UC2: Step 1: Display new Mitigation Request

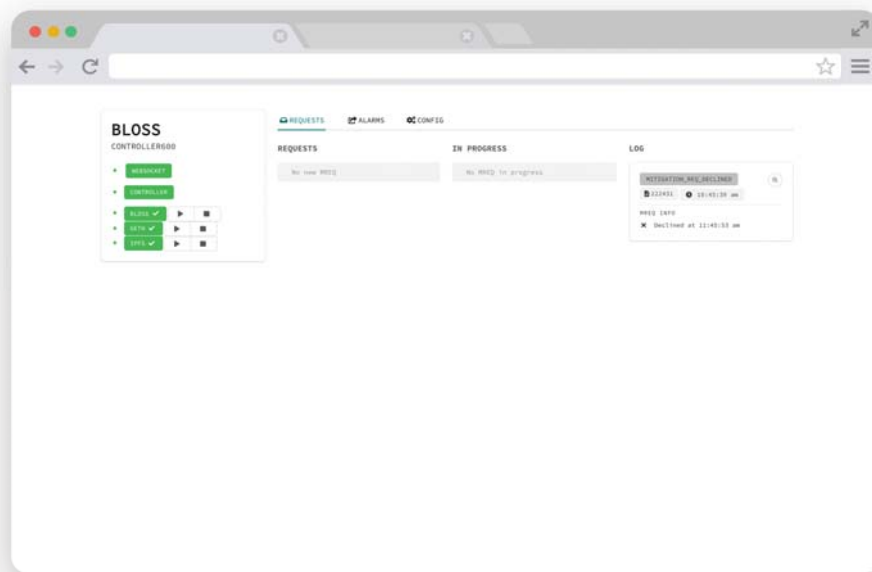


Figure B.12: UC2: Step 2A: Mitigation Request declined

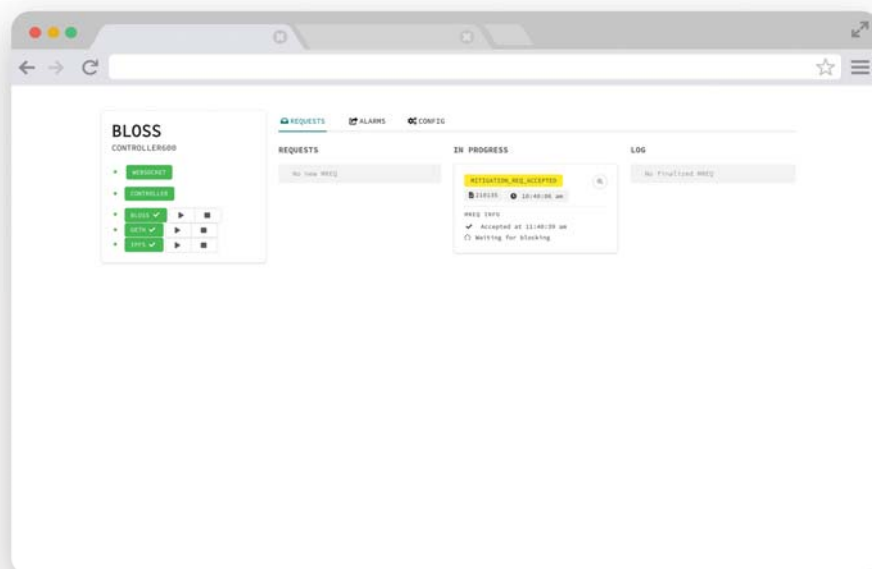


Figure B.13: UC2: Step 2B: Mitigation Request accepted

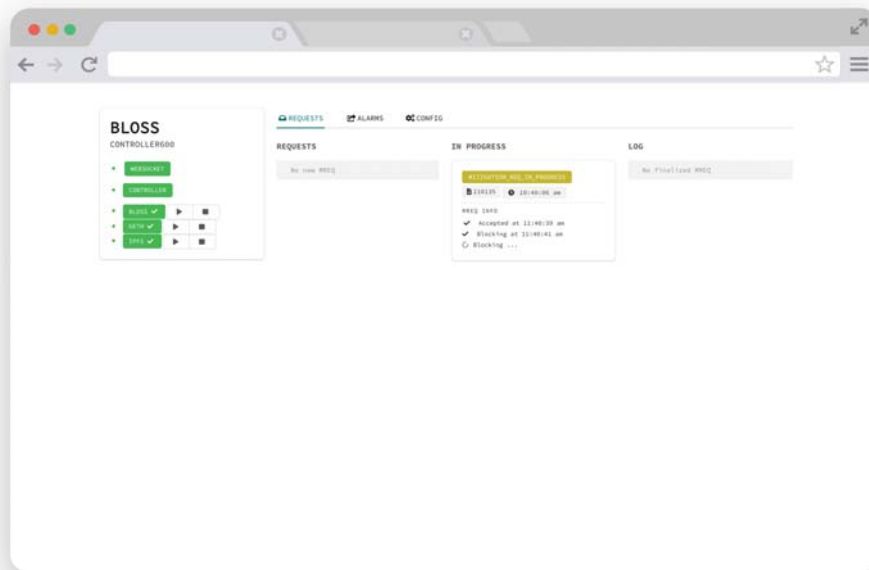


Figure B.14: UC2: Step 3: Mitigation Request in progress

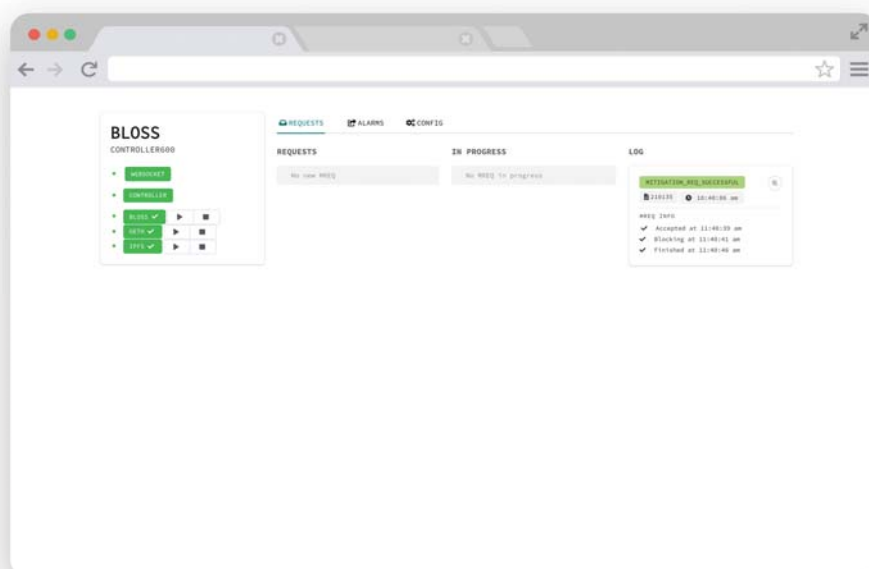


Figure B.15: UC2: Step 4: Mitigation Request successful