Hochschule für angewandte Wissenschaften München

Fakultät für Informatik und Mathematik

Master's Thesis

Master of Science

in course of studies Informatik

Secure IoT Bootstrapping: A BRSKI Extension for Constrained Devices

---

Author:	Julian Krieger
Matriculation number:	01084121
Submission Date:	17. January 2025
Supervisor:	Prof. Dr. Thomas Schreck
Advisor:	Prof. Dr. Stefan Wallentowitz

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 17. January 2025 Julian Krieger

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. Thomas Schreck, for his invaluable guidance and unwavering support throughout this journey. His insights and encouragement have been instrumental in shaping this work and pushing it to its highest potential. I am profoundly thankful to my colleague, Tobias Hilbig, whose unparalleled advice and collaborative spirit have been a source of inspiration and motivation. Special thanks go to Steffen Fries at Siemens, who demonstrated exceptional patience and generosity in answering my countless questions about the BRSKI protocol family. His willingness to share his expertise was pivotal in deepening my understanding and advancing this work. I also would like to thank Prof. Dr. Stefan Wallentowitz and Prof. Dr. Peter Trapp for allowing and encouraging me to switch my master's studies to IT-Security, even though the deadlines for registration had long passed. When I came to Munich, I did not originally intend to pursue this field, but I fell in love at first sight when I randomly visited a lecture. Their support enabled me to embark on this path, which has since not only become my passion, but has also led me to pursue a PhD in the field. Finally, I want to thank my family and friends for their unwavering support and encouragement.

To all who have supported me during this journey, whether through academic guidance, moral support, or inspiration, I extend my heartfelt appreciation. This thesis would not have been possible without your contributions.

Abstract

The integration of constrained IoT devices into enterprise networks presents significant challenges, particularly when these devices lack initial network connectivity due to security constraints. This thesis addresses these challenges by investigating existing bootstrapping solutions and developing a novel approach tailored to the unique requirements of such devices. After an analysis and comparison of various enrollment protocols, none fully met the constraints and connectivity requirements. To bridge this gap, this work proposes and implements cBRSKI-PRM, a fusion of two promising protocols: BRSKI-PRM and cBRSKI. This hybrid approach enables a pledge-passive, one-touch enrollment process by leveraging Bluetooth Low Energy (BLE) as an out-of-band communication channel. This process allows devices without initial network connectivity to enroll securely.

To validate the feasibility of cBRSKI-PRM, a prototype was implemented using Rust and deployed an Android phone and on the ESP32 platform. The security of the protocol was evaluated by analyzing potential attack vectors introduced during the design process and assessing the changes made to the original protocols. This evaluation demonstrated the solution's robustness and its effectiveness in securely enrolling constrained IoT devices into enterprise networks, addressing a critical gap in IoT device management.

Contents

1 Introduction **1**
- 1.1 Motivation & Relevancy . . . 1
- 1.2 Scope . . . 2
- 1.3 Research Questions . . . 3

2 Related Work **5**
- 2.1 Pre-provisioned Keys . . . 5
- 2.2 Embedded Attestation . . . 6
- 2.3 ACME . . . 6
- 2.4 Enrollment over Secure Transport . . . 7
- 2.5 BRSKI . . . 7

3 Background **9**
- 3.1 Enrollment . . . 9
- 3.2 Bluetooth Low Energy . . . 10
- 3.3 The Rust Programming Language . . . 11
- 3.4 BRSKI-PRM . . . 13
- 3.5 cBRSKI . . . 16

4 Requirements **17**

5 Architecture **21**
- 5.1 Pledge . . . 21
- 5.2 Registrar-Agent . . . 22
- 5.3 Domain-Registrar . . . 22
- 5.4 MASA . . . 23
- 5.5 Proposed Protocol Extension . . . 23

6 Implementation **27**
- 6.1 BRSKI-PRM Prototype . . . 27
- 6.2 cBRSKI-PRM . . . 30
  - 6.2.1 Switching from JSON to CBOR . . . 30
  - 6.2.2 Bluetooth Low Energy . . . 34
- 6.3 Android Registrar-Agent . . . 37
- 6.4 ESP32 Prototype . . . 40
  - 6.4.1 Research and Setup . . . 40
  - 6.4.2 Pre-Enrollment Status Query . . . 41
  - 6.4.3 ESP32 firmware . . . 42

7 Evaluation **45**
- 7.1 Comparison of BRSKI-PRM and cBRSKI-PRM . . . 45

7.2 Attack Vectors 47

8 Discussion & Future Work **49**

9 Conclusion **51**

Bibliography **55**

Glossary **61**

A Listings **63**

B Tables **71**

C Certificates **73**
C.1 Pledge Certificate 73
C.2 Registrar-Agent Certificate 74
C.3 Registrar Certificates 75
C.4 MASA Certificates 78

D Images **83**

1 Introduction

The number of Internet-of-Things (IoT) devices worldwide is expected to grow to more than 29 billion devices in 2030 [71] [63]. With the rapid proliferation of IoT devices both in business and home automation, security has become increasingly important. A valid threat and protection model is required not only to strengthen the security of existing devices and the entire management process like bootstrapping/enrollment and device off-boarding processes. Ideally, devices must first establish trust by providing identity and authenticity proofs. To establish trust, peripheral devices ask their users for authentication via a password or biometric information connected with an institutionally known and verifiable identity. Embedded devices cannot rely on such procedures because they are shipped with pre-installed firmware to the customer and must prove their authenticity via automation. They are low-powered devices that usually lack means of identity beyond their physical network adapter address, vendor-chipset combinations, or serial number, all of which are easily forgeable attributes that cannot be relied upon for encryption. This creates a principal problem when evaluating the initial device installation and bootstrapping process: To gain enough trust to enroll, each device must provide a unique proof of authenticity to the trust-verifying entity. This calls for a trustable hardware-backed attestation token that differs from the aforementioned identifiers. The attestation token must be unique to the device, must be unforgeable, and must prove its authenticity to the trust-verifying entity. Meeting the particular security requirements of the enrollment process is crucial to the overall security of the device.

1.1 Motivation & Relevancy

Internet of Things devices face security challenges unique to their application domain [1]. Compared with conventional computing devices, IoT devices are often low-powered, have limited storage space, and are frequently deployed in large numbers. They are often tiny, battery-powered devices that may be deployed in remote locations, making them difficult to manage. Regularly ordered in large numbers from foreign vendors, they are a prime candidate for being the weakest link in a network [57] or are targets in supply chain attacks [53] [67]. Furthermore, one must assume that the shipment could contain malicious parasitic companions, or that a malevolent on-site technician could replace a unit with a rogue device [42] [50]. In order to increase security, devices should be required to provide proof of their authenticity and identity before they are permitted to access the network. In essence, each corresponding device needs to build a trust relationship with their communication partner.

Customers often have little influence on the manufacturing process of these devices, and as such, they cannot rely on a vendor to provide sufficient security guarantees for hardware or software. To secure these devices, a security model is required to be established, which

the customer controls, and which is independent of on-site personnel and end users. To guarantee security right from the beginning, a secure enrollment scheme must be created that is built with security at its forefront and which is as automated as possible. It should also withstand supply chain attacks and rogue actors eavesdropping. While there exist some schemes for secure bootstrapping of consumer or high-powered devices, hardware-limited IoT devices bring up unique challenges that need to be addressed. Creating a secure enrollment scheme specifically focused on the hardware constraints of IoT devices is the main motivation for this work.

1.2 Scope

The focus of this work is limited to securing the on-site enrollment process. It is also the first step entirely in the control of the device customer. A suitable enrollment scheme, which best fits the unique requirements to be developed and discussed later, will thus be found and built upon. To make the bootstrapping process as secure as possible, all participating entities need to be able to provide guarantees of identity and proof of key possession to each other. For example, the domain registration authority may only provide network access to a target device if it can prove its identity. Identity-proving and encryption material need to be available from start to finish of the bootstrapping process. Only then can the required security assurances be met: Any IoT device that enters and exists within the customer domain must be known before and during its entire existence, enabling the keeping of records throughout the device lifecycle. Furthermore, a well-defined process allows a granular management of communication permissions as well as a way to update identification and transmission material, such as certificates required for connectivity in enterprise networks.

To enable this workflow at scale, a way of automating the entire enrollment process as much as possible is needed. Providing a zero-touch or one-touch enrollment process is necessary to guarantee bootstrapping with minimal friction. To accomplish the bootstrapping process, all stakeholders must act with security in mind, from supply chain manufacturers providing read-only attestation keys to service technicians involved in the enrollment process and finally technical customer domain managers who contribute the essential infrastructure. Ideally, each partaking entity needs to ensure to all other communication partners of their identity before any data exchange takes place. Data transfer must be encrypted at best, and integrity must be provided by signatures when encryption is not possible at the specific point in time. Limited machines work with only tiny amounts of stack space and restricted hardware access. Many are not fitted with real-time clocks and operate with minimal power. In large scales, embedded computing systems come with software flashed at the end of the manufacturing process. These limitations impose security challenges, like the restricted choice of initial network access strategies or inadequate storage space for public key material and certificates.

1.3 Research Questions

The primary focus of this study is the development of a secure and proven solution for the enrollment process of embedded devices. The general use case has been described in the previous section, so the specific requirements and their needs must now be inferred. The following research questions are presented as a comprehensive overview of the challenges involved.

RQ1 What are the feature requirements for a networkless, minimal-touch enrollment protocol for hardware constrained devices? The required scalability, security, seamlessness, and networkless nature of the enrollment process were considered. It must also be taken into account that constrained IoT devices often lack the computational power to perform complex cryptographic operations. In addition, they are commonly limited in terms of memory and storage, which can make it difficult to store and manage cryptographic keys and public key material.

RQ2 What extensions to an existing protocol provide compatibility for hardware constrained devices? To answer this question, existing enrollment protocols had to be explored first. The extensions needed that are suitable for fulfilling the requirements were then considered. Next, the security guarantees, scalability, and complexity of the protocols are discussed. Furthermore, the compatibility of the protocols with embedded devices and the ease of implementation were also explored. Finally, the extensions needed to provide compatibility with a specific embedded platform were planned.

RQ3 How can the proposed solution be implemented on low powered hardware? To arrive at the answer to this research question, the challenges and considerations unique to embedded hardware were explored. Various embedded platforms were compared based on hardware assessments that featured limitations, constraints, and advantages. Next, a specific hardware framework was selected, for which a prototype was developed within the constraints of the requirements emerging from choosing a protocol specification document in RQ1. To evaluate these research questions, the base enrollment protocol was compared to the novel solution. Potential new attack vectors that might have been introduced by the extension of the original protocol were also considered. Both of these are further explained in Chapter 7

2 Related Work

This section provides the context for this study by exploring previous studies on secure device enrollment. Initially, research is highlighted that aligns with the stated objective of securely incorporating devices into a network. Next, previous work is explored which focuses on device authentication but not necessarily in the context of bootstrapping. Finally, a short overview is provided of published enrollment solutions directly related to this study, and from which inspiration was drawn for the resulting solution.

2.1 Pre-provisioned Keys

A way to provide security during the bootstrapping process is to ship IoT devices with pre-shared keys, where a private key is securely stored on the device and a corresponding public key is shared with the network operator at some arbitrary point in time. Shipping a device with pre-shared keys has, however, some drawbacks: Pre-provisioning a private key creates the need to install it at some time during the manufacturing process. The process of sharing the corresponding public key with the network operator can be implemented in various ways, such as printing the key on a sticker, storing it in a QR code, or transmitting it over a secure channel. Some of these methods are more secure than others, and the choice of method depends on the particular case and the security requirements of the given network. A single secret alone is not enough to provide device identity and trustworthiness, because it carries the risk of being leaked at some point in the supply chain [68]. Ideally, these private keys must be unique to each device, and since the resulting public key is unique as well, a potentially large list of device public keys must be shared with the customer, which requires substantial management effort by the network operator. A leaked secret opens up security hazards, which may allow malicious actors to eavesdrop, and which potentially enables the installation of foreign hardware. On initial network access, the network operating service may offer authenticating credentials to the device, often in the form of an X.509 certificate. When the enrollment process relies solely on pre-shared keys, devices have no way to verify the authenticity of the network operator using this payload. Therefore, in the context of device enrollment, some standards define initial communication to work without client-side authentication, instead opting for a strategy called Trust On First Use (TOFU) [7] [65].

When IoT devices are enrolled with pre-provisioned keys, some strategies employ decentralized PSK solutions to provide a secure and decentralized way to manage device identities and keys. In *Ethereum for Secure Authentication of IoT using Pre-Shared Keys (PSKs)* [31], the authors proposed a system in which IoT devices store their private key, and the public key exists in a decentralized fashion on the *Etherium* blockchain. By making use of smart contracts, the system uses distributed ledger technology to provide a secure and tamperproof process to manage device identities. With this, the authors aim to

provide a secure and reliable strategy to authenticate IoT devices in a decentralized manner, without the need for a public key infrastructure or a centralized certificate authority. They validated their approach by implementing a prototype, that they evaluated using a simulation approach.

2.2 Embedded Attestation

To gain trust and guarantee device integrity, embedded devices can ship with so-called *attestation keys*. Various protocols exist that aim to provide a scalable proof-of-integrity process. They consider that embedded devices often ship into decentralized target domains susceptible to imperfect conditions, network disruptions and device outages. One example is PASTA [37], a distributed attestation protocol for autonomous embedded systems, which defines a remote key attestation protocol for millions of interconnected low-powered embedded devices by introducing a concept of collective attestation, wherein multiple provers verify groups of devices forming multisignature pools, thereby avoiding a computationally intensive single source of truth. In essence, most attestation solutions share one feature: They rely on guaranteeing trust through a remote attestation key with secure properties, wherein such an attestation key is a private cryptographic key generated by an entity that in turn, can prove the key's ownership to a remote party. The key is best generated and stored inside a secure enclave isolated from the main processing unit [72] [43] [66]. Providing a valid attestation key is crucial in order to implement any secure enrollment protocol, where device integrity and authenticity must be guaranteed.

2.3 ACME

Automatic Certificate Management Environment (ACME) [2] is a protocol designed to automate the process of issuing and managing digital certificates, primarily for securing websites using HTTPS. In the context of enrollment, ACME can be used in conjunction with different tools to automate the provisioning process of device certificates. For example, *certbot* [23] is a popular tool that uses the ACME protocol to automate the process of obtaining and renewing certificates for web servers. X.509 certificates used for HTTPS are not that different from device attestation certificates that can be used for TLS and network access. Therefore, the initial aim was to adapt ACME to handle the provisioning of device-identifying certificates. According to the specification document, clients can request and retrieve certificates from a central service in an automated manner. However, they must initially establish a Dynamic Host Configuration Protocol (DHCP) connection and employ a discovery scheme like DNS Service Discovery (DNS-SD) to find the central service. Hence, for devices without a network connection, this approach is not feasible. Introducing a proxy device that acts as a bridge between the device and the central service could theoretically overcome this limitation. Taking this approach would, however, introduce additional complexity, because the proxy exchange would need to be planned securely, which is nontrivial and out of scope for this work.

2.4 Enrollment over Secure Transport

"Enrollment over Secure Transport (EST)" defines a protocol for secure certificate enrollment and management using HTTPS as the transport mechanism [52]. The main purpose of EST is to provide a mechanism for devices to automatically enroll and obtain X.509 certificates, which are used in various security protocols, such as TLS/SSL. EST defines communication between an enrollment server and clients, which are preconfigured with key material for mutual authentication and authorization. It supports mutual trust verification based on a variety of schemes, such as pre-shared secrets or trust anchor information in the form of public key certificates. Clients initiate a secure communication session via an encrypted channel. They then follow the protocol standard by requesting information from EST services distinguished by URI paths and parameters. When using certificates for authorization, a certificate authority server *co-located* with the EST server issues an end-entity certificate with a built-in device identifier upon successful enrollment. EST is primarily defined for HTTP-only communication in the specification. The certificates issued in EST are PKCS7 objects called Cryptographic Message Syntax (CMS) documents, which are cryptographically protected messages – either by signature, digest or encryption [32]. The domain certificate authority can certify authorization privileges for the Pledge's certificate request by analyzing the security credentials used in the TLS connection. While EST can enroll devices into a local domain with identifying X.509 certificates, devices must already have joined a network previously before enrollment can begin.

2.5 BRSKI

In the realm of secure device bootstrapping, Bootstrapping Remote Secure Key Infrastructure (BRSKI) has emerged as a foundational protocol [51]. BRSKI defines the secure initiation process of devices into pre-existing network environments. This section gives a brief overview of existing research and implementations surrounding BRSKI, BRSKI enhancements and existing work in the context of IoT devices. Post-bootstrapping zero-trust strategies will also be looked at to fortify device security after successful enrollment.

BRSKI is a fairly recent protocol as of time of writing, having been published as RFC 8995 and which was recently formally verified [62]. It defines a zero-touch bootstrapping process of a so-called Secure-Key-Infrastructure using a three-piece combination of manufacturer-provided components and a customer-domain authority. The former consist of a pre-installed manufacturer device certificate, a manufacture provided Initial Device Identifier and a vendor managed authorizing service. For them, BRSKI relies on other internet standards for the specification of data transfer formats and endpoints. The architecture and enrollment process defined within BRSKI is based upon Enrollment over Secure Transport (EST), using its capabilities for the request and issuance of domain specific device certificates [51] [52]. The process of bootstrapping a device comes with the explicit goal of deploying secure key material in compliance with the local domain onto a device.

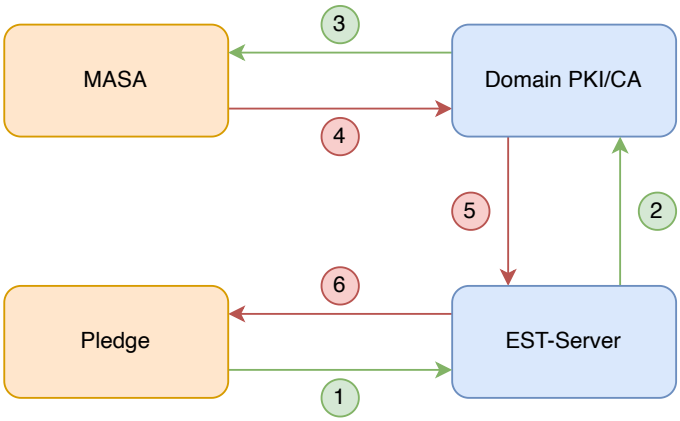

Figure 2.1: *The architecture behind RFC 8995 [51].*

Correctly implemented, BRSKI provides the benefit of a fully automated device enrollment process where security be guaranteed for all bootstrapping operations in the entire process chain. The process behind BRSKI is shown in Figure 2.1 and works as follows: Once powered up and connected to the local network, the device – or Pledge – starts the bootstrapping process. By making use of the manufacturer provided PKIX certificate, the Pledge signs a request for an imprinting token. After discovering the domain's EST service via discovery mechanisms similar to DHCP or DNS-SD, the Pledge sends an initial TLS handshake to the Registrar's EST service. This TLS handshake includes a certificate belonging to the Domain-Registrar's PKI infrastructure, which the Pledge now uses among other data to form a request for an enrollment token, which is sends to the Registrar's EST service (1). This EST service is a service as defined in RFC 7030 [52], which has been extended for functionality in BRSKI. Note that the Registrar's certificate is only provisionally accepted at this time, as the Pledge has not yet the ability to verify the Registrar's identity. The Registrar in turn makes an informed decision on whether to accept this token request or not (2). If the token request is accepted, the Registrar sends a request, for an imprinting token to a manufacturer provided service, called Manufacturer Authorized Signing Authority (MASA) (3). It uses the Pledge's identifier, which is a certificate carrying a unique extension, to locate the correct MASA service. The Registrar also includes a domain specific trust chain in the form of a certificate chain, which includes the CA certificate that signed the EST server's TLS certificate. This imprinting token is a signed artifact, which is known in BRSKI and similar protocols as a *voucher* [73]. The MASA creates this Voucher token and signs it before sending it back to the Domain-Registrar (4), examined and inspected for possible tempering (5), before being sent to the Pledge (6). The Pledge can now both authenticate the Voucher object and the Registrar's identity, since the Voucher token contains sufficient information to verify the Registrar's certificate. Since all parties have now verified each other's identity, the Pledge can now request a domain certificate from the Registrar's EST service as defined in RFC 7030 [52].

3 Background

An overview of the background information necessary to understand the research presented in this thesis is provided in this chapter. First, details about the enrollment process and the challenges that arise when bootstrapping IoT devices will be discussed. Next, Bluetooth Low Energy (BLE) will be introduced, and the knowledge needed to understand the parts of the Bluetooth Low Energy (BLE) protocol used in this work will be detailed. After that, the Rust programming language will be introduced, and its suitability for developing secure IoT applications will be explained. Finally, the BRSKI protocol and its constrained variant Constrained Bootstrapping Remote Secure Key Infrastructure (cBRSKI) will be introduced.

3.1 Enrollment

IoT devices, designed for specialized tasks, are often shipped and installed in large quantities across diverse environments. They commonly lack user interfaces, which introduces challenges in configuring them for initial setup and integration into existing networks. While bootstrapping strategies in the past may have relied on manual processes or the manufacturer-provided installation of default credentials, these approaches are no longer viable in today's threat landscape. Consequently, they necessitate innovative approaches to secure identity bootstrapping, a critical process for their integration into larger systems. Bootstrapping involves a sequence of steps, beginning with the verification of a device's identity, followed by the issuance of cryptographic key material, and the distribution of essential configuration data [74]. This process not only ensures the device's secure enrollment but also plays a pivotal role in maintaining the integrity and scalability of enterprise systems. Without effective bootstrapping schemes, enterprises face challenges in integrating IoT devices while preserving robustness and adaptability in their infrastructure.

In an enterprise setting, securing the bootstrapping process is of utmost importance, as any compromise could expose sensitive systems and data to threats. Device tracking often complements the process, leveraging a registration service hosted within the customer's domain to monitor and manage enrolled devices. To streamline this, vendors typically embed identification information in devices during the manufacturing stage. This could include an attestation key stored within a secure enclave, ensuring that each device has a unique and immutable identifier. By avoiding shared keys, the risk of unauthorized access due to key compromise is mitigated. Once devices are delivered, customers can extract this identification information and cross-reference it with a vendor-maintained database of legitimate devices. Such verification ensures that only authenticated devices are integrated into enterprise systems.

Vendors may provide this identification information in multiple ways. It could accompany physical device shipment in physical form, or be made accessible through an externally hosted service designed for automated verification [26]. The latter method enables real-time validation, allowing vendors to vouch for the authenticity of their devices during the bootstrapping phase. This approach ensures that devices customers incorporate into their networks meet the required security standards and originate from trusted sources. Additionally, this system facilitates easier lifecycle management by maintaining visibility into the devices' origin and ongoing operational status.

To enhance trust, many bootstrapping schemes employ X.509 end-entity certificates as attestation mechanisms [26] [52] [68]. These certificates provide a standardized way to establish a trust relationship between the device and the customer's infrastructure. Vendors can include the issuing CA certificate, allowing customers to verify the certificate chain and ensure its authenticity. Depending on the enterprise architecture, customers can further provision devices with domain-specific certificates and key materials, granting them access to secured services. Once bootstrapping is successfully completed, this cryptographic information can be used to establish secure connections, enabling encrypted communication and ensuring the confidentiality, integrity, and authenticity of transmitted data.

Beyond enrollment, effective bootstrapping also lays the groundwork for ongoing device management. It can offer additional features such as firmware updates, access control, and secure decommissioning when devices reach the end of their lifecycle. By embedding trust at the core of the process, enterprises can future-proof their IoT integrations, ensuring scalability and resilience in dynamic and evolving digital environments.

3.2 Bluetooth Low Energy

BLE (Bluetooth Low Energy) is a wireless communication protocol specifically designed for efficient, short-range data transmission [30]. It emphasizes low power consumption, which makes it ideal for battery-operated devices and applications requiring long operational lifespans. Unlike traditional Bluetooth, BLE is optimized for quick and lightweight data exchanges, enabling seamless communication without requiring device pairing. This characteristic makes it especially well-suited for applications such as real-time control systems, wearable technology, IoT devices, and smart infrastructure. BLE's ability to provide reliable, low-latency communication in these contexts contributes to its widespread adoption across industries, including healthcare, fitness, home automation, and industrial automation. One of BLE's key strengths lies in its cost-effective scalability and flexibility. Devices leveraging BLE can form complex mesh networks, enabling robust communication across multiple nodes. These networks are particularly useful in smart city implementations, where thousands of sensors and devices need to interact efficiently. BLE also supports broadcasting, allowing a device to send data to multiple receivers simultaneously, which is valuable for beacon-based applications like location tracking, marketing, or proximity alerts.

BLE organizes data using Generic Attribute Profile (GATT) [30, G], which structures information into a hierarchy of *attributes*. Attributes serve as fundamental data units, each identified by a universally unique identifier (UUID). Within GATT, two primary elements are *services* and *characteristics* [30, G-2.6.2]. A GATT service groups related characteristics, where each service provides a logical grouping of data points, simplifying the organization and interpretation of transmitted information. GATT characteristics, on the other hand, represent individual data points within a service [30, G-2.6.4]. They include specific values or attributes, such as the current heart rate or a temperature reading. Characteristics often include additional metadata, such as permissions (read, write, or notify) and *operations* that define how they can be accessed or modified [30, G-3.3.1.1]. These attributes enable precise control over device interactions, ensuring secure and reliable data exchange. Notifications allow a device to be informed of changes in real-time without repeatedly querying for updates, thus improving efficiency and reducing latency. Indications offer a similar function but include an acknowledgment mechanism, ensuring that the data was received successfully. These operations are essential in dynamic, real-time applications where responsive communication is critical. Moreover, BLE incorporates robust security mechanisms, such as encryption and authentication, to protect data integrity and privacy. Its ability to dynamically scale from point-to-point communication to large mesh networks, combined with its energy efficiency and flexible architecture, makes BLE a cornerstone technology for modern IoT ecosystems. It continues to evolve with advancements like Bluetooth 5.0 and beyond, introducing features like extended range, higher throughput, and improved coexistence with other wireless technologies. These improvements further expand its potential in diverse applications, from smart homes to industrial enterprise systems.

3.3 The Rust Programming Language

Rust is a modern systems-programming language that has gained significant attention due to its focus on safety, performance, and concurrency [25]. First released in 2013 under the sponsorship of Mozilla, Rust was designed to address common challenges in low-level programming, such as memory safety and data race conditions, without compromising on efficiency. The language has since evolved into a popular choice for developers seeking robust solutions for a variety of applications, ranging from embedded systems to web services [69]. Rust emerged during a period when traditional systems programming languages, such as C and C++, faced criticism due to their difficulty in managing memory safety and concurrency [36]. While these languages provide developers with fine-grained control over system resources, they are also prone to common, memory-related bugs like buffer overflows and race conditions [54]. Recognizing these challenges, the Rust project set out to create a language that combined the performance characteristics of C with a strong emphasis on safety. The aforementioned principles are achieved through several unique features:

First, Rust introduces an ownership model that ensures memory safety without requiring a garbage collector. Every value in Rust has a single owner, and strict borrowing rules enforce compile-time guarantees against data races and use-after-free errors. In short, there can either be a single mutable reference or multiple immutable references to a value at any given time, but never both. Variables are automatically freed at the end of their lifetime, which happens either when a reference goes out of scope or, in the case of multiple readable references, when the last reference is dropped. Once code passes the compiler's strict checks, developers can be confident that their programs are free from memory-related bugs. In essence, this approach eliminates the need for manual memory management, making Rust code less error-prone and more secure.

Second, Rust aims to provide abstractions that incur no runtime overhead, thereby allowing developers to write high-level code without sacrificing performance. Therefore, Rust is particularly suitable for systems programming and performance-critical applications. The language's zero-cost abstractions are achieved through a combination of static dispatch, inlining, and other compiler optimizations. Rust's expressive type system enables developers to write code that is both safe and efficient, without compromising on readability or maintainability. At the cost of some additional complexity, Rust's type system allows developers to express complex relationships between data structures and enforce critical invariants at compile-time.

Third, by enforcing strict compile-time checks on data access, Rust prevents data races in concurrent programs. This ensures that multithreaded applications can leverage modern hardware capabilities safely and efficiently. References and data cannot be shared across thread boundaries by default, and the compiler enforces this rule during compilation. Instead, Rust provides explicit, thread-safe access through a combination of its ownership model and explicit markers for shareable data structures. These interfaces are required to be transitively used, meaning that any structure or type that marks itself thread-safe must ensure that all of its fields are also thread-safe. Additionally, Rust's standard library provides powerful, thread-safe abstractions for most use-cases, which allow developers to write concurrent code that is both safe and efficient.

Rust represents a significant advancement in programming language design, combining the performance and control of traditional systems languages with modern safety guarantees. Its innovative features, thriving community, and expanding ecosystem make it an important language in contemporary software development. Even though Rust is still a relatively young language, its adoption continues to grow [69], driven by its unique combination of safety, performance, and concurrency features. It is a powerful tool for developers seeking to build reliable, efficient, and secure software systems, and it's an increasing choice of large players in the industry, such as Amazon, Microsoft, and Google.

3.4 BRSKI-PRM

Bootstrapping Remote Secure Key Infrastructure with Pledge in Responder Mode (BRSKI-PRM) (BRSKI with Pledge in Responder Mode) is a BRSKI extension and a work in progress standard that offers bootstrapping of pledges without the requirement of a pre-enrollment network connection to the Domain-Registrar [26]. To provide enrollment functionality within initially networkless environments, BRSKI-PRM introduces a third-party service called the *Registrar-Agent* as a mediator between the Pledge and the customer Domain-Registrar. The Registrar-Agent is a network component that is responsible for establishing a secure connection with both the Pledge and the Domain-Registrar.

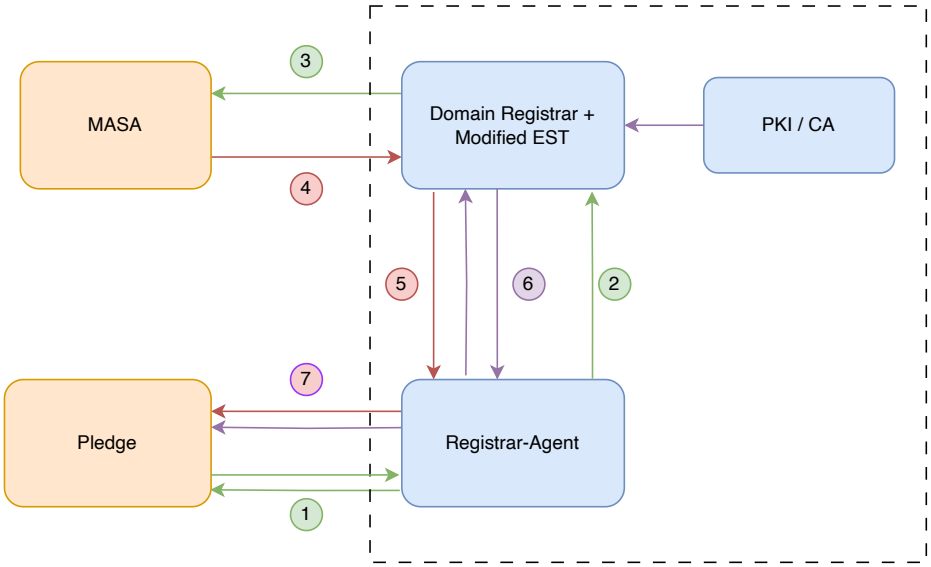

Figure 3.1: *The architecture behind BRSKI-PRM [26].*

BRSKI-PRM differs slightly from the original BRSKI standard. It offers multiple modes, this work, however, will primarily focus environments where the Registrar-Agent is *co-located* with the Registrar, where both entities communicate directly without the use of a proxy. Its architecture is depicted in Figure 3.1, which will now be explained in more detail. First, the Pledge is powered up and is now able to be located by the Registrar-Agent. Pledge's offer their serial number and other metadata to the Registrar-Agent, which is then used to locate the correct Pledge. Upon having found the Pledge, the Registrar-Agent will then initiate the enrollment process, either by manual action or by an unspecified automated process.

In BRSKI, the Pledge is the initiator of the enrollment process, where it sends a *Voucher-Request* [51, p. 3] [26, p. 7.1.2] to the Domain-Registrar. Instead, in BRSKI-PRM, the Registrar-Agent triggers the Pledge to create both a Voucher-Request, and an additional *Enrollment-Request* (1) [26, p. 7.2.2]. To create these artifacts, the Registrar-Agent informs

the Pledge of metadata by way of trigger objects. They are called *Voucher-Request-Trigger* [26, p. 7.1.1] and *Enrollment-Request-Trigger* [26, p. 7.2.1] respectively.

The first includes the Domain-Registrar's identifying X.509 End-Entity certificate, which is supplied to the Registrar-Agent by way of configuration. This certificate proves that the Registrar-Agent is authorized to act on behalf of the Domain-Registrar. However, at this point in time, the Pledge cannot verify the Registrar-Agent's identity. Later in the process, it will receive the domain's trust chain, which includes the CA certificate that signed both the Registrar's EE certificate and the Registrar-Agent's EE certificate. The Voucher-Request-Trigger also includes a signed object carrying either a nonce or a creation-date as well as the identified pledge's serial number. This signed object is embedded into the Voucher-Request created by the pledge and serves as a protection against replay attacks. The Pledge will now also create an Enrollment-Request artifact, which is a signed payload that includes a Certificate Signing Request (CSR).

Like in BRSKI, the Voucher-Request is sent to the Domain-Registrar by way of the Registrar-Agent proxy via a TLS connection between the Registrar and the Registrar-Agent (2). This TLS connection handshake is secured by the Registrar-Agent's End-Entity certificate, which *must* originate from the same trust anchor as the Registrar's certificate. This step is crucial, as both the Registrar-Agent and the Registrar must be able to verify each other's identities. Additionally, the Registrar wraps the Voucher-Request in a Registrar Voucher Request (RVR), a signed artifact that includes additional metadata. Here, BRSKI-PRM introduces the notion of a *Pinned-Domain-Cert*, which is a certificate chosen by the Registrar to be included in the RVR and which will also be included in the final Voucher artifact. This Pinned-Domain-Cert is a trust anchor certificate, and is at its simplest form, the Domain-Registrar's PKI CA certificate, which was used to sign the Registrar's EE certificate. As in BRSKI, the Pledge's Voucher-Request object contains the Pledge's certificate, where an API URL is included, which points to its vendor's services. The Registrar sends the RVR to the now located MASA, which then creates a Voucher artifact (3). The manufacturer signs this Voucher and responds (4), which the Registrar will both log and inspect for possible tampering. Before sending a copy to the Registrar-Agent (5), who will hold onto it until a later stage, the Domain-Registrar re-signs the Voucher object in-flight with the same EE certificate that was originally provided to the Registrar-Agent by way of configuration, and which is included in the trigger Voucher-Request artifact (see (1)).

In BRSKI, the protocol flow ends with the Pledge receiving the issued voucher. Control would be handed to EST for the process of enrolling a domain CA issued certificate onto the Pledge. Instead, BRSKI-PRM includes functionality of EST where needed and extends it with additional processes. The Enrollment-Request, containing the CSR sent by the Pledge to the Registrar-Agent at an earlier step, is now sent to the Domain-Registrar (6). Using all data available to the Registrar at this time, it can now verify the Pledge's identity. It then queries its domain Certificate Authority (CA) for a Pledge identity EE certificate, which it will then send back to the Registrar-Agent. This pledge certificate is signed by the Domain-Registrar's CA, and is used to authenticate the Pledge for the network connection

in the customer domain's enterprise network. In parallel to this process, the Registrar-Agent also send a request to the Domain-Registrar for a CA certificate bundle, which is essentially the trust chain needed to verify both the Registrar-Agent's EE certificate and the Registrar's EE certificate.

First, after having acquired Voucher artifact, the issued domain device certificate and the CA certificate bundle, the Registrar-Agent can now send this data back to the Pledge (7). The Pledge can confirm the Voucher by first verifying the MASA's signature on the Voucher object by using factory provided trust anchors. As mentioned previously, the Voucher object contains the pinned domain trust anchor. Using this, the Pledge can validate the Registrar-Agent's EE certificate, as well as the Registrar's added in-flight signature of the Voucher object.

Second, the Registrar-Agent sends the CA certificate bundle domain trust chain to the Pledge. Unlike the pinned-domain cert, the CA certificate bundle must also be the root of trust for the device certificate intended for the Pledge at the final step. This CA-certificate bundle is also transferred in the form of a payload signed by the Registrar with its own EE certificate. The Pledge can now validate the signature on this bundle, as in the step before, by using the pinned domain trust anchor from the Voucher object. Once verified, the Pledge installs these CA certificates into its trust store.

Finally, the Registrar-Agent sends the issued device domain certificate to the Pledge. By using the CA domain certificate bundle acquired in the previous step, the Pledge can now verify its own domain certificate. Once verified, the Pledge can install the domain certificate into its trust store, which it can finally use to establish a mutually authenticated TLS connection with whatever service it needs to connect to.

As can be seen from the above description, BRSKI-PRM is a more complex protocol than BRSKI. It also potentially turns the procedure into a one-touch process in which manual interaction is required, compared to the zero-touch process defined in BRSKI. Furthermore, BRSKI-PRM introduces a new security model, by making an encryption connection between the Pledge and the Registrar-Agent optional. Instead, (most) payloads are signed by all parties involved before being exchanged, thereby ensuring object security and foregoing transport security. While a TLS or mTLS connection is still recommended, this enables BRSKI-PRM can thus be used in heavily restricted environments where a TLS connection is not possible. Because the Pledge acts as a server in BRSKI-PRM, additional functionality is added where a client can request status and security information from the device before, during and after enrollment. For example, the Registrar-Agent can query the Pledge on whether the Voucher has been successfully installed. The Pledge can then respond with different response codes depending on its state. This allows more granular logging and responses to different error state, making BRSKI-PRM potentially more reliable than its predecessor. One more advantage of BRSKI-PRM is that since the Registrar-Agent acts as a full proxy between the Registrar and the Pledge, the Registrar can now operate agnostic of the Pledge's communication protocol.

3.5 cBRSKI

cBRSKI (Constrained Bootstrapping Remote Secure Key Infrastructure) is a BRSKI extension which aims to provide a secure bootstrapping process for constrained devices [55]. cBRSKI grew out of the need for an enrollment protocol with a focus on embedded devices with limited power throughput, computational capabilities, and flash storage. As in BRSKI, a Voucher artifact is exchanged between a device and a domain owner's authoritative service to facilitate mutual authentication. cBRSKI does not use the Hypertext Transfer Protocol (HTTP) transport protocol defined in BRSKI and BRSKI-PRM. Instead, it makes use of Constrained Application Protocol (CoAP), thereby also exchanging the HTTP-based EST protocol for *EST-coaps* [5]. CoAP is a lightweight protocol designed for constrained devices and networks [61], which works similar to HTTP.

Due to its focus on low-powered embedded devices, cBRSKI does not rely on JavaScript Object Notation (JSON) as its serialization format. Instead, it employs the use of Concise Binary Object Representation (CBOR) [3], a binary data format with the goal of providing an interface that results in minimal code and message size. CBOR is similar to JSON, with the added benefit that map values can be binary-encoded without needing to be Base64 encoded. In CBOR, map keys are encoded by providing a key map to the sender and receiver. Each key is assigned a minimal identifier, which can be encoded by the client and reproduced by the receiving party. The values are encoded as binary using an encoding scheme specified in the CBOR standard. cBRSKI uses this format for all exchanged data specified in the BRSKI protocol, as well as for the Voucher artifact. To support signing this Voucher object, as well as other BRSKI artifacts, cBRSKI employs a signing standard which supports its goals of minimal data exchange sizes. For this, its authors use CBOR Object Signing and Encryption (COSE) [60] [59], a signed message format similar to the JSON signing scheme JavaScript Object Signing and Encryption (JOSE). The data are first encoded according to the specifications and then packed into a COSE message. This message features protected and unprotected headers, where additional relevant metadata can be saved. Signatures protect the integrity of the included payload and the protected header parameter. The *payload* field includes the data to be exchanged between the server and client.

To reduce the transfer size, cBRSKI employs a constrained version of the Voucher [73]. For example, instead of including the Registrar's certificate to prove the proximity of the Registrar-Agent to the Pledge, a constrained Voucher only includes its public key. At a later point in time, when communication in BRSKI calls for exchange of information, which would include entire trust chains that has been sent previously, only the hash of each trust chain certificate public key is sent to minimize packet size. cBRSKI also uses X.509 certificates, as defined in BRSKI, encoded in a special format that fits the COSE specification [44].

4 Requirements

The aim of this chapter is to identify the requirements that are imposed by searching for a maximally secure enrollment scheme for low-powered IoT devices. First, a scenario in which the proposed scheme is applied is introduced. From this, additional desired properties of the proposed scheme are derived, as well as definite requirements that the scheme must fulfill. They are the focus of the first research question, **RQ1**.

The scenario considered is an enterprise environment in which numerous low-powered IoT devices are deployed. These devices are installed at various locations, ranging from office buildings to industrial plants. The devices are intended to be connected to an enterprise network through which they can communicate with other devices and services. In addition, the enterprise environment in the proposed scenario may exist in a public domain. Thus, devices may be physically exposed to potential attackers who may attempt to compromise the devices or the network. These devices may be mission critical, meaning that they exist in a high security environment, where data loss or device compromise could have severe consequences. Due to the limited scope of this study, only the enrollment process as the potential entry point for attackers, was considered. Therefore, this study focused on securing the initial device enrollment process. This study examines the enrollment process for constrained devices under the umbrella term of *IoT devices*. They exist in constrained environments, are low-power, and have limited computational resources. In this scenario, it is also a security requirement that no initial network access is granted before the enrollment process succeeds. Finally, the enrollment process should be as user-friendly as possible, requiring minimal manual intervention by service personnel. Service personnel should only require minimal, nontechnical training and as little as possible security clearance to enroll devices.

In this scenario, several desired properties of an ideal enrollment scheme are derived. First, in the context of academic research, only open, accessible research and software should be considered. To this end, this study will focus on openly accessible, third-party solutions as the basis of this work, and we cannot rely on proprietary solutions for fully transparent research. Second, to build on security-proven foundations, the proposed scheme should be based on established industry standards. Third, the proposed scheme should be easily extensible for further research and development. This will allow future work to build upon our results, in turn giving us the opportunity to give back to the security research community within the IoT domain.

With these properties and the above scenario in mind, the following requirements are identified for the enrollment scheme: First, because the focus is specifically on enrolling constrained devices in enterprise environments, and the compatibility with the operational limitations of low-powered constrained devices must be ensured. With only a few kilobytes of Random Access Memory (RAM) and Read-Only Memory (ROM), these devices have limited computational power and memory. Therefore, the bootstrapping

scheme must allow the use of lightweight and computationally efficient cryptographic algorithms. Additionally, data exchanged during the bootstrapping process should be kept to a minimum to reduce the computational load on the device. The scheme must be scalable to accommodate varying numbers of devices without compromising performance. Second, since IoT devices may be used in large numbers in enterprise environments, the proposed scheme must be able to accommodate varying large-scale device deployments. There should be little to no compromise on performance while also avoiding network congestion side effects when large batches of devices are enrolled simultaneously. Effortless capability should enable the proposed scheme to be employed in all types of enterprise environments, regardless of the number of devices enrolled. Employing passive devices that wait for an initial impulse to start the bootstrapping process reduces network load and is therefore a desirable feature.

Third, since the stated scenario requires the ability to employ service personnel with minimal training, the scheme should require minimal outside intervention. Complexity should be mostly hidden from the user, who only needs to verify the accuracy of the installation location. Essentially, the only interaction required is to initiate the enrollment process by use of an enrollment tool, either in physical form or as a domain-integrated software application. This approach simplifies the user experience for untrained service personnel and enhances efficiency. By limiting the necessary interactions, the proposed scheme can be made more user-friendly and less error-prone. The proposed approach also has the advantage of reducing attack surfaces and potential entry points for threat actors.

Fourth, in order to prevent network access to unauthorized devices in potential mission-critical environments, it is required that devices are entirely offline before finishing enrollment. When entirely isolating devices before and during the enrollment, one can guarantee even better security than relying on managed network solutions to prevent unauthenticated devices from accessing the network. If misconfigurations are present, for example, in a domain network's firewall, a disconnected-by-default device will not be able to connect to the network until it has been vetted. This is particularly important in environments where sensitive data is stored or processed. Therefore, focus should be placed on an enrollment scheme in which initial network connections are not a definite requirement and which offer alternatives to network-based enrollment.

Fifth, to build upon openly accessible solutions, the requirement is imposed that the scheme should be based on open-source solutions and software. It should not rely on a specific proprietary hardware or software solution. Better security through transparency and community engagement is hoped to be gained by utilizing open-source solutions. Open-source solutions are also more likely to be supported by a larger community, which can help with troubleshooting and development. This also allows for more innovation and flexibility in the development process. The use of open-source solutions also makes it easier to integrate the proposed scheme with existing infrastructure. Furthermore, by developing a rather complex system in the open, it is hoped that future research will build upon and improve the security of the work. Reliance on established enterprise standards provides us with the opportunity to work closely with protocol authors and industry

experts. This enables us to build upon a security-proven foundation and to benefit from the experiences of others. In turn, we hope to contribute to and further improve existing standards.

Sixth, since IoT devices in enterprise environments may access mission critical infrastructures or sensitive data, the scheme must include robust security measures. As part of the enrollment process, the selected scheme should include a method to securely distribute configuration parameters and network-specific key materials to the device. Some vendors may offer to fit devices with a single pre-shared network access key identical for all devices. This is a security risk because a single key can be extracted from a single device and used to access the customer's target network. Instead, the proposed scheme should incorporate installation attestation keys unique to each device during the manufacturing process. In this way, the device can be identified by the customer, and the network administrator can be sure that the device is genuine. The vendor's contribution to the identity-proofing process is of utmost importance. Therefore, the requirement is described that vendors provide openly accessible interfaces that customers can access to verify the identity of a device's key.

Chapter 3 and Chapter 2 discuss several secure enrollment schemes. We investigated ACME (see Section 2.3) as a candidate, but ultimately deemed it unsuitable for the scenario due to the need for a pre-enrollment network connection. The entire exchange between a proxy component connected to the network and the networkless device that is to be enrolled would also need to be engineered. EST (see Section 2.4) is also not suitable for the stated scenario because, like ACME, it requires devices to have already joined a network before enrollment can begin. Furthermore, EST requires either manual action or an unspecified automated process to confirm device identity before enrollment. We then looked into BRSKI (see Section 2.5), which is a foundational protocol for secure device bootstrapping. While BRSKI provides a bootstrapping process for devices once they are connected to the customer's network, it has multiple disadvantages when considering the previously determined requirements. First of, like EST, BRSKI is not intended for use in unstable networks or with devices that lack a network connection. When connected to the local domain, pledges in the BRSKI standard immediately start to relentlessly query administrative endpoints to request enrollment. This may affect scaleability because large amounts of yet to be enrolled pledges may cause congestion effects in a constrained network. BRSKI-PRM (see Section 3.4) is the most promising so far. At the cost of turning the enrollment process into a one-touch process, BRSKI-PRM offers an enrollment process for initially passive devices. It also theoretically allows for enrolling devices without a pre-existing network connection, but this behavior is entirely unspecified and therefore up to the implementor. However, like its predecessors, BRSKI-PRM is defined with a Internet Protocol (IP)-based transport protocol and is not sufficient for this work's needs of a low-powered, Bluetooth-focused process of exchanging data. Its transport protocol relies on JSON, which is suitable for a web server context and allows for human-readable payloads, but is not suitable for low-powered devices. There are several additional properties of BRSKI-PRM that are not ideal for devices with low-powered hardware. This is why cBRSKI (see Section 3.5), an extension of BRSKI aimed at providing a secure bootstrapping process

for constrained devices, was looked into. While cBRSKI meets many of the imposed requirements concerning the use of low-powered hardware, it is similar to BRSKI in that it uses IP-based transport protocols, and it defines pledges as active clients, that spam administrative endpoints with Enrollment-Requests as soon as they are connected to the local domain. Within the context of the existing work that we have examined, we have not found a scheme that meets all stated requirements.

Thus, to answer **RQ2**, this study proposes a hybrid approach. It specifically focuses on the BRSKI-PRM and cBRSKI schemes, because a unison of both fulfill a large part of the stated requirements. Since the original BRSKI specification was designed with extensibility in mind, and since both BRSKI-PRM and cBRSKI are largely congruent, the resulting enrollment scheme should still be compatible with the BRSKI specification. Here, the basic functionality of BRSKI-PRM is maintained and extended it with features from cBRSKI where applicable. Therefore, this study utilizes the entities and interactions defined in the BRSKI-PRM specification. Our resulting scheme will exist within this document under the name **Constrained Bootstrapping Remote Secure Key Infrastructure with Pledge in Responder Mode (cBRSKI-PRM)**.

5 Architecture

As described in Chapter 4, BRSKI-PRM and cBRSKI were selected as suitable bases upon which the intended enrollment scheme was constructed. The following sections present the architecture of the implemented system. First, the physical components required for the implemented system are described. Next, the needed software components are then discussed. Finally, the exact changes to the BRSKI-PRM protocol are presented, some of which are drawn from the cBRSKI specification.

Several components are necessary for the implementation of the hybrid enrollment protocol. Because the hybrid scheme will be built on BRSKI-PRM, it makes use of the same components. The general architecture behind BRSKI-PRM is as depicted in Figure 5.1. The following section describes the these required components of the proposed system and details their specific requirements. It also showcases at a high level how the different components interact with each other.

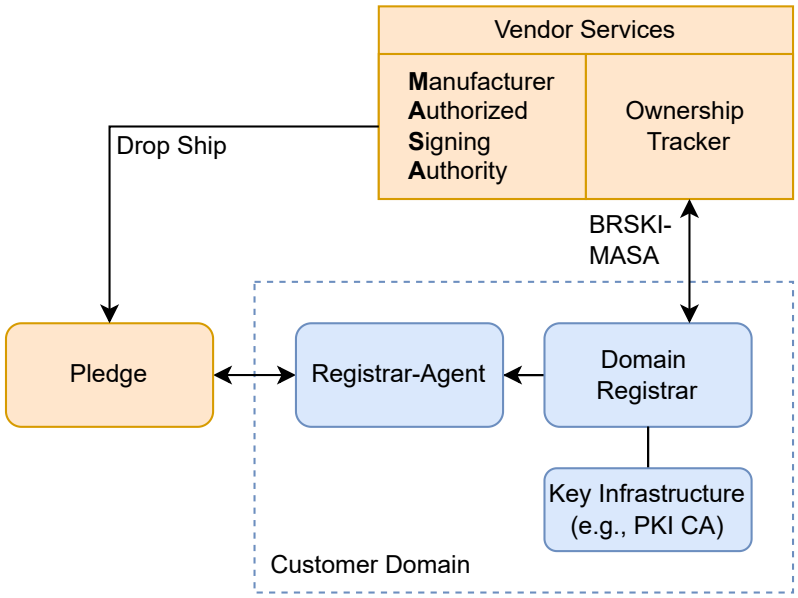

Figure 5.1: *BRSKI-PRM's architecture*

5.1 Pledge

As mentioned in Section 2.5, Pledges are devices that must be enrolled into a specific network. They are acquired in varying quantities from a vendor and manufactured at their facilities. In BRSKI based protocols, Pledges have a link-local network connection to a customer-operated service that allows communication with a registration authority during the enrollment process. In this study's scenario and according to the derived

requirements, this capability is removed, with an alternative communication protocol that does not rely on any network connection being opted for instead. This however comes with the requirement that a communication bridge between the Pledge and the entity tasked with enrolling the Pledge into the network be constructed. Ultimately, a Pledge must have the capability to store trust anchors and certificates, as defined in BRSKI-based specification documents. This information is used to establish a secure Transport Layer Security (TLS) connection to the domain network.

5.2 Registrar-Agent

The aforementioned communication bridge between the Pledge and the Domain-Registrar is BRSKI-PRM's Registrar-Agent. In this study's scenario, the Registrar-Agent exists as a tool on a physical device that is used by an operator to initiate the enrollment process manually. Specifically, an Android phone was selected. This phone acted as a bootstrapping tool and was to be equipped with the Registrar-Agent software. It is tasked with discovering the Pledge devices and providing a communication channel between the Pledge and the Domain-Registrar. Upon requesting the discovery process, the phone will scan for available nearby pledges and show them to the operator. After confirming the correct installation location with additional information, operators can make an informed decision on whether to initiate the enrollment process for a discovered Pledge. The complexity of the enrollment process will be hidden from the operator, who will receive either a success or failure message upon completion or cancelation of the enrollment process. The Registrar-Agent exists in what BRSKI-PRM calls a *co-located* [26] environment and therefore exists within the same network as the Domain-Registrar. The debugging information is sent to a central logging facility in the domain network.

5.3 Domain-Registrar

The customer Domain-Registrar is the device customer's central authority in the enrollment process. It is responsible for authenticating the Pledge and providing its trust root to the enrollment candidates. For the first step, the Domain-Registrar establishes a communication bridge between the Registrar-Agent and the device manufacturer. At some point in the protocol, the Pledge must create a certificate signing request, which will be sent to the Domain-Registrar by way of the Registrar-Agent. BRSKI-based protocols make no assumption of how the CSR attributes needed for creation of the CSR will be distributed to the Pledge. Thus, in the proposed protocol extension, a step is added where the Registrar-Agent requests domain-specific CSR attributes from the Domain-Registrar. This query will happen *after* a connection to the Pledge has been established and *before* the Registrar-Agent sends an Enrollment-Request-Trigger artifact (see Section 3.4). These attributes will be included in the artifacts listed above and sent to the Pledge. The Pledge can now include these attributes in the CSR it creates and sends to the Domain-Registrar. The Domain-Registrar will verify the CSR's signature and requests the issuance of a certificate

from the domain CA. In BRSKI and BRSKI-PRM, the Registrar has the task of verifying the issued Voucher artifact and provides additional, unspecified verification steps. In this scenario, the Domain-Registrar will be tasked with logging all available information, which is then sent to a central logging collection entity in the domain network.

5.4 MASA

As per BRSKI (see Section 2.5), the MASA is a service that provides two tasks in the enrollment process. First, it provides a public interface for the Domain-Registrar to communicate with the device manufacturer. Using this interface, the MASA can receive information from the customer's Domain-Registrar, including the data required for the enrollment process. This data specifies the device's identity certificate, a certificate of the customer's choosing for the trust chain pinning process, and further information like a creation date to identity when requests for a Voucher are first created. Second, the MASA provides internal logging services for ownership tracking and identification purposes to match pledges to their respective customers. Since the MASA is essentially simulated in this study's scenario, a service was developed, that provides the minimum required functionality for the BRSKI-PRM Voucher issue process. While real vendors may have more complex verification systems in place, a Voucher is issued if the provided Pledge identifier certificate is valid and has been created by the MASA CA certificate.

5.5 Proposed Protocol Extension

As described in Chapter 4, a fusion of BRSKI-PRM and cBRSKI fulfills the requirements of the proposed solution. Therefore, and to answer **RQ**2, the exact modifications required for BRSKI-PRM to work on hardware-constrained devices are described in the following section. The concrete implementation of these modifications is discussed in Chapter 6.

First, both BRSKI-PRM and cBRSKI utilize IP-based communication using HTTP(S) and CoAP, respectively. Due to the requirement of having no initial network connection, BRSKI-PRM cannot rely on an IP-based communication bridge to between the Pledge and the Registrar-Agent. Therefore, the proposed solution replaces the IP stack used in BRSKI-PRM with a different transport layer protocol. BLE is a suitable candidate for this task, as it is a low-power, short-range communication protocol that is supported by a variety of integrated devices. BLE also has the advantage of being supported by most modern smartphones, the latter of which can be used for the development of an enrollment tool, the Registrar-Agent.

HTTP and CoAP are both request-response protocols that allow clients to communicate with servers. Application-layer protocols for BLE are not request-response-based, but instead rely on the Generic Attribute Profile (GATT) to communicate (see Section 3.2). The first two protocols define the communication of intent from client to server using *routes* and *HTTP methods*. BLE, on the other hand, is a simple point-to-point protocol that

does not have these features. Since BLE instead operates using services and attributes to communicate, the BRSKI-PRM protocol has been adapted to use these concepts instead, which is detailed in Subsection 6.2.2.

Devices with constrained computational resources benefit from serialization formats and transfer sizes designed to minimize memory usage.

BRSKI and BRSKI-PRM rely on JSON for message encoding, which is a text-based format that is not well-suited for constrained devices. In addition, the size of BRSKI data packets is further increased due to the inclusion of multiple, sometimes large, certificate chains, which serve as proof of identity, as well as carrier information for the device's public key, which is used to sign transmitted messages. As specified in the BRSKI standard, binary data like these certificates must be encoded in Base64, a binary-to-text scheme that converts three bytes of binary data into four bytes of ASCII text, adding an overhead which can prove significant for devices with limited stack space. To address these limitations, a more suitable serialization format for constrained IoT devices can help reduce the payload size. Therefore, cBRSKI-PRM uses CBOR for message encoding, which is the binary counterpart of JSON, and which is also used in cBRSKI. CBOR eliminates the need for binary data conversion to and from Base64, because it directly transmits the data in its native format. While BRSKI and BRSKI-PRM rely on JSON Web Signature (JWS) tokens to encode signed JSON payloads, cBRSKI uses CBOR Object Signing and Encryption (COSE), a protocol designed to transport signed CBOR payloads. Implementation details of this technique are described in Subsection 6.2.1.

BRSKI-PRM signs data to secure its integrity. When a Voucher is issued by a vendor in BRSKI-PRM, this signed Voucher artifact is sent to the Domain-Registrar, which then signs the artifact again before sending the now twice-signed Voucher to the Registrar-Agent. At the time of writing, no library in the Rust ecosystem supports the behavior of signing an already signed token in-flight. After consultation with BRSKI-PRM's authors, we suggested a wrapping solution instead, where the Voucher artifact would be wrapped into a *Registrar-Wrapped-Voucher*, creating a signed outer shell around the masa-signed voucher. However, after consulting with the authors of BRSKI-PRM, they declined this approach. Therefore, this study includes a forked version of a pre-existing library for added support for this behavior. The implementation details of this process are explained in Subsection 6.2.1.

Even when using a more efficient serialization format, the size of the data packets exchanged during the enrollment process can still be too large to be processed natively by the BLE protocol. Therefore, this study includes a bespoke BLE packet fragmentation and reassembly algorithm, as well as a routing mechanism that allows for the transmission of large data packets over BLE. This work is detailed in Subsection 6.2.2. HTTP and COAP usually rely on the *Content-Type* header to negotiate the format of the message and on *ACCEPT* headers to negotiate the format of the response. Both are defined for every step in the BRSKI-PRM protocol. Bluetooth Low Energy does not have such a content negotiation mechanism. Therefore, cBRSKI-PRM includes two public facing endpoints that replicate content-type negotiation. Pledges advertise their capabilities concerning supported sig-

nature algorithms, and serialization formats to the Registrar-Agent an additional, newly defined first step of the protocol, before the enrollment process is initiated. The exact workings of this mechanism are explained in Subsection 6.4.2.

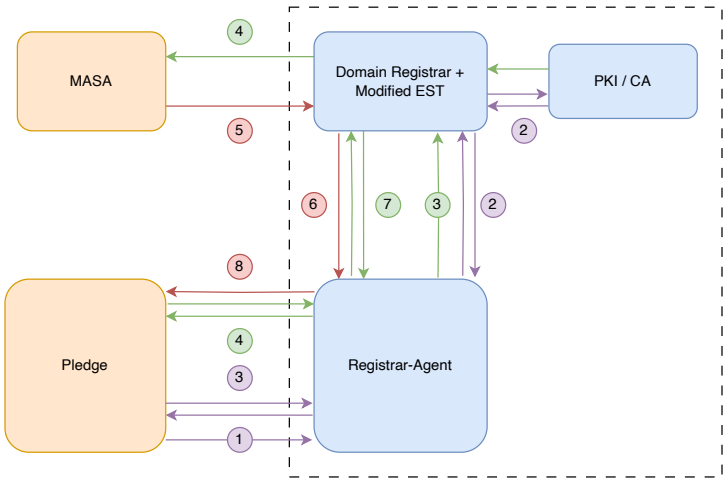

Figure 5.2: *The proposed architecture of cBRSKI-PRM*

There are more modifications required to adapt BRSKI-PRM for use with constrained devices. They are not limited to software changes, but also include changes to the protocol itself. For the cBRSKI-PRM protocol, this study proposes the following modification to BRSKI-PRM as seen in Figure 5.2. Lines in red and green indicate functionality that works as specified in BRSKI-PRM (see Section 3.4). Purple lines indicate new functionality added to the protocol. First, to accommodate for the aforementioned advertisement for Pledge capability to the Registrar-Agent, we have introduced a new initial step (1). This step is necessary to ensure that the Registrar-Agent is aware of the Pledge's capabilities before the enrollment process is initiated. Next, we modify the BRSKI-PRM defined steps for the exchange of the Voucher-Request trigger artifact and the enrollment trigger artifact. Now, and as described in Section 5.3, the Registrar-Agent is responsible for acquiring the domain defined certificate attributes needed for the Pledge's CSR. This is a bespoke diversion from the BRSKI-PRM standard, which does not define any strategy for when and how CSR attributes are communicated to the Pledge. Using this strategy, a new, modified Pledge enrollment artifact is created by the Pledge and sent to the Registrar as usual (3). Now, with the Pledge capabilities communicated and the CSR attributes sent to the Pledge, the enrollment scheme proceeds as defined in BRSKI-PRM (4-8).

6 Implementation

This chapter answers **RQ3**. It's explicit goal is to showcase how the cBRSKI-PRM proposal was implemented on low-powered hardware. The implementation of the prototype at first consisted of a simple implementation of BRSKI-PRM, which was then iteratively enhanced to include features of cBRSKI when applicable. Later, the implementation was extended to include the Android Registrar Agent, which acts as the Registrar-Agent and is used as a commissioning tool. Finally, the implementation was extended to include an ESP32-based prototype. The ESP32 is a common, high-level microcontroller used in IoT devices, which and was used to demonstrate the feasibility of the cBRSKI-PRM enrollment scheme in a constrained environment.

6.1 BRSKI-PRM Prototype

The first step in the implementation process was to develop a prototype of the BRSKI-PRM enrollment scheme. This served as the foundation for the subsequent integration of cBRSKI features. The prototype was developed using the Rust programming language, which is well-suited for embedded systems and provides a high level of control over system resources. Because Rust is a programming language with a focus on memory safety, it is ideal for the development of secure systems. In the initial prototype, all services are simulated on a single device, with each service running in its own thread. As per BRSKI-PRM, Pledge, Registrar-Agent, the Registrar and the MASA are all implemented as web services. Even though the final work will be implemented using a Pledge and a Registrar-Agent, which do not communicate via network, implementing both as web servers first allowed the verification of the correct communication between the components. For easier testing, the prototype includes a CLI tool that can be used to start the services and interact with them.

The prototype is openly accessible on GitHub [38]. It can be configured via command line arguments or a configuration file. An example of the required configuration parameters is shown in Listing A.1. They include the certificates and keys used by the service, as well as the URLs of other services that it needs to communicate with. Similar to BRSKI and BRSKI-PRM, this work includes the testing certificates used (see Appendix C). They include the CA certificates for the MASA and the Registrar, as well as the end-entity certificates for each party. For services that must to share a common trust anchor for verification purposes, a configuration entry is also added. Each of the services is reachable under the .well-known directory, which is a location for standardized endpoints in web-services. To identify BRSKI-related functionality, the services are hosted under */.well-known/brski* suffix.

The Manufacturer Authorized Signing Authority (MASA) is a critical component of the BRSKI-PRM architecture. As mentioned in Section 3.4, the MASA is responsible for

issuing vouchers to the pledges, which include security information about the target domain. This work chooses the simplest strategy for the MASA's decision process on whether to issue a voucher. If it can verify that the Pledge's certificate in the voucher-request has been signed by its own CA certificate, it will issue a voucher. Therefore, the MASA service only has to provide a single endpoint for the Pledge to request a voucher, as seen in Table 6.1.

Service Name	URL	Description
Request Voucher	`/.well-known/brski/requestvoucher`	Request a Voucher by POST-ing a Voucher Request Artifact

Table 6.1: *MASA Service Endpoints*

The Domain-Registrar acts as the customer's central authority in the BRSKI-PRM architecture. In BRSKI-PRM, the Registrar offers several services to Registrar-Agents, as is shown in Table 6.2. It combines functionality defined in EST for handling the requests of vouchers by pledges (1) and also when handling end entity certificate requests (2). After the Voucher is issued, a Pledge may request the domain trust anchor (3), which it installs into its own trust store for later verification and establishment of TLS connections. After the enrollment is complete, pledges must send telemetry data to the Domain-Registrar (4). This will help domain operators recognize errors and react accordingly.

Service Name	URL	Description
Request Voucher	`/.well-known/brski/requestvoucher`	Request a Voucher by POST-ing a Registrar Voucher Request Artifact (1)
Request Enroll	`/.well-known/brski/requestenroll`	Request a new Pledge EE certificate by POST-ing a Pledge Enroll Request Artifact (2)
Wrapped CA Certs	`/.well-known/brski/wrappedcacerts`	Request the Domain Trust Chain (3)
Enroll Status	`/.well-known/brski/enrollstatus`	Send Enroll Status Telemetry to Domain Administrators (4)

Table 6.2: *Registrar Service Endpoints*

Our implementation enhances the specification documents with additional verification steps. For example, BRSKI-based protocols only verify the correct format of a Pledge's Voucher-Request by checking if the HTTP content type information matches the payload. Instead, this implementation parses the payload into the expected data structure, thereby sacrificing speed for a correctness verification step at the earliest communication endpoint.

The Registrar-Agent is added to BRSKI in BRSKI-PRM and adds select functionalities to aid Pledges when joining a customer network. Essentially, the Registrar-Agent as a component is tasked with relaying information between the Pledge and the Registrar, while also adding security information to ensure the integrity of the proxied payloads. It is important to note that according to the specification document, the transport layer between Pledge and Registrar-Agent must be able to guarantee security without relying on TLS as a security protocol. In order to establish connection to the Pledge and to send the initial impulse to begin enrollment, operators need some way to interact with the Registrar-Agent. The BRSKI-PRM specification does not detail how this interaction should be implemented. Therefore, the Registrar-Agent in this implementation adds a route that allows for the initiation of the enrollment process via manual POST requests. It is depicted in Table 6.3.

Service Name	**URL**	**Description**
Init	`/.well-known/brski/init`	Begin the enrollment process

Table 6.3: *Registrar-Agent Service Endpoints*

According to BRSKI-PRM's specification, Pledges need to support the functionality defined in Table 6.4. First off, they need to answer initialization requests carrying trigger-metadata with a Pledge-Voucher-Request artifact (1). Next, Pledge's will need to answer a trigger artifact with a Pledge-Enroll-Request, which communicates the intent to enroll into the network (2). After the Voucher has been issued, it will be sent to the Pledge by the Registrar-Agent, for which the Pledge offers a route (3). Later, for verification purposes, the domain's trust chain will need to be sent to and installed by the Pledge (4). Once the domain device certificate has been issued, it is sent to the device as well (5). Finally, there is an endpoint for querying telemetry data from the Pledge (6).

Service Name	**URL**	**Description**
TPVR	./well-known/brski/tpvr	Trigger Pledge Voucher Request
TPER	./well-known/brski/tper	Trigger Pledge Enroll Request
SVR	./well-known/brski/svr	Supply Voucher to Pledge
SCAC	./well-known/brski/scac	Supply CA Certificates to Pledge
SER	./well-known/brski/ser	Supply Enroll-Response to Pledge
QPS	./well-known/bsrki/qps	Query Pledge Status

Table 6.4: *Simulated Pledge Service Endpoints*

6.2 cBRSKI-PRM

This chapter showcases the implementation of extensions needed for the cBRSKI-PRM protocol as specified in Section 5.5. First, it discusses the modifications to the BRSKI-PRM prototype needed for easily switching serialization and signature protocols. The differences, advantages and disadvantages between using JSON/JWS and CBOR/COSE as serialization and signature formats are shown. They showcase why CBOR/COSE are more efficient for constrained devices. Finally, the implementation details behind transferring cBRSKI-PRM related payloads via Bluetooth Low Energy are depicted.

6.2.1 Switching from JSON to CBOR

As mentioned in Section 2.5, BRSKI and most of its extensions employ JSON for the serialization of data. Depending on the step in the enrollment process, exchanged artifacts are composed of nested, signed JSON objects, which can grow to a considerable size. Some signed structures, like the Voucher-Request artifact (shown in Listing A.2), contain another, different signed objects. At its simplest form, the JSON-based Voucher-Request artifact is about kilobyte in size. This is due to the fact that JSON is a text-based format. Its size is further inflated by the inclusion of binary data, like certificates, which are encoded in the Base64 format due to BRSKI-PRM's data model [4] [26] [73]. Base64 is a binary-to-text encoding scheme that converts three bytes of binary data into four bytes of ASCII text [35], adding an approximate overhead of 33% (not including line-break sequences) to the original data. The JSON signature format, JOSE, further inflates the payload size, because its contents and metadata are also Base64 encoded (as depicted in Listing 6.1). The overhead is significant: When testing, the measured size of a JSON encoded and JOSE signed Voucher-Request artifact is around 3.3 kilobytes. Depending on the length of the certificate chains included in BRSKI messages, the payload's size may increase significantly. Therefore, it is worth considering alternative serialization formats for encoding data structures in BRSKI messages.

```
{
    "payload": BASE64URL(UTF8(data)),
    "signatures": [
        {
            "protected": BASE64URL(UTF8(HEADER)),
            "signature": BASE64URL(SIG)
        }
    ]
}
```

Listing 6.1: *Example of a JOSE signature*

Instead of JSON, cBRSKI chooses CBOR as an alternative serialization format. CBOR is a binary encoding format for data structures, which is designed to be more compact than JSON. It saves space by encoding data in a more efficient format, thereby reducing the overhead of encoded data by omitting long string keys. Since CBOR is a binary format, it can represent binary data directly without the need for Base64 encoding. Measuring the

BRSKI-PRM Flow	JSON/JOSE (in Bytes)	CBOR/COSE (in Bytes)
tPVR	1545	1205
PVR	3301	2464
tPER	37	33
PER	1750	1184
SVR	4107	2131
SVR Response	1118	607
SCAC	2839	2074
SCAC Response	0	1
SER	507	506
SER Response	1128	615

Table 6.5: *BRSKI-PRM Artifact Sizes with JSON/JOSE and CBOR/COSE*

size of payloads when encoded in CBOR shows a possible decrease of approximately 20-30% of the payload size compared to JSON. While this decrease may not seem significant at first, the difference between a couple of hundred kilobytes is substantial when stack space is extremely limited. In BRSKI, Pledge's may need to hold references to a Voucher object, the target domain's trust chain, the Pledge's own manufacturer-provided certificate and private key and finally the certificate Domain-Registrar's may issue, before verification of key material can commence and enrollment may complete. Expensive operations like data serialization or the computation of key material for verification purposes with an already heavy stack may exceed its RAM's capacity, triggering the Pledge's operating system to issue an out-of-memory error, which may crash the kernel. Therefore, optimizing encoding formats can make the difference between successful operations or a crashed kernel.

CBOR is not the only possible strategy for saving data size. In BRSKI-PRM, communication is secured at minimum by way of guaranteeing object integrity. This is achieved by signing messages with a private key, and verifying the signature with the corresponding public key. The public key is typically obtained from payload header, which includes a certificate chain. As per specification, the first certificate in this chain contains the public key of the signer. The certificate chain may include additional certificates depending on the implementation. Depending on the BRSKI extension, payloads are signed using different signing protocols. While BRSKI uses CMS and BRSKI-PRM uses JOSE, cBRSKI uses a CBOR compatible signing protocol named COSE (see Section 3.5). Swapping to CBOR and COSE saves considerable space in the payload, as binary data is no longer required to be encoded via Base64. In the scope of this study, a measurement of the payload size for both requests and responses using both JSON/JWS and CBOR/COSE was conducted. This data is depicted in Table 6.5, where one can see that the payload size is decreased considerably for most artifacts. For verification purposes, the voucher artifact and the domain trust chain need to exist simulatenously in system memory at the same time. Therefore, we can estimate an upper bound of at least 8 kilobytes of available stack space.

It is of utmost importance to ensure that the signing and verification of payloads is done correctly. It is inadvisable to implement signing and verification algorithms manually. For cryptographic primitives, the Rust ecosystem offers a variety of libraries, such as the `ring` library [64]. However, the maturity of the ecosystem is still a concern, as libraries are still under heavy active development and may not be suitable for production use. Since this implementation is based on BRSKI-PRM, libraries that support JOSE were first searched for and evaluated. JOSE-supporting backends for the popular *RustCrypto* ecosystem [49] were first investigated. However, upon closer inspection, *RustCrypto/jose-jws* [70] only supports parsing and serializing already signed JOSE objects. Searching *crates.io* [56], the official Rust package registry, yielded a couple of promising candidates. JWS allows for three different serialization formats for its tokens: *Flattened* [34, p. 7.2.2] and *Compact* [34, p. 7.1] for single signature tokens, and *General* [34, p. 7.2.1] for multi-signature tokens. BRSKI-PRM requires the use of the latter format, and the only available package that supports it is *josekit* [33]. While being the most promising candidate, there is a shortcoming of *josekit*. It supports multi-signature tokens, but only enables adding all signatures once, at the time of creation of the JOSE object. Since BRSKI-PRM defines an operation where a signature is added in-flight, called a *Registrar Countersigned Voucher* [26, Section 7.6.3], a custom fork of the library is used in this implementation. In BRSKI-PRM, the process of adding a signature in-flight is called *countersigning*.

In a multi-signature capable token, like JSON Web Signature (JWS) [34, Section 7.2.1] and COSE [58, Section 4.5] tokens, the second and subsequent signatures are commonly applied to the same components as the first signature, including the headers and payloads. Additional signatures, like the first, sign the payload, but they commonly do not re-sign any prior signatures. Therefore, in this case, each signature is independent of the others, and the order of signatures is not important for verification. For *josekit*'s fork, adding a signature to an existing token warrants first deserializing into a struct. An example object for JWS-dependent implementations is shown in Listing 6.2. This object represents JWS token as a Rust struct, into which JSON encoded JWS objects can be parsed.

```
struct Signature {
    protected: String,
    signature: String,
}

struct SignedJWS {
    payload: String,
    signatures: Vec<Signature>,
}
```

Listing 6.2: *Signed JWS object*

The process of countersigning an object is illustrated in Figure 6.1. It is explained using the step in the BRSKI-PRM protocol flow for the Registrar countersigning an issued Voucher sent by the MASA. First, the token is deserialized into the *SignedJWS* struct, which offers access the payload and signatures. Next, a dummy JWS token is created that contains the original token's payload and protected header metadata. The in-flight signing client signs

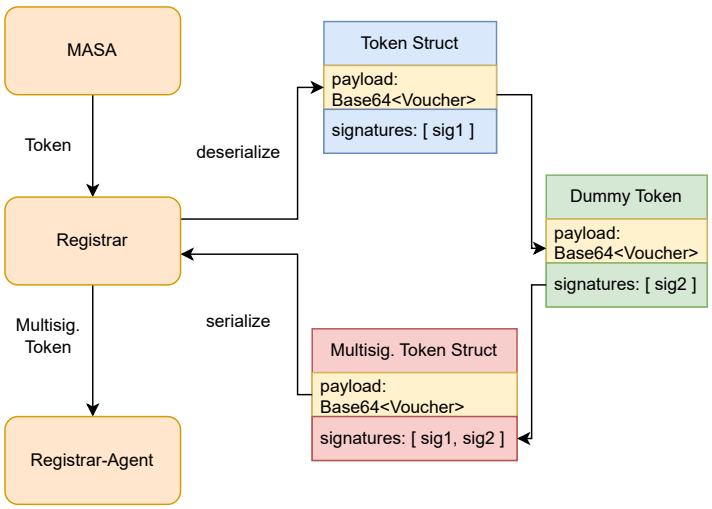

Figure 6.1: *Registrar adding its signature in-flight to a Voucher containing token*

this dummy JWS object, and the signature is extracted from the dummy object's *Signatures* array. With the original JWS still in deserialized form, the new computed signature can be added to the *Signatures*. The resulting token now contains the original payload, the original signature and the new signature. This workflow is the same for all tokens that support multiple signatures, and can be used to add signatures to any token that supports multiple signatures, such as JWS and COSE tokens.

Subsection 6.2.1 discusses the implementation of a different encoding and signature format. In order to enable the use of different signing protocols, this implementation abstracts the signing and verification of payloads into a separate module. This module can be easily replaced with a different implementation, allowing for the use of different signing protocols. To facilitate this, this implementation defines a multitude of interfaces that need to be implemented by any signing and verification module. All signing and verification modules share a set of supported algorithms, as well as a unified set of header fields that may be used. For example, a header may include a certificate chain used to verify the signature, a nonce to prevent replay attacks, as well as a subject key identifier to identify the signer. Each signing and verification module must implement the interfaces defined in the module, and must be able to sign and verify payloads using the algorithms defined in the module. A concrete implementation must implement a *sign* and *verify* function, as well as an *add_signature* function to add a signature to a payload in-flight. The interfaces are defined as per Listing A.3. Using this technique allows frictionless swapping of signature algorithms and encoding formats. This is especially useful in the context of BRSKI, where different extensions use different signing protocols. Finally, binary signing and verification schemes for human-readable alternatives can be swapped out on the fly, which is useful for debugging and testing purposes.

Function	Description	UUID
	Service	9b574847-f706-436c-bed7-fc01eb0965c1
TPVR	Read	9b574847-f706-436c-bed7-fc01eb0965c2
	Write	9b574847-f706-436c-bed7-fc01eb0965c3

Table 6.6: *UUIDs mapped to BRSKI-PRM REST routes*

6.2.2 Bluetooth Low Energy

As mentioned in Chapter 4, we aim to create an enrollment prototype with capabilities for network-less communication, using the Android Registrar-Agent as a proxy to communicate with the Domain-Registrar. While the first version of the prototype is based on using a web server for IP-based communication BLE communication was later built into the ESP32-based Pledge. Because the Android phone also has native BLE capabilities, Pledge and Registrar-Agent can use this communication bridge to enable the Pledge to be enrolled without the need for a prior network connection.

BRSKI and its extensions make use of REST over HTTP, using both HTTP routes and GET and POST requests to communicate intent when exchanging data. Using GATT's structured architecture allows mapping each REST route to a BLE service. Instead of using a URI path like in REST, BLE makes use of predefined, static UUIDs to differentiate between different services. The process itself is the same for all operations. Each route described in Table 6.4 is mapped to a threefold bundle of UUIDs, one for identifying the service and two for the reading and writing characteristics. Table 6.6 shows an example of the mapping of a single REST route to a BLE service with a read- and a write-characteristic. The full mapping table is shown in Table B.1. Each service's UUID is incremented by one for the read and write characteristics, respectively, to ensure a unique mapping for each route and also to ensure that UUIDs are kept in a structured order.

A bespoke *BLE Router* enables the structured registration of these Bluetooth routes, similar to a REST router in a web server. The BLE router is responsible for managing the mapping of routes to services and for handling the communication between the Pledge and the Registrar-Agent. Inside a registered service, the *write* and *read* characteristic share a buffer, which is first used to store the request data sent from a client, allowing the server to transform this data into a response when the *read* characteristic is queried. When reading the computed data via the read characteristic, the server serializes the computed data into a response. Most BLE devices appear handle advertisement of their services and characteristics via a technique called *Legacy Advertisement* [28] [20, Ext-Adv]. The advertising strategy transmits the advertisement data, which includes the BLE server's name, as well as a list of services and characteristics via a single packet. This packet is limited to a size of 31 bytes, which is not enough for the number of services made available as show in Table B.1. This warrants a switch to a relatively new advertising strategy, called *Extended Advertising* [48] [30, B-2.3.4]. It allows for larger advertising

frames of up to 1650 bytes [30, B-2.3.4.9], which is more than enough to advertise all services and characteristics.

This concrete implementation uses *NIMBLE* as the BLE stack, which is a lightweight, open-source BLE implementation for embedded devices. The Registrar-Agent acts as a GATT client, while the Pledge acts as a GATT server, which is also called a central device. Since BRSKI-PRM payloads can be larger than the maximum BLE packet size, a bespoke packet fragmentation mechanism is used to split the payload into smaller chunks, which are then sent over multiple packets. This mechanism is further described in Subsection 6.2.2. The communication process between the client and server is as seen in Figure 6.2.

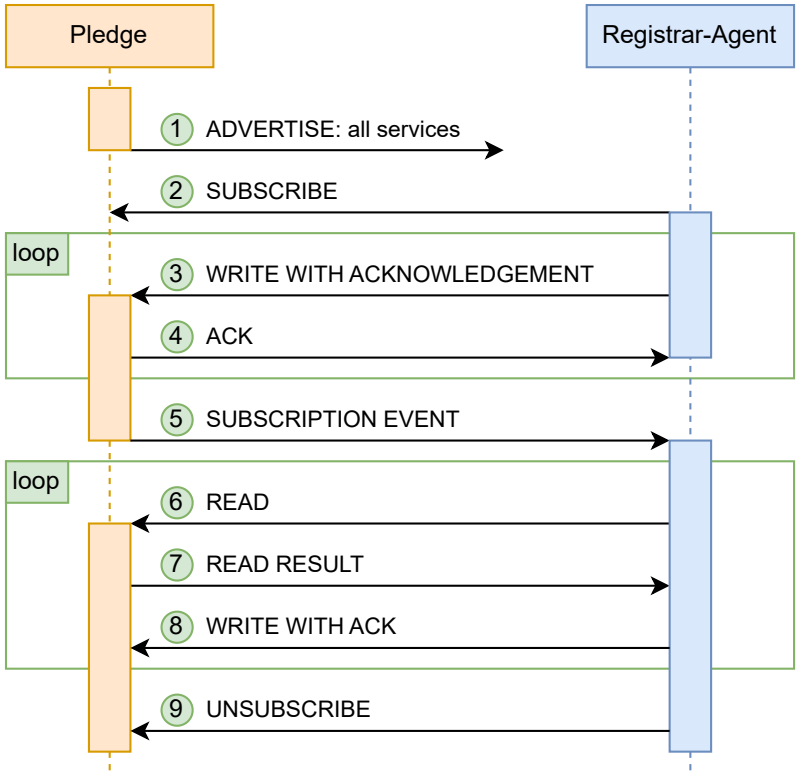

Figure 6.2: *BLE packet exchange using GATT*

The process unfolds as follows: When the Central is powered on, it begins by advertising all available services via their UUIDs (1). Each service is associated with a pair of characteristics for writing and reading data. Instead of initiating a traditional request, the client subscribes to the write-enabled characteristic (2), transmits data in chunks (3), and receives a confirmation for each write operation (4). Once all data is received, deserialized, and processed by the Central, a subscription event is triggered (5). Upon completing its computations, the Pledge serializes the response data and writes it to the designated read-enabled characteristic. The client then reads the response in chunks (6), obtains the complete result (7), and confirms the successful data retrieval (8). Finally, the subscription to the characteristic is terminated (9).

Bluetooth Low Energy has a maximum limit of 512 bytes per characteristic value [30, p. 3.2.9]. This limit is only sparsely mentioned in the BLE specification documents, and may differ between implementations. While the ESP32 BLE implementation appears to not follow the specification closely in this regard [20, ATTR_LEN], the native Android BLE implementation will refuse to handle characteristic values larger than the aforementioned limit [29]. This restriction is not to be confused with the negotiated GATT packet size, which is a separate value and is also known as a BLE MTU, often set to 23 bytes [30, A-5.1]. BLE already supports sending payloads which are larger than the negotiated MTU by splitting the payload into multiple packets, but only up to the aforementioned 512 byte limit. This is a problem when transferring BRSKI and BRSKI-PRM payloads, as the certificate chain included in the exchanged payloads often increases the data swapped between Pledge and Registrar-Agent past this ceiling.

To circumvent this limitation, a bespoke packet fragmentation mechanism was implemented, which lives on top of the BLE stack. This mechanism splits the payload into smaller fragments, which can then be sent in sequence by writing a characteristic's value for each chunk until the entire payload is sent and can be consumed by the server. A single frame is defined as per Listing A.6, which will now be explained. First, the code defines the maximum size of a native BLE packet (0). A data frame consists of a header (1) with meta-information, and a byte buffer of data (2). The size of the data buffer is calculated during compile-time by subtracting the size of the header from the maximum BLE payload size (3). This ensures that the frame is always the same size, regardless of the payload size. While some efficiency is lost by padding the data, this approach simplifies the implementation and makes it easier to handle the data on the receiving end. The *offset* field (4) is used to determine the position of the frame in the payload, while the *ident* field (5) is used to distinguish the origin characteristic of the packet. The *length* field (6) is used to determine the number of packets that will be sent within a transmission stream until the data is complete.

For each single *service route* defined by a read- and a write-characteristic, a writing and reading functionality is defined in the router. The writing functionality is responsible for fragmenting the data into frames and sending them to the Pledge. It is displayed in Listing A.4. It works as follows: First, the data is received from the writing characteristic (1) and deserialized into a data frame (2). Next, the target length and offset from the data frame are extracted (3, 4), after which the data frame is pushed into a buffer (5). If the data frame buffer reaches the intended length, the buffer is sorted by offset (6) and concatenated a single payload (7, 8). To save stack space, the buffer is then cleared (9), after which the payload is sent to the route handler (10). Once a lock is acquired on the computed buffer mutex, it is also cleared (11). Finally, the computed buffer is extended with the resulting data from the route handler. (12)

The reading functionality is responsible for sending the data back to the client. It is displayed in Listing A.5, and works like detailed in the following paragraph: First, a lock on the computed buffer pointer is acquired (1). Then, characteristic identifier is extracted from the characteristic's address (2). The computed buffer is split into frames (3), which

at this point is just raw binary data. Before enumerating through all frames, a buffer is created (4), which will contain a single serialized frame. For each frame, the frame is serialized into the buffer (5). Finally, the value of the reading characteristic is set to a pointer to the buffer (6).

In conclusion, this chapter outlines the critical extensions made to the cBRSKI-PRM protocol, focusing on improving efficiency and flexibility for constrained devices. By exploring the transition from JSON/JWS to CBOR/COSE for serialization and signature formats, we demonstrated significant space savings, highlighting the impact of reduced overhead on system performance, especially in resource-limited environments. The performance benefits of CBOR over JSON, along with its ability to directly encode binary data without the need for Base64, were quantified through size comparisons of different BRSKI-PRM artifacts. This optimization proves essential for devices like Pledges, where memory constraints can lead to system failures without such improvements.

6.3 Android Registrar-Agent

As mentioned in Section 5.2, the concrete prototype of the Registrar-Agent is implemented on an Android phone. It is used as a co-located enrollment tool as specified in BRSKI, functioning as a proxy component to facilitate communication between the Pledge and the Registrar.

The prototype is implemented using the open-source UI kit flutter, which uses the Dart programming language and allows for the development of cross-platform applications. The application is designed to be user-friendly and easy to use, with a simple interface that hides as much of the complexity of the BRSKI protocol as possible. The first screen of the application shows an initially empty list of discovered devices, which are pledges that are currently in the network. For development purposes, the application is preconfigured to only work with a single Pledge, and it will automatically ping the Pledge's IP address and port, waiting for an answer to finish the discovery process.

Since the Pledge's definition in the BRSKI-PRM specification was extended in this study to further include the exchange of pre-enrollment information, the Registrar-Agent first queries the Pledge for its capabilities. This information is then displayed to the user, along with the Pledge's serial number. Using this information, the user can then decide to start the enrollment process, which communicates to the android phone that the necessary enrollment artifacts are to be sent to the Pledge. The application will then guide the user through the process, showing the current state of the Pledge and the Registrar, as well as any errors that might occur during the process.

The implementation of the BRSKI-PRM protocol as a library in Section 6.1 allows the use of safe Rust code on the Android platform. By use of an FFI bridge, Rust code can be run natively on the Android device, where it interacts with Dart code using a Foreign Function Interface. This ensures that the communication between the Pledge and the Registrar is secure and reliable, while still providing a user-friendly interface for the user

to interact with. It also allows easier testing of the initial Rust-only prototype and keep the android-specific code separate from the core functionality of the Registrar-Agent.

There are a variety of ways to integrate Rust code into android applications. This study utilizes *Flutter Rust Bridge* [27], a library that allows for the seamless integration of Rust code into flutter applications. It provides a simple interface for calling Rust functions from Dart code, which enables calling Rust code in the background of the flutter application. The bridge library comes with a code generation tool, which can automatically generate the necessary bindings between Rust and Dart code. This way, one can easily call Rust functions from the Dart code, and vice versa, without having to write the bindings ourselves.

In order to communicate with pledges that support different transport protocols, Registrar-Agent implementations need to be able to consume the provided library and provide an implementation of a common connection interface. In Rust, interfaces are defined as traits, and the implementation of a trait can be provided for any structural type. The Pledge-Communication-Interface is defined in Listing A.7.

The Rust programming language does not use *inheritance* as found in common object-oriented programming languages like Java or C++. Instead, it relies on so-called *supertraits*, which defines any interface as a subset of another. In this case, the *PledgeCommunicator* trait is defined as a subset of the *Send* and *Sync* traits (1), which are used to define whether a type can be sent between threads and shared between threads, respectively. This allows, but does not limit implementing members of this trait to be used in a multithreaded context. Using the *PledgeCommunicator* trait requires implementing structures to define functions that send different types of messages to the Pledge. How that data is sent is left to the implementation, which can be over a network, serial port, or any other transport protocol. Each required function can be mapped to a function in Table 6.4. Each function takes a *Vec<u8>* as the message to be sent (3), and a *PledgeCtx* (4)(6) which contains information about the Pledge that the data is sent to. The functions are asynchronous, returning a *Result* type that can either be an error or a successful response (5). Like input data, output data is also formatted as a vector of bytes.

We provides implementations of the *PledgeCommunicator* trait, for example in the form of the *HTTPCommunicator* structure. For example, the *send_pvr_trigger* function in the *HTTPCommunicator* structure sends a POST request to the Pledge's URL with the trigger data in the body. An implementation for the HTTPCommunicator can be seen in Listing A.8.

The *HTTPCommunicator* structure (1) contains a *reqwest::Client* instance. A popular HTTP client library, *reqwest* [45], provides a simple interface for sending HTTP requests. The *send_pvr_trigger* function constructs the URL to send the trigger data to (2). It then sets the HTTP *ACCEPT* header, which denotes the MIME type the sender, in this case, the Registrar-Agent, is willing to accept for a response. The *CONTENT_TYPE* header is set to the Pledge's supported data interchange format (5). Both are set to values which the Registrar-Agent client has previously acquired from the Pledge before starting the

enrollment process. The payload is sent in the body of the POST request (6). The response is then checked for success (7), and if successful, the response data is converted to a vector of bytes (8) and returned.

Using the *PledgeCommunicator* trait allows easily switching between different implementations of the Pledge communication interface. It is also a shared interface between the Android and the Rust implementation, allowing for easy testing of the Rust implementation on a desktop machine before deploying it to the Android device. For Bluetooth Low Energy, an implementation of the *PledgeCommunicator* trait in the form of the *FFIBLECommunicator* structure is provided. It is a more complex structure compared to the *HTTPCommunicator*, as it needs to handle the communication between the Dart code and the Rust code and needs to make use of auto-generated, interfacing code. It is defined as follows in Listing A.9. Due to Rust's focus on correctness, correct typing for structures containing pointers to asynchronous, thread-safe functions shared among FFI bounds can get unwieldy. The function *ffi_send_pvr_trigger* (1) serves as an example to showcase the trade-off between security and inherent type-complexity which satisfies the compiler. *Arc* (2) is a Rust type for *Atomic Reference Counted* pointers. It is needed for multiple, read-only references to the same data. In this context, Arc ensures that the function can be safely shared across multiple threads. *Box* (3) is a smart pointer that allocates data on the heap. It provides a pointer type for explicit allocations of data where the size is not known at compile time. In Rust, *dyn Fn* (4) is a type for objects of some shared interface which represent a function or closure. *DartFnFuture<Vec<u8»* is the return type of the function. It represents a Dart-compatible future provided by *rust_flutter_bridge* that will eventually resolve to a Vec<u8>. The *FFIBLECommunicator* structure contains pointers to concrete implementations used by the methods defined in the *PledgeCommunicator* trait. Instead of the implementation living directly in the interface implementation, the corresponding FFI boundary partner is used to send the payload to the Pledge (1).

To create an instance of a FFI bridge, the *FFIBLECommunicator* structure is used as per Listing A.11. The Android Dart code makes use of the *getPledgeFFIBootstrapper* function (0) to create an instance of the *Bootstrapper* class, which contains the *FFIBLECommunicator* instance. A reference to a *BleRouter* class is passed to the function (1), which is used to route the trigger data to the correct BLE characteristic. It is an analogue implementation of the BleRouter structure in Rust (see Subsection 6.2.2). A reference to a *CharacteristicPackage* class is also passed to the function (2), which contains references to the UUIDs which identifiy the BLE characteristics used to communicate with the Pledge. The *FFIBLECommunicatorBuilder* is initialized (3), and the *setPvrFfi* function is called to set the function that will be called when the *send_pvr_trigger* function is called (4). The *build* function is then called to create the *FFIBLECommunicator* instance (5). The *Bootstrapper* class is then initialized with the *FFIBLECommunicator* instance (6) and config object and returned. The config object contains the certificates and private and public keys needed for the BRSKI-specific communication. In the prototype, these are included in the application assets included in the Flutter application. In a production environment, these would be securely stored on the device, inside the trust store.

When a service operator scans for Pledges in the general vicinity of the Registrar-Agent, the Flutter application will display a list of discovered Pledges. With the click on *connect*, the Android device will establish a Bluetooth Low Energy connection to the Pledge and start the enrollment process. A depiction of this is available in Figure D.1. When enrollment is complete, the Pledge will disappear from the selection list and the user will be notified of the successful enrollment. There is also a debug screen available to the operator, which shows the characteristics of the connected Pledge, as seen in Figure D.2.

6.4 ESP32 Prototype

As previously mentioned, in this study's scenario, the Pledge is an ESP32 with entirely custom firmware. In this section, the functionality of the software and the protocol extensions, as well as all the work that went into enhancing existing technology, will be described.

6.4.1 Research and Setup

There exists a vast variety of ESP32 devices on different architectures [9]. The prototype detailed in this study was developed on a ESP32-C3, which is a low-power, low-cost device with a RISC-V architecture. Although it is not as powerful as the ESP32-S3, it is more than sufficient for running cBRSKI-PRM. In theory, 16KB of flash storage and 8KB of memory are more than enough, but more powerful devices are easier to work with during development. After the prototype was finished, the maximum memory usage of the firmware was measured in order to determine the minimum requirements for cBRSKI-PRM. Since one of the requirements is the development of memory safe and reliable software, a Rust programming environment was set up for the ESP32-C3. Espressif delivers an official software development kit for all their devices, with additional Rust bindings to the C-native SDK.

To run the firmware on the ESP32-C3 during testing, a small laboratory environment was installed on a breadboard (see Figure 6.3). In order to prove the compatibility with more than one device, the firmware was tested on additional Espressif devices. The rightmost device is an ESP32-C3-DevKitM-1 [16] based on the ESP32-C3-Mini-1. The ESP32-C3-DevKitM-1 is a development board with 4 MB of available flash storage and a 400 KB SRAM module. This limited amount of memory was a definite challenge during initial testing. Therefore, a ESP32-C6-DevKitM-1 [17] based on the ESP32-C6-MINI-1(U) (at the right-hand side) was also acquired. Development on embedded chips can be difficult, and errors may be difficult to debug with just a serial console. For debugging purposes, a ESP-PROG debug board [19] was connected to the ESP32-C3-DevKitM-1. Debugging Rust firmware on the ESP32 platform is relatively novel. At the time of writing, debugging using the ESP-PROG board was error-prone. The ESP32-S3-DevKitC-1 [18] (on the left-hand side) includes a built-in debugger, which worked better. However, the board is based on the XTENSA architecture, which came with its own set of challenges. Cooperation

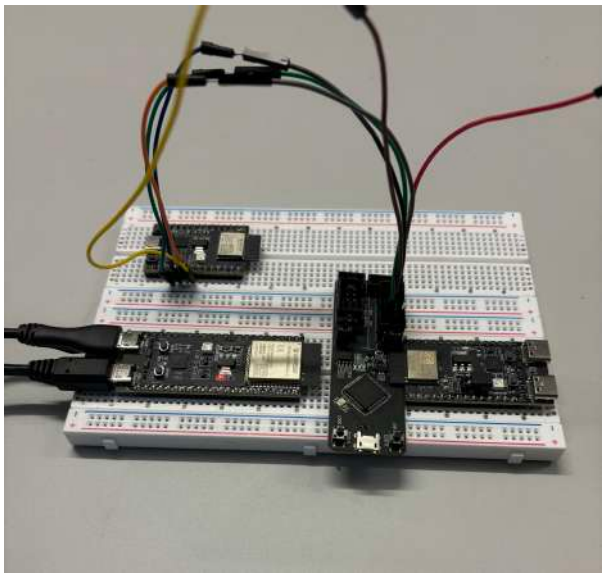

Figure 6.3: *A test environment featuring different Espressif devices*

with Espressif's Rust team was necessary to overcome these challenges. The Espressif team removed OpenSSL support in newer versions of the ESP-IDF, which was necessary for cryptographic operations. They instead now rely on WolfSSL or mbedTLS, which at the time of writing offered no Rust bindings. Cooperative work with *ring*'s author, Brian Smith, enabled the use of cryptographic primitives based on BoringSSL in the firmware. With a working development environment, the firmware was developed and tested on the ESP32-C3-DevKitM-1.

6.4.2 Pre-Enrollment Status Query

Pledges may be arbitrary powerful and may support different serialization formats to handle the data exchange. Ideally, cBRSKI-PRM's Pledges should be able to handle both JOSE and COSE encoded data. Pledges may also offer just a limited set of signature algorithms, or may not support all the algorithms that the Registrar-Agent supports. To handle these cases, the Pledge should be able to advertise its capabilities before the enrollment process starts. There is another reason than extensibility or flexibility to include this context providing functionality on top of the BRSKI-PRM standard. Since cBRSKI-PRM's prototype does not use IP-based communication, the context usually provided by HTTP headers is missing. As mentioned in Section 5.5, BRSKI-PRM uses *CONTENT-TYPE* and *ACCEPT* or accept headers to negotiate the format of the message and the response. BLE does not have such a content negotiation mechanism. The pledge must instead include such information in the capability advertisement step. For this, two additional endpoints are added to the pledge's service. The first endpoint returns a *DataInterchangeFormat* (DIF) enum, which is defined in Listing 6.3. This resource will return the encoding format that the Pledge is able to handle. At the time of writing, the

Pledge is able to handle both JOSE and COSE encoded data, as defined in Listing 6.3. The data interchange format is always returned as a UTF-8 encoded string.

```
enum DataInterchangeFormat {
    JSON,
    CBOR,
}
```

Listing 6.3: *DataInterchangeFormat Enum Values*

Once the Registrar-Agent has received the Pledge's data interchange format, it can then query the Pledge's capabilities by using the *PledgeInformation* resource. This resource will return the Pledge's capabilities in the structure defined in Listing A.12. Before the firmware is flashed onto the Pledge, manufacturers statically set the Pledge's capabilities. Neither the DIF nor the PI resource is encrypted nor signed by the Pledge, since it does not contain critical information. Upon receiving the Pledge's capabilities, the Registrar-Agent can then enter a code path that is able to handle the correct payload format and signature containing data types.

6.4.3 ESP32 firmware

The ESP32 firmware is based on the ESP32 SDK provided by Espressif. It is a composition of device APIs of various complexity, ranging from simple GPIO control to wireless configuration or interaction with the device's flash storage. Rust abstractions and translations for this SDK are available within the *esp-rs* project on GitHub [12]. There, community members and Espressif employees maintain a selection of Rust libraries for working with ESP32 devices. There are three different abstraction levels for working with the ESP32 in Rust, all of which are used in the prototype. First, *esp-idf-svc* [14] offers a higher level of abstraction over the native ESP32 functionality. While ESP-IDF-SVC is a good starting point, it offers neither Bluetooth Low Energy nor a task API. The library *esp-idf-sys* [15] offers a lower level of abstraction, which is necessary for working with the ESP32's Bluetooth stack [15]. It is used in this prototype for interaction with the ESP32's RTOS, the task management system, as well as manipulating the watchdog timer and the ESP32's NVS storage. Finally, *esp-hal* [13] offers a direct interface to the ESP32's hardware, which is necessary for working with the ESP32's GPIO pins and UART module.

ESP32 projects using the ESP32 SDK are configured using *Kconfig*, which is a menu-based configuration system also employed by the Linux kernel [20]. This prototype uses a configuration file akin to the one shown in Listing A.13. First, the maximum stack size for the main task and the NimBLE host task (1)(2) needs to be increased. This is necessary since the ESP32's default stack size is too small to run Rust code, which uses more stack space per default as compared to C programs. The next options enable the Bluetooth stack and the NimBLE host stack (3), which is a lightweight Bluetooth Low Energy implementation more suitable for embedded devices. They further enable the NimBLE host stack to persist its state in the ESP32's flash storage. As described in Subsection 6.2.2, legacy BLE advertising cannot handle the number of services that the Pledge will offer.

Therefore, the extended advertising feature is enabled (4). Because classic Bluetooth is not needed for this project, it is disabled (5). The system is also configured to halt on a kernel panic (6), which is useful for debugging purposes. Finally, the firmware is larger than intended by Espressif and does not fit on the default application partition. To fix this, the partition table needs to be adjusted.

```
1 # ESP-IDF Partition Table
2 # Name, Type, SubType, Offset, Size, Flags
3 nvs,data,nvs,0x9000,0x6000,, # (1)
4 phy_init,data,phy,0xf000,0x1000,, # (2)
5 factory,app,factory,0x10000,3M,, # (3)
```

Listing 6.4: *Partitions as per partitions.csv*

The partition table is defined in a CSV-like format. It defines the partitions that the ESP32's flash storage is divided into. First, the NVS partition is defined (1) for the NimBLE BLE stack. It is also used to save the Pledge's configuration, as well as the Pledge's private key and factory certificate. The NVS partition will also be used to store the domain certificate issued by the Registrar. Next, the *phy_init* partition (2) is used to store calibration data for the ESP32's radio, which also includes Wi-Fi and BLE. Finally, the *factory* partition (3) holds the firmware image.

The planned architecture warrants the usage of both the ESP32's Wi-Fi Mode and using Bluetooth Low Energy. Most ESP32 Devices only have a single 2.4 GHz antenna, which means that the Pledge can only be in one mode at a time. While the SDK offers functions to switch between modes, there is no native support for task switching which could allow yielding control to another task. To overcome this challenge, the ESP32's RTOS is used in tandem with Rust's *async* support, to handle the switching of modes asynchronously. Switching tasks manually is complex and difficult to use correctly, which is why this prototype uses both *tokio* [40] and *mio* [41]. They are asynchronous runtimes designed to handle task switching. Their exact setup is shown in Listing A.14. First, using *mio* warrants enabling *eventfd* on the ESP32 (1), which is a file descriptor that can be used to signal events between tasks. Asynchronous runtimes for Rust on the ESP32 use this file descriptor to signal that a task needs to be *polled*, which is the term used for checking if a task has new work. the asynchronous runtime is started in the firmware's main task. Since the ESP32 offers only a single compute core, the runtime is configured to run in single-core mode (2). Calling *enable_all* (3) enables asynchronous time-based events, as well as *IO* events, which are necessary for both using Wi-Fi and Bluetooth. The runtime is then built (4) and the main task is started (5).

The firmware's main task is shown in Listing A.15 and will be explained as follows. It first sets up a pointer to the device's peripherals (1), as well as a reference to the ESP32's native system event loop. Next, a timer service is initialized (3) and a reference to the ESP32's default NVS partition is created (4). The ESP-32's WiFi stack is initialized (5) by using the modem peripheral, a reference RTOS sysloop and the aforementioned pointer to the NVS partition. For asynchronous task switching, the WiFi peripheral is wrapped in an async context (6). Both WiFi and BLE are started concurrently (7) using *tokio*'s *join!*

macro. To stop the firmware from immediately exiting, the main task is put to sleep for a second (8) if no work is available. However, stopping the main task like this comes with a disadvantage. In order to prevent loose tasks from blocking the main thread indefinitely, the ESP32's RTOS will kill a task if no event is sent for a certain amount of time. Each task has a watchdog timer that needs to be reset periodically, as shown in Listing A.16.

7 Evaluation

This chapter evaluates the proposed scheme's implementation as discussed in Chapter 6. Here, the cBRSKI-PRM protocol and its prototype implementation, the details of which answered **RQ3**, are first compared with BRSKI-PRM. For each change made, the protocol extension and the prototype implementation will be analyzed to measure the impact on security, usability, and performance. It is of utmost importance to ensure that no new vulnerabilities are introduced by the changes made. The goal of this comparison is to ensure that the cBRSKI-PRM protocol is at least as secure as BRSKI-PRM.

Next, potential attack vectors will be examined. By identifying potential scenarios where the system could be exploited by an attacker, a better understanding of where the system is vulnerable to manipulation can be achieved.

7.1 Comparison of BRSKI-PRM and cBRSKI-PRM

Compared to BRSKI-PRM, cBRSKI-PRM is technically not reliant on a specific data encoding and signature scheme. At the time of writing, the prototype supports the use of both JSON and JOSE, as well as CBOR and COSE. This study's prototype has not implemented bespoke cryptographic schemes, choosing instead to rely on popular and well-established libraries. The chosen JSON and JOSE libraries rely on *OpenSSL* [22] [24] for cryptographic operations. As per the BRSKI-PRM specification, the signature algorithm used in this work is based on Elliptic Curve Cryptography (ECC) [47] with the NIST P-256 curve [39]. The NIST P256 curve is widely used in the industry and is considered secure, offering a good balance between secure keys and minimal key size. A dynamic linking approach is used to ensure that the system's default version of *OpenSSL* is used. This allows the implementation to offer a high degree of flexibility, as potential OpenSSL security updates can be applied without the need to recompile the implementation by way of system updates. For CBOR and COSE, the libraries *cborium*[46] and *coset*[6] were used. Both do not include any cryptographic operations, but instead focus on encoding and decoding data. Instead, they offer the library's consumer to provide their own cryptographic operations. For work with CBOR and COSE on embedded devices shipping without *OpenSSL*, the prototype uses *ring* [64] for cryptographic operations. Ring uses a Google SSL implementation called *BoringSSL*, which is relied upon by multiple well-known companies such as Cloudflare. This work relies on well-established, security-proven libraries for any cryptographic operations, which should instill confidence that the protocol's cryptographic operations are as secure as possible.

The cBRSKI-PRM protocol adds two new information endpoints to pledges, which registrars can access by way of proxy using the Registrar-Agent. They broadcast the Pledge's capabilities, including but not limited to the Pledge's supported data encoding protocol, signature schemes and supported key types. This potentially allows Registrars to imple-

ment a security strategy, with which they can decide whether to enroll pledges based on this information. If the Pledge's supported key type or signature scheme is considered not secure enough, the Registrar can choose to not enroll the Pledge. On the other hand, Pledges may support a variety of key types and signature schemes, which eliminates the risk of the Registrar's requested key type or signature scheme not being supported, which could lead to a failed enrollment, requiring the Registrar-Agent to re-trigger the enrollment process. This process was implemented to reach feature parity with BRSKI-PRM, in which such information is negotiated by way of HTTP header information the partaking parties exchange. Since cBRSKI-PRM does not add any security-dependent features, the security of the underlying protocol is not impacted by this change.

cBRSKI-PRM foregoes IP-based protocols in favor of a more flexible, transport-agnostic approach. The study's prototype relies on BLE as a transport protocol, but only for the data exchange between Pledge and Registrar-Agent. MASA and Registrar communication is done as in BRSKI-PRM, using HTTP with optional TLS support for secure communication. In BRSKI-PRM, Pledge and Registrar-Agent initially communicate using HTTP, but they may use HTTPS for secure communication later in the process. Instead of encryption, BRSKI and its extensions rely on transport security by guaranteeing payload integrity using signature schemes. This security guarantee is not lost when transferring data over BLE, even if the data is fragmented and reassembled during the communication process. When data is deserialized by the server or the client, it must be reassembled in its entirety, upon which the signature will be verified, just as in BRSKI. BLE is also not without optional security primitives. Depending on the implementation, cBRSKI-PRM can use BLE security features when first establishing a connection between the client and server. However, BRSKI-PRM's security guarantees are still upheld even if this step is skipped.

Finally, the prototype showcased in this work is based upon a Registrar using an Android phone, as well as a Pledge using a ESP32. Theoretically, the key material should always be stored in the device's trust store, which for android is a secure enclave, and for the ESP32 is a secure element which reserved space on the chip, utilizing flash or partition encryption for NVS storage where the certificate and key material is placed. To make the ESP32 even more secure in production environments, the JTAG/UART Boot eFuse should be permanently burned [11] [8], disabling the ability for JTAG debugging [10]. They should furthermore enable Secure Boot, which is a feature of the ESP32 that allows the device to boot securely, ensuring that only signed firmware is booted. For maximum security, vendors should make use of encryption firmware, which encrypts the firmware and data stored in the flash memory. By targeting embedded devices, the cBRSKI-PRM protocol introduces a new attack vector, which is the device itself. These devices are often physically accessible, and therefore can be tampered with. To mitigate this, the devices should be stored in secure locations, and tamper-evident seals should be used to detect unauthorized access. Finally, the ESP32 device is just an example used within the scope of this work. The cBRSKI-PRM protocol is not limited to this device and can be used with any device that supports BLE communication or potentially any other transport protocol. A chip's security features are dependent on its vendor, and therefore the security of the

device is dependent on the vendor's implementation of these features. This identifies the chip itself as the biggest introducer of potential security flaws.

7.2 Attack Vectors

The cBRSKI-PRM protocol introduces new attack vectors, some of which are inherent to the use of BLE as a transport protocol. Others are introduced by the specific embedded platform that may be used to implement the protocol.

The ESP32 specifically offers a variety of security features that can mitigate these risks [21]. For example, the *Secure Boot* feature ensures that only signed firmware is booted, and the *Flash Encryption* feature encrypts the firmware and data stored in the flash memory. It also offers a *Device Identity* feature, where a private key is stored in a secure enclave on the chip. This private key can be used to sign data via the chip's hardware cryptographic accelerator, but the key itself cannot be extracted from the chip. While this is not enough for a BRSKI implementation, since the chip's private key does not come with an accompanying vendor certificate, it is a good start. Additionally, the ESP32 supports optional *Memory Protection*, which protects the chip's firmware from manipulation by third-party peripheral code that may interact with the chip. There is also the option of disabling any outfacing debug interfaces, such as JTAG, to prevent unauthorized access to the chip. Finally, the ESP32 offers *Secure Storage*, a secure NVS partition that can be used to store sensitive data, such as certificates and keys, and which is used in the implementation of this work. While the ESP32 SDK offers a variety of security features, it is up to the developer to use of them. Ultimately, the security of the device is dependent on the customer's chip choice and the vendor's implementation of security features.

BLE is a wireless protocol, and as such, it is susceptible to interference and eavesdropping. An attacker could potentially intercept the communication between the Pledge and the Registrar-Agent, since BRSKI does not rely on encryption for the data exchange. However, the security of the protocol is not compromised, as BRSKI requests are signed by the Pledge, which the Registrar can verify with the help of the vendor. If the Pledge's certificate is properly stored in a secure enclave and is not accessible by a rogue entity, the object security of the swapped data is guaranteed. BRSKI objects also come with a replay protection mechanism, which ensures that the same request cannot be replayed by an attacker. Therefore, and contrary to environments where encrypted communication is required, the security of the protocol is not impacted when using unencrypted BLE communication. The ESP32 prototype is also not susceptible to downgrading connections to legacy BLE pairing methods, where encryption may be weaker.

However, there is a clear weak point that comes with Pledge's offering an open access port to third parties. The device is limited in power, and technically anyone can send data to any characteristic that the device offers. This is a potential attack vector, as an attacker could potentially send numerous crafted requests to the Pledge, which could lead to a denial of service attack. They could also send a single, large payload to the device

which could potentially overflow the device's stack, leading to a stack overflow attack and a crash of the device's firmware. However, the device is set to reset after a kernel panic, which would render the attack effective only once per request. There is a potential way to mitigate this attack vector. The prototype rejects any packets that are not in the format described in Subsection 6.2.2. It also stops deserializing the packet if the number of packets specified in the frame's header is exceeded. To avoid a buffer overflow crashing the kernel, one could add a check on each frame's deserialization that returns an error if the frame's size exceeds the available stack space.

8 Discussion & Future Work

This research contributes to the field of IoT-security by addressing the unique challenges posed by constrained devices during the initial enrollment phase. By building upon existing research, a hybrid approach that combines BRSKI-PRM and cBRSKI was developed. The cBRSKI-PRM protocol demonstrates enhanced compatibility with low-powered devices while maintaining the security guarantees of BRSKI-PRM. Finally, feasibility of the proposed scheme was shown successfully by implementing a prototype using an Android phone and a ESP32.

The proposed protocol introduces BLE as a novel transport approach for the BRSKI protocol family. This technique equips the protocol with the ability to enroll devices that are not capable of using IP-based protocols, or which are intended to be disconnected from a network during enrollment. The adoption of CBOR and COSE and introduced in cBRSKI allows glsBRSKI-PRM's architecture to support a wider range of devices. For example, by eliminating BRSKI-PRM's reliance on JSON and JOSE, the scheme can now enroll devices where payload data size would be a limiting factor. In turn, BRSKI-PRM's Registrar-Agent as a proxy component between Pledge and the Domain-Registrar further enhances the protocol for use in environments where the Pledge and the Domain Registrar do not exist on the same network. It also supports scenarios where there is a logical boundary between the Pledge and the Registrar as a potential domain critical component.

A working Android companion app, which acts as a Registrar-Agent, demonstrates that the protocol's comissioning tool can be used in a real-world scenario. The phone provides a user interface on a familiar device, which gives technicians and domain operators an easy tool to initiate the enrollment process for a Pledge. By needing manual intervention in the way of a single button press if a nearby Pledge is detected, the trigger mechanism for the enrollment process is user-friendly.

Using the Rust programming language leverages the language's safety features to ensure that the implementation is robust and memory secure. Public cryptographic libraries, which are widely used and have been tested in the field, were made use of to deliver maximum security for cryptographic operations. Valuable feedback, as well as software fixes were also provided to the Rust Espressif community. Furthermore, within the scope of this work, collaboration with various open-source projects and communities was achieved. Contributing to the open-source community is a key aspect of this work, as it helps to improve the security of the IoT ecosystem. For example, critical bugs in the ESP32's flash tool for Rust applications were pinpointed and fixed through a group effort. Contributions also helped the Ring project by improving their documentation, as well as working together with Ring's maintainers to compile the library for XTENSA and Risc-V architectures. The Rust ESP32 effort was helped in its entirety by implementing missing features, such as critical missing functionality in the device's NimBLE stack. Working closely with BRSKI-PRM's architects, Siemens, the first open-source up-to-spec

implementation of the protocol was developed. As of the time of writing, BRSKI-PRM was still an internet draft on its way to becoming a ratified RFC document, which has since finished. Our cooperating effort will be forever marked in this RFC, and this work's author's name has been included in the contributor section of the RFC document. The prototype developed in this work is also linked in the document.

Finally, the feasibility of the protocol and its prototype implementation was demonstrated not only in this work, but also in a six-page paper submitted to the 20th European Dependable Computing Conference (EDCC) conference. As of time of writing, this work has been accepted for publication in the conference's proceedings.

9 Conclusion

This thesis has addressed the critical challenges of securely enrollment constrained IoT devices into enterprise networks. By analyzing existing protocols and identifying their limitations, cBRSKI-PRM, a hybrid protocol that combines the strengths of BRSKI-PRM and cBRSKI, was proposed and implemented. This novel approach leverages Bluetooth Low Energy (BLE) as an out-of-band communication channel, ensuring compatibility with low-powered devices while maintaining robust security. The design, implementation, and evaluation of cBRSKI-PRM demonstrated the feasibility and effectiveness of the proposed solution. The protocol not only bridges significant gaps in existing standards but also ensures scalability, minimal user intervention, and compliance with security requirements. Furthermore, the use of CBOR-based serialization and object security significantly reduces the overhead, making it well-suited for resource-constrained environments. Extensive testing confirmed the robustness of cBRSKI-PRM against attack vectors, while also validating its usability in real-world enterprise scenarios. By addressing both technical and operational challenges, this thesis contributes to the advancement of secure IoT device management and provides a solid foundation for future research in this area.

Future work could explore further optimization of BLE communication efficiency, integration with other protocols, and broader testing in diverse deployment scenarios. Extending the protocol to support additional device classes and use cases could further enhance its applicability. By contributing to the field of secure IoT device management, this thesis lays the groundwork for advancing secure and scalable solutions tailored to the unique challenges of constrained devices. The findings reinforce the importance of innovative approaches to address the evolving IoT security demands.

List of Figures

2.1 The architecture behind RFC 8995 [51]. 8

3.1 The architecture behind BRSKI-PRM [26]. 13

5.1 BRSKI-PRM's architecture 21
5.2 The proposed architecture of cBRSKI-PRM 25

6.1 Registrar adding its signature in-flight to a Voucher containing token . . . 33
6.2 BLE packet exchange using GATT 35
6.3 A test environment featuring different Espressif devices 41

D.1 The Flutter Application showing the scan for devices screen 84
D.2 The Flutter Application showing a debug panel with device characteristics 85

List of Listings

6.1 Example of a JOSE signature 30
6.2 Signed JWS object 32
6.3 DataInterchangeFormat Enum Values 42
6.4 Partitions as per partitions.csv 43

A.1 Open-BRSKI Configuration File 63
A.2 Example of a Voucher-Request encoded in JSON 63
A.3 Signing and Verifying Interface 63
A.4 Packet Defragmentation in Write Characteristic 64
A.5 Packet Fragmentation in Read Characteristic 65
A.6 Data Frame for Fragmented Payloads 65
A.7 Pledge Communication Interface 65
A.8 HTTP Communication Implementation 66
A.9 BLE FFI Communicator Structure 67
A.10 BLE FFI Communication Implementation 68
A.11 Create FFI Bridge using Dart 68
A.12 PledgeInfo Structure 69
A.13 Configuration using sdkconfig.defaults 69
A.14 Setting up the async Runtime 69
A.15 ESP32 Main Task 70
A.16 Resetting the Watchdog Timer 70

C.1 MASA issued Pledge EE Certificate (IdevID) 73
C.2 Registrar-Agent EE certificate 74

C.3 Registrar Certificate Authority certificate 75
C.4 Registrar EE certificate 77
C.5 MASA CA certificate 78
C.6 MASA EE certificate.txt 80

List of Tables

6.1 MASA Service Endpoints 28
6.2 Registrar Service Endpoints 28
6.3 Registrar-Agent Service Endpoints 29
6.4 Simulated Pledge Service Endpoints 29
6.5 BRSKI-PRM Artifact Sizes with JSON/JOSE and CBOR/COSE 31
6.6 UUIDs mapped to BRSKI-PRM REST routes 34

B.1 UUIDs mapped to BRSKI-PRM REST routes 71

Bibliography

[1] H. Aldowah, S. Rehman, and I. Umar. "Security in Internet of Things: Issues, Challenges, and Solutions". In: July 2019. ISBN: 978-3-319-99006-4. DOI: 10.1007/978-3-319-99007-1_38.

[2] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. *Automatic Certificate Management Environment (ACME)*. RFC 8555. RFC Editor, Mar. 2019. DOI: 10.17487/RFC8555. URL: https://www.rfc-editor.org/info/rfc8555.

[3] C. Bormann and P. E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 8949. RFC Editor, Dec. 2020. DOI: 10.17487/RFC8949. URL: https://www.rfc-editor.org/info/rfc8949.

[4] C. Bormann and Z. Shelby. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)*. RFC 7959. RFC Editor, Aug. 2016. DOI: 10.17487/RFC7959. URL: https://www.rfc-editor.org/info/rfc7959.

[5] P. V. der Stok, P. Kampanakis, M. Richardson, and S. Raza. *EST-coaps: Enrollment over Secure Transport with the Secure Constrained Application Protocol*. RFC 9148. RFC Editor, Apr. 2022. DOI: 10.17487/RFC9148. URL: https://www.rfc-editor.org/info/rfc9148.

[6] D. Drysdale and P. Crowley. *Google/Coset: A Set of Rust Types for Supporting COSE*. Version 0.3.8. Google, July 24, 2024. URL: https://github.com/google/coset (visited on 09/20/2024).

[7] V. Dukhovni. *Opportunistic Security: Some Protection Most of the Time*. RFC 7435. RFC Editor, Dec. 2014. DOI: 10.17487/RFC7435. URL: https://www.rfc-editor.org/info/rfc7435.

[8] Espressif. *Burn Efuse - ESP32 - — Esptool.Py Latest Documentation*. June 18, 2024. URL: https://docs.espressif.com/projects/esptool/en/latest/esp32/espefuse/burn-efuse-cmd.html (visited on 01/05/2025).

[9] Espressif. *Chip Series Comparison - ESP32 - — ESP-IDF Programming Guide v5.0.7 Documentation*. May 7, 2022. URL: https://docs.espressif.com/projects/esp-idf/en/v5.0.7/esp32/hw-reference/chip-series-comparison.html (visited on 12/26/2024).

[10] Espressif. *Configure Other JTAG Interfaces - ESP32-S3 - — ESP-IDF Programming Guide v5.4 Documentation*. Sept. 18, 2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-guides/jtag-debugging/configure-other-jtag.html (visited on 01/05/2025).

[11] Espressif. *eFuse Manager - ESP32 - — ESP-IDF Programming Guide v5.4 Documentation*. Aug. 19, 2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/efuse.html (visited on 01/05/2025).

[12] Espressif. *Esp-Rs*. Feb. 9, 2023. URL: https://github.com/esp-rs (visited on 01/16/2025).

[13] Espressif. *Esp-Rs/Esp-Hal*. esp-rs, Jan. 16, 2025. URL: https://github.com/esp-rs/esp-hal (visited on 01/16/2025).

[14] Espressif. *Esp-Rs/Esp-Idf-Svc: Type-Safe Rust Wrappers for Various ESP-IDF Services (WiFi, Network, Httpd, Logging, Etc.)* Version 0.51.0. Jan. 16, 2025. URL: https://github.com/esp-rs/esp-idf-svc (visited on 01/16/2025).

[15] Espressif. *Esp-Rs/Esp-Idf-Sys*. esp-rs, Jan. 16, 2025. URL: https://github.com/esp-rs/esp-idf-sys (visited on 01/16/2025).

[16] Espressif. *ESP32-C3-DevKitM-1 - ESP32-C3 - — ESP-IDF Programming Guide v5.2 Documentation*. Sept. 5, 2023. URL: https://docs.espressif.com/projects/esp-idf/en/v5.2/esp32c3/hw-reference/esp32c3/user-guide-devkitm-1.html (visited on 12/26/2024).

[17] Espressif. *ESP32-C6-DevKitM-1 - ESP32-C6 - — Esp-Dev-Kits Latest Documentation*. Dec. 11, 2024. URL: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitm-1/user_guide.html (visited on 12/26/2024).

[18] Espressif. *ESP32-S3-DevKitC-1 v1.1 - ESP32-S3 - — Esp-Dev-Kits Latest Documentation*. Dec. 11, 2024. URL: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32s3/esp32-s3-devkitc-1/user_guide.html (visited on 12/26/2024).

[19] Espressif. *Introduction to the ESP-Prog Board - - — ESP-IoT-Solution Latest Documentation*. Sept. 11, 2024. URL: https://docs.espressif.com/projects/esp-iot-solution/en/latest/hw-reference/ESP-Prog_guide.html (visited on 12/26/2024).

[20] Espressif. *Project Configuration - ESP32-C3 - — ESP-IDF Programming Guide v5.3.2 Documentation*. Sept. 3, 2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/api-reference/kconfig.html?highlight=ble_ext_adv#config-bt-nimble-ext-adv (visited on 12/26/2024).

[21] Espressif. *Security - ESP32-C3 - — ESP-IDF Programming Guide v5.4 Documentation*. Jan. 4, 2025. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/security/security.html (visited on 01/05/2025).

[22] S. Fackler. *Sfackler/Rust-Openssl*. Jan. 2, 2025. URL: https://github.com/sfackler/rust-openssl (visited on 01/05/2025).

[23] E. F. Foundation. *Certbot/Certbot*. Certbot, Jan. 5, 2025. URL: https://github.com/certbot/certbot (visited on 01/05/2025).

[24] O. Foundation. *OpenSSL*. Version 3.4.0. Oct. 22, 2024. URL: https://openssl.org/ (visited on 01/05/2025).

[25] R. Foundation. *Rust Programming Language*. Jan. 12. URL: https://www.rust-lang.org/ (visited on 12/16/2024).

[26] S. Fries, T. Werner, E. Lear, and M. Richardson. *BRSKI with Pledge in Responder Mode (BRSKI-PRM)*. Internet-Draft draft-ietf-anima-brski-prm-15. Internet Engineering Task Force / Internet Engineering Task Force, Aug. 26, 2024. 113 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-anima-brski-prm/15/.

[27] fzyzcjy. *Fzyzcjy/Flutter_rust_bridge: Flutter/Dart <-> Rust Binding Generator, Feature-Rich, but Seamless and Simple*. URL: https://github.com/fzyzcjy/flutter_rust_bridge (visited on 01/15/2025).

[28] Google LLC. *Bluetooth Low Energy Advertising*. Android Open Source Project. Dec. 12, 2024. URL: https://source.android.com/docs/core/connect/bluetooth/ble_advertising (visited on 12/26/2024).

[29] Google LLC. *Gatt.Py - Android Code Search*. May 17, 2022. URL: https://cs.android.com/android/platform/superproject/main/+/main:external/python/bumble/bumble/gatt.py;l=63;drc=f06a35713f6db739e3c53738eda25598efcd47f4 (visited on 12/26/2024).

[30] B. S. I. Group. *Bluetooth Core Specification Version 6.0 Vol. 3*. Bluetooth Special Interest Group, Aug. 27, 2024. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification/.

[31] M. El-Hajj, A. Fadlallah, M. Chamoun, and A. Serhrouchni. "Ethereum for Secure Authentication of IoT Using Pre-Shared Keys (Psks)". In: *2019 International Conference on Wireless Networks and Mobile Communications (WINCOM)*. 2019. DOI: 10.1109/WINCOM47513.2019.8942487.

[32] R. Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652. RFC Editor, Sept. 2009. DOI: 10.17487/RFC5652. URL: https://www.rfc-editor.org/info/rfc5652.

[33] H. Izuno. *Hidekatsu-Izuno/Josekit-Rs*. Dec. 20, 2024. URL: https://github.com/hidekatsu-izuno/josekit-rs (visited on 12/26/2024).

[34] M. B. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. RFC Editor, May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/info/rfc7515.

[35] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. RFC Editor, Oct. 2006. DOI: 10.17487/RFC4648. URL: https://www.rfc-editor.org/info/rfc4648.

[36] S. Klabnik. "The History of Rust". In: *Applicative 2016 on - Applicative 2016*. Applicative 2016. New York, NY, USA: ACM Press, 2016. ISBN: 978-1-4503-4464-7. DOI: 10.1145/2959689.2960081. URL: http://dl.acm.org/citation.cfm?doid=2959689.2960081 (visited on 12/16/2024).

[37] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser. "A Practical Attestation Protocol for Autonomous Embedded Systems". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019 IEEE European Symposium on Security and Privacy (EuroS&P). June 2019. DOI: 10.1109/EuroSP.2019.00028. URL: https://ieeexplore.ieee.org/abstract/document/8806746 (visited on 12/07/2023).

[38] J. Krieger. *Hm-Seclab/Open-Brski*. Version 0.0.1. SecLab Munich, Oct. 24, 2024. URL: https://github.com/hm-seclab/open-brski (visited on 12/17/2024).

[39] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. RFC 7748. RFC Editor, Jan. 2016. DOI: 10.17487/RFC7748. URL: https://www.rfc-editor.org/info/rfc7748.

[40] C. Lerche. *Tokio-Rs/Tokio*. Tokio, Jan. 5, 2025. URL: https://github.com/tokio-rs/tokio (visited on 01/05/2025).

[41] C. Lerche, T. de Zeeuw, and tokio-rs. *Mio*. Lib.rs. Aug. 12, 2024. URL: https://lib.rs/crates/mio (visited on 09/20/2024).

[42] Y. Liu, J. Wang, J. Li, S. Niu, and H. Song. *Machine Learning for the Detection and Identification of Internet of Things (IoT) Devices: A Survey*. Jan. 25, 2021. DOI: 10.48550/arXiv.2101.10181. arXiv: 2101.10181 [cs]. URL: http://arxiv.org/abs/2101.10181 (visited on 01/05/2025). Pre-published.

[43] L. Lundblade, G. Mandyam, J. O'Donoghue, and C. Wallace. *The Entity Attestation Token (EAT)*. Internet Draft draft-ietf-rats-eat-25. Internet Engineering Task Force, Jan. 15, 2024. 101 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-rats-eat (visited on 02/05/2024).

[44] J. P. Mattsson, G. Selander, S. Raza, J. Höglund, and M. Furuhed. *CBOR Encoded X.509 Certificates (C509 Certificates)*. Internet-Draft draft-ietf-cose-cbor-encoded-cert-11. Internet Engineering Task Force / Internet Engineering Task Force, July 8, 2024. 73 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-cose-cbor-encoded-cert/11/.

[45] S. McArthur. *Seanmonstar/Reqwest*. Jan. 5, 2025. URL: https://github.com/seanmonstar/reqwest (visited on 01/05/2025).

[46] N. McCallum and L. Mammino. *Enarx/Ciborium*. Enarx, Jan. 2, 2025. URL: https://github.com/enarx/ciborium (visited on 01/05/2025).

[47] Y. Nir, S. Josefsson, and M. Pégourié-Gonnard. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*. RFC 8422. RFC Editor, Aug. 2018. DOI: 10.17487/RFC8422. URL: https://www.rfc-editor.org/info/rfc8422.

[48] M. ooley. *Exploring Bluetooth 5 -What's New in Advertising?* Bluetooth® Technology Website. Feb. 27, 2017. URL: https://www.bluetooth.com/blog/exploring-bluetooth5-whats-new-in-advertising/ (visited on 12/17/2024).

[49] R. Organization. *Rust Crypto*. Apr. 1, 2020. URL: https://github.com/RustCrypto (visited on 01/05/2025).

[50] R. Prakash, J. Neeli, and S. Manjunatha. "A Survey of Security Challenges, Attacks in IoT". In: *E3S Web of Conferences* (Feb. 21, 2024). DOI: 10.1051/e3sconf/202449104018.

[51] M. Pritikin, M. Richardson, T. Eckert, M. H. Behringer, and K. Watsen. *Bootstrapping Remote Secure Key Infrastructure (BRSKI)*. RFC 8995. RFC Editor, May 2021. DOI: 10.17487/RFC8995. URL: https://www.rfc-editor.org/info/rfc8995.

[52] M. Pritikin, P. E. Yee, and D. Harkins. *Enrollment over Secure Transport*. RFC 7030. RFC Editor, Oct. 2013. DOI: 10.17487/RFC7030. URL: https://www.rfc-editor.org/info/rfc7030.

[53] V. V. Rao, R. Marshal, and K. Gobinath. "The IoT Supply Chain Attack Trends-Vulnerabilities and Preventive Measures". In: *2021 4th International Conference on Security and Privacy (ISEA-ISAP)*. 2021. DOI: 10.1109/ISEA-ISAP54304.2021.9689704.

[54] A. Rebert and C. Kern. *Secure by Design: Google's Perspective on Memory Safety*. Google Security Engineering, 2024.

[55] M. Richardson, P. V. der Stok, P. Kampanakis, and E. Dijk. *Constrained Bootstrapping Remote Secure Key Infrastructure (cBRSKI)*. Internet-Draft draft-ietf-anima-constrained-voucher-25. Internet Engineering Task Force / Internet Engineering Task Force, July 8, 2024. 88 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-anima-constrained-voucher/25/.

[56] Rust Foundation. *Crates.Io: Rust Package Registry*. June 25, 2024. URL: https://crates.io/ (visited on 12/19/2024).

[57] T. Sasi, A. H. Lashkari, R. Lu, P. Xiong, and S. Iqbal. "A Comprehensive Survey on IoT Attacks: Taxonomy, Detection Mechanisms and Challenges". In: *Journal of Information and Intelligence* 6 (2024). ISSN: 2949-7159. DOI: 10.1016/j.jiixd.2023.12.001.

[58] J. Schaad. *CBOR Object Signing and Encryption (COSE)*. RFC 8152. RFC Editor, July 2017. DOI: 10.17487/RFC8152. URL: https://www.rfc-editor.org/info/rfc8152.

[59] J. Schaad. *CBOR Object Signing and Encryption (COSE): Initial Algorithms*. RFC 9053. RFC Editor, Aug. 2022. DOI: 10.17487/RFC9053. URL: https://www.rfc-editor.org/info/rfc9053.

[60] J. Schaad. *CBOR Object Signing and Encryption (COSE): Structures and Process*. RFC 9052. RFC Editor, Aug. 2022. DOI: 10.17487/RFC9052. URL: https://www.rfc-editor.org/info/rfc9052.

[61] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. RFC Editor, June 2014. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[62] M. Singh and M. Ranganathan. *Formal Verification of Bootstrapping Remote Secure Key Infrastructures (BRSKI) Protocol Using AVISPA*. National Institute of Standards and Technology, Oct. 7, 2020. DOI: 10.6028/NIST.TN.2123. URL: https://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.2123.pdf (visited on 01/07/2024).

[63] S. Sinha. *State of IoT 2023: Number of Connected IoT Devices Growing 16% to 16.7 Billion Globally*. IoT Analytics. May 24, 2023. URL: https://iot-analytics.com/number-connected-iot-devices/ (visited on 01/28/2024).

[64] B. Smith. *Briansmith/Ring*. Oct. 15, 2024. URL: https://github.com/briansmith/ring (visited on 10/16/2024).

[65] F. Stajano and R. Anderson. "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks". In: *Security Protocols*. Ed. by B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000. ISBN: 978-3-540-45570-7. DOI: 10.1007/10720107_24.

[66] M. StJohns. *Attestation Attributes for Use with Certification Signing Requests*. Internet Draft draft-stjohns-csr-attest-02. Internet Engineering Task Force, June 7, 2023. 17 pp. URL: https://datatracker.ietf.org/doc/draft-stjohns-csr-attest (visited on 02/05/2024).

[67] *Supply Chain Compromise, Technique T1195 - Enterprise | MITRE ATT&CK®*. Apr. 18, 2018. URL: https://attack.mitre.org/techniques/T1195/ (visited on 01/28/2024).

[68] S. Symington, W. Polk, and M. Souppaya. *Trusted Internet of Things (IoT) Device Network-Layer Onboarding and Lifecycle Management*. Whitepaper. NIST, Sept. 8, 2020. URL: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.09082020-draft.pdf (visited on 01/03/2025).

[69] R. S. Team. *2023 Annual Rust Survey Results | Rust Blog*. Feb. 19, 2024. URL: https://blog.rust-lang.org/2024/02/19/2023-Rust-Annual-Survey-2023-results.html (visited on 12/16/2024).

[70] Tony Arcieri, Nathaniel McCallum, and Trevor Gross. *RustCrypto/JOSE*. Rust Crypto, Nov. 5, 2024. URL: https://github.com/RustCrypto/JOSE (visited on 12/19/2024).

[71] L. S. Vailshery. *IoT Connected Devices Worldwide 2019-2030*. Statista. July 27, 2023. URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (visited on 01/28/2024).

[72] C. Wallace and S. Turner. *Key Attestation Extension for Certificate Management Protocols*. Internet Draft draft-ietf-lamps-key-attestation-ext-00. Internet Engineering Task Force, Oct. 17, 2022. 14 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-lamps-key-attestation-ext (visited on 02/05/2024).

[73] K. Watsen, M. Richardson, M. Pritikin, T. Eckert, and Q. Ma. *A Voucher Artifact for Bootstrapping Protocols*. Internet-Draft draft-ietf-anima-rfc8366bis-12. Internet Engineering Task Force / Internet Engineering Task Force, July 8, 2024. 43 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-anima-rfc8366bis/12/.

[74] M. H. Zoualfaghari and A. Reeves. "Secure & Zero Touch Device Onboarding". In: *Living in the Internet of Things (IoT 2019)*. Jan. 2, 2019. DOI: 10.1049/cp.2019.0133.

Glossary

ACME Automatic Certificate Management Environment.

BLE Bluetooth Low Energy.
BRSKI Bootstrapping Remote Secure Key Infrastructure.
BRSKI-PRM Bootstrapping Remote Secure Key Infrastructure with Pledge in Responder Mode.

CA Certificate Authority.
CBOR Concise Binary Object Representation.
cBRSKI Constrained Bootstrapping Remote Secure Key Infrastructure.
cBRSKI-PRM Constrained Bootstrapping Remote Secure Key Infrastructure with Pledge in Responder Mode.
CMS Cryptographic Message Syntax.
CoAP Constrained Application Protocol.
COSE CBOR Object Signing and Encryption.
CSR Certificate Signing Request.

DHCP Dynamic Host Configuration Protocol.
DNS-SD DNS Service Discovery.

ECC Elliptic Curve Cryptography.
EDCC European Dependable Computing Conference.
EST Enrollment over Secure Transport.

GATT Generic Attribute Profile.

HTTP Hypertext Transfer Protocol.

IoT Internet-of-Things.
IP Internet Protocol.

JOSE JavaScript Object Signing and Encryption.
JSON JavaScript Object Notation.
JWS JSON Web Signature.

MASA Manufacturer Authorized Signing Authority.

RAM Random Access Memory.
ROM Read-Only Memory.
RVR Registrar Voucher Request.

TLS Transport Layer Security.
TOFU Trust On First Use.

A Listings

```
mode = "PRM" # unspecified "other" mode not implemented

[masa]
ca_certificate = "vendor-ca.cert"
ca_key = "vendor-ca.key"
masa_certificate = "vendor.cert"
masa_key = "vendor.key"
registrar_ee_certificate = "registrar.cert"
masa_url = "http://localhost:3000"

[registrar]
ca_certificate = "registrar-ca.cert"
ca_key = "registrar-ca.key"
registrar_certificate = "registrar.cert"
registrar_key = "registrar.key"
reg_agt_ee_cert = "registrar-agent.cert"

[registrar_agent]
ee_certificate = "registrar-agent.cert"
ee_key = "registrar-agent.key"
registrar_certificate = "registrar-ca.cert"
registrar_url = "http://localhost:3001"
bootstrap_serials = ["00-D0-E5-F2-00-02"]

[pledge]
idevid_certificate = "pledge.cert"
idevid_privkey = "pledge.key"
idev_id = "00-D0-E5-F2-00-02"
```

Listing A.1: *Open-BRSKI Configuration File*

```
{
   "ietf-voucher-request:voucher": {
      "created-on": "2021-04-16T00:00:02.000Z",
      "nonce": "eDs++/FuDHGUnRxN3E14CQ==",
      "serial-number": "vendor-pledge4711",
      "assertion": "agent-proximity",
      "agent-provided-proximity-registrar-cert": "b64==",
      "agent-signed-data": "b64=="
   }
 }
```

Listing A.2: *Example of a Voucher-Request encoded in JSON*

```
pub trait SignerVerifyer<T> {
    fn sign(
        &self,
        payload: T,
        header: HeaderSet,
```

```
6           privkey: &[u8],
7       ) -> Result<Vec<u8>, SigneableError>;
8
9       fn verify(
10          &self,
11          signed_data: &[u8],
12      ) -> Result<VerifyResult<T>, SigneableError>;
13
14      fn add_signature(
15          &self,
16          signed_data: &[u8],
17          header: HeaderSet,
18          privkey: &[u8],
19      ) -> Result<Vec<u8>, SigneableError>;
20  }
21
22  pub struct BasicSigningContext {
23      skid: Option<String>,
24      algorithm: Algorithm,
25  }
26
27  pub struct BasicVeryingContext {
28      pub pub_key: Option<Vec<u8>>,
29  }
30
31  pub struct VerifyResult<T> {
32      pub(crate) payload: T,
33      pub(crate) headers: HeaderSet,
34  }
```

Listing A.3: *Signing and Verifying Interface*

```
1
2   let received_data = args.recv_data() // (1)
3   let deserialized: DataFrame = ... // (2)
4
5   let target_len = deserialized.header.length; // (3)
6   let offset = deserialized.header.offset; // (4)
7
8   write_buf.insert(offset, deserialized); // (5)
9
10  if write_buf.len() == target_len {
11      write_buf_ref.sort_by(|a, b|{
12          a.header.offset.cmp(&b.header.offset) // (6)
13      });
14      let arr = write_buf.iter(); // (7)
15      let payload = arr.iter().fold(vec![], |mut acc, df| {
16          acc.extend_from_slice(&df.data); // (8)
17          acc
18      });
19
20      write_buf.clear(); // (9)
21
22      let result: Response = route_handler(payload); // (10)
```

```
23
24      let computed_buf_ptr =
25          &mut computed_buf.lock(); // (11)
26      computed_buf_ptr.clear();
27      computed_buf_ptr.extend_from_slice(
28          &result.data()
29      ); // (12)
30  }
```

Listing A.4: *Packet Defragmentation in Write Characteristic*

```
1
2  let computed_buf_ptr = computed_buf_ptr_copy.lock() // (1);
3  let ident = _args.address().val()[0] // (2);
4  let frames =
5      DataFrame::split_from_vec(
6          computed_buf_ptr, ident
7      ); // (3)
8
9  let mut buf = [0u8; EXPECTED_DATA_SIZE]; // (4)
10 for frame in frames {
11     buf.clear();
12     serialize_func(&frame, &mut buf); // (5)
13     _this.set_value(&buf); // (6)
14 }
```

Listing A.5: *Packet Fragmentation in Read Characteristic*

```
1
2  const BTLE_MAX_PACKET_SIZE: usize = 512 - 3; // (0)
3
4  struct DataFrame {
5      header: DataFrameHeader // (1),
6      data: [u8; EXPECTED_DATA_SIZE] // (2),
7  }
8
9  const EXPECTED_DATA_SIZE: usize = {
10     BTLE_MAX_PACKET_SIZE
11         - std::mem::size_of::<DataFrameHeader>() // (3)
12 }
13 struct DataFrameHeader {
14     offset: u8, // (4)
15     ident: u8, // (5)
16     length: u8, // (6)
17 }
```

Listing A.6: *Data Frame for Fragmented Payloads*

```
1
2  pub trait PledgeCommunicator: Send + Sync { // (1)
3
4      async fn send_pvr_trigger( // (2)
5          &self,
6          trigger: Vec<u8>, // (3)
```

```
7           ctx: PledgeCtx, // (4)
8       ) -> Result<Vec<u8>, ServerError>; // (5)
9
10      async fn send_per_trigger(
11          &self,
12          trigger: Vec<u8>,
13          ctx: PledgeCtx,
14      ) -> Result<Vec<u8>, ServerError>;
15
16      async fn send_voucher(
17          &self, voucher: Vec<u8>, ctx: PledgeCtx
18      )-> Result<Vec<u8>, ServerError>;
19
20      async fn send_ca_certs(
21          &self, cacerts: Vec<u8>, ctx: PledgeCtx
22      ) -> Result<(), ServerError>;
23
24      async fn send_enroll_response(
25          &self,
26          cacerts: Vec<u8>,
27          ctx: PledgeCtx,
28      ) -> Result<Vec<u8>, ServerError>;
29
30      async fn get_data_interchange_format(
31          &self,
32          pledge: DiscoveredPledge,
33      ) -> Result<String, ServerError>;
34
35      async fn get_pledge_info(
36          &self,
37          pledge: DiscoveredPledge,
38          format: DataInterchangeFormat,
39      ) -> Result<Vec<u8>, ServerError>;
40  }
41
42
43  pub struct PledgeCtx { // (6)
44      pub ctx: String,
45      pub pledge_serial: String,
46      pub pledge_url: String,
47      pub pledge_info: PledgeInfo,
48  }
```

Listing A.7: *Pledge Communication Interface*

```
1
2   pub struct HTTPCommunicator { // (1)
3       client: reqwest::Client,
4   }
5
6   impl PledgeCommunicator for HTTPCommunicator {
7
8   async fn send_pvr_trigger(
9       &self,
```

```
    trigger: Vec<u8>,
    ctx: PledgeCtx,
) -> Result<Vec<u8>, ServerError> {
    let url = format!(
        "{}/.well-known/brski/tpvr", ctx.pledge_url // (2)
    );

    let response = self
        .client
        .post(url) // (3)
        .header(
            ACCEPT,
            ctx
            .pledge_info
            .supported_voucher_type
            .as_content_type(), // (4)
        )
        .header(
            CONTENT_TYPE,
            ctx
            .pledge_info
            .data_interchance_format
            .as_content_type(), // (5)
        )
        .body(trigger) // (6)
        .send()
        .await?;

    if !response.status().is_success() {
        return Err(ServerError::BadResponse(format!(
            "{}, {}",
            "Sending TPVR to Pledge failed",
            response.status()
        )));
    }

    // ...

    let response_data = response.bytes().await?; // (7)

    std::result::Result::Ok(response_data.to_vec()) // (8)
    }
}
```

Listing A.8: *HTTP Communication Implementation*

```

#[derive(Clone)]
pub struct FFIBLECommunicator {
    ffi_send_pvr_trigger: // (1)
        Arc< // (2)
            Box< // (3)
                dyn Fn( // (4)
                    Vec<u8>, PledgeCtx
```

```
                ) -> DartFnFuture< // (5)
                    Vec<u8>
                > + Sync + Send
            >
        >,
    ffi_send_per_trigger: ...
    ffi_send_voucher: ...
    ffi_send_ca_certs: ...
    ffi_send_enroll_response: ...
    ffi_get_data_interchange_format: ...
    ffi_get_pledge_info: ...
}
```

Listing A.9: *BLE FFI Communicator Structure*

```

#[async_trait::async_trait]
impl PledgeCommunicator for FFIBLECommunicator {
    async fn send_pvr_trigger(
        &self, trigger: Vec<u8>, ctx: PledgeCtx
    ) -> Result<Vec<u8>, ServerError> {
        let result =
            (self.ffi_send_pvr_trigger)(trigger, ctx)
            .await; // (1)
        Ok(result)
    }
    // ...
}
```

Listing A.10: *BLE FFI Communication Implementation*

```

Future<Bootstrapper> getPledgeFFIBootstrapper( // (0)
    BleRouter bleRouter // (1),
    CharacteristicPackage cpackage // (2)
) async {
    var builder = await
        FFIBLECommunicatorBuilder.init(); // (3)
    builder = await builder.
        setPvrFfi(callback: (trigger, ctx) =>
            bleRouter.route(
                cpackage.tpvr, trigger
            )); // (4)

    // ...

    var communicator = await builder.build(); // (5)

    var config = await getParsedConfig();

    var bootstraper =
        await Bootstrapper.init(
            config: config, communicator: communicator
    ); // (6)
```

```
24
25      return bootstraper;
26  }
```

Listing A.11: *Create FFI Bridge using Dart*

```
1  pub struct PledgeInfo {
2      pub data_interchance_format: DataInterchangeFormat,
3      pub supported_token_type: PlainTokenType,
4      pub supported_voucher_type: VoucherTokenType,
5  }
6
7  pub enum PlainTokenType {
8      JOSE,
9      COSE,
10 }
11
12 pub enum VoucherTokenType {
13     JWS,
14     COSE,
15 }
```

Listing A.12: *PledgeInfo Structure*

```
1  CONFIG_ESP_MAIN_TASK_STACK_SIZE=16000 # (1)
2  CONFIG_BT_NIMBLE_HOST_TASK_STACK_SIZE=16000 # (2)
3
4  ### (3) ########################
5  CONFIG_BT_ENABLED=y
6  CONFIG_BT_BLE_ENABLED=y
7  CONFIG_BT_BLUEDROID_ENABLED=n
8  CONFIG_BT_NIMBLE_ENABLED=y
9  CONFIG_BT_NIMBLE_NVS_PERSIST=y
10 CONFIG_BT_NIMBLE_EXT_ADV=y (4)
11 ################################
12
13 ### (5) ########################
14 CONFIG_BTDM_CTRL_MODE_BLE_ONLY=y
15 CONFIG_BTDM_CTRL_MODE_BR_EDR_ONLY=n
16 CONFIG_BTDM_CTRL_MODE_BTDM=n
17 ################################
18
19 ### (6) ########################
20 CONFIG_ESP_SYSTEM_PANIC_PRINT_REBOOT=n
21 CONFIG_ESP_SYSTEM_PANIC_PRINT_HALT=y
22 CONFIG_ESP_SYSTEM_USE_EH_FRAME=y
```

Listing A.13: *Configuration using sdkconfig.defaults*

```
1  use esp_idf_svc::io::vfs::initialize_eventfd;
2
3  fn main() {
4
5      initialize_eventfd(1).unwrap(); // (1)
```

```
6
7       tokio::runtime::Builder::new_current_thread() // (2)
8           .enable_all() // (3)
9           .build() // (4)
10          .unwrap()
11          .block_on(async move {
12              run().await; // (5)
13          });
14  }
```

Listing A.14: *Setting up the async Runtime*

```
1
2   async fn run() {
3       let peripherals = Peripherals::take(); // (1)
4       let sysloop = EspSystemEventLoop::take(); // (2)
5       let timer = EspTaskTimerService::new(); // (3)
6       let nvs = EspDefaultNvsPartition::take(); // (4)
7
8       let esp_wifi = EspWifi::new( // (5)
9           peripherals.modem,
10          sysloop.clone(),
11          Some(nvs)
12          ).unwrap();
13
14      let wifi = AsyncWifi::wrap( // (6)
15          esp_wifi,
16          sysloop,
17          timer
18      ).unwrap();
19
20      join!(run_wifi(wifi), run_ble()); // (7)
21
22      loop {
23          tokio::time::sleep(
24              tokio::time::Duration::from_secs(1) // (8)
25          ).await;
26      }
27  }
```

Listing A.15: *ESP32 Main Task*

```
1
2   loop {
3       FreeRtos::delay_ms(1000);
4   }
```

Listing A.16: *Resetting the Watchdog Timer*

B Tables

Origin	Function	Description	UUID
Native	TPVR	Service	9b574847-f706-436c-bed7-fc01eb0965c1
		Read	9b574847-f706-436c-bed7-fc01eb0965c2
		Write	9b574847-f706-436c-bed7-fc01eb0965c3
	TPER	Service	bd2ccaf3-11b7-459d-b469-3b223318a49d
		Read	bd2ccaf3-11b7-459d-b469-3b223318a49e
		Write	bd2ccaf3-11b7-459d-b469-3b223318a49f
	SVR	Service	98daa21b-b987-4285-92fc-73089b9b45f3
		Read	98daa21b-b987-4285-92fc-73089b9b45f4
		Write	98daa21b-b987-4285-92fc-73089b9b45f5
	SCAC	Service	79739f27-6baf-4f91-9b82-37e59146e15a
		Read	79739f27-6baf-4f91-9b82-37e59146e15b
		Write	79739f27-6baf-4f91-9b82-37e59146e15c
	SER	Service	d58c360d-08eb-4106-8b9a-d3c3ebffbf51
		Read	d58c360d-08eb-4106-8b9a-d3c3ebffbf52
		Write	d58c360d-08eb-4106-8b9a-d3c3ebffbf53
	QPS	Service	3fdfbece-8c96-4420-a325-2d9a8ebe8044
		Read	3fdfbece-8c96-4420-a325-2d9a8ebe8045
		Write	3fdfbece-8c96-4420-a325-2d9a8ebe8046
Custom	PI	Service	4175a295-911a-4234-ba82-29e1becd149f
		Read	4175a295-911a-4234-ba82-29e1becd149d
		Write	4175a295-911a-4234-ba82-29e1becd149e
	DI	Service	aa84c01b-6112-4701-88d8-5982764bbf11
		Read	aa84c01b-6112-4701-88d8-5982764bbf12
		Write	aa84c01b-6112-4701-88d8-5982764bbf13

Table B.1: *UUIDs mapped to BRSKI-PRM REST routes*

C Certificates

C.1 Pledge Certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 49 (0x31)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer:
            commonName                = masa-ca.example.com CA
        Validity
            Not Before: Oct 23 17:22:26 2024 GMT
            Not After : Oct 23 17:22:26 2025 GMT
        Subject:
            serialNumber              = 00-D0-E5-F2-00-02
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:3c:3c:ba:08:2b:8a:99:1f:92:44:1c:2e:42:69:
                    6b:47:af:be:02:1b:7b:20:44:11:6a:25:ba:ff:07:
                    91:7b:5c:9d:60:ee:86:f2:b9:9c:42:47:b7:e6:23:
                    c9:23:97:4f:0d:cf:04:27:fc:9e:44:f3:5b:11:03:
                    d9:73:45:1f:95
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:00-D0-E5-F2-00-02
            X509v3 Subject Key Identifier:
                C5:7A:3C:6E:D8:38:22:C5:C2:CA:D3:
                    9C:AC:25:2C:BD:02:FC:16:D9
            X509v3 Basic Constraints: critical
                CA:FALSE
            1.3.6.1.5.5.7.1.32:
                localhost:3000
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:7b:b4:be:17:3d:e1:82:03:a7:de:85:47:c4:1f:
         59:e8:46:d8:3f:5e:a9:9f:4d:b1:de:48:88:e8:cc:0b:4e:ac:
         02:21:00:8a:80:4c:88:48:70:35:75:3a:3b:80:25:75:bd:1b:
         78:5c:58:61:a4:33:22:0b:5f:4f:15:54:14:45:89:45:ed
-----BEGIN CERTIFICATE-----
MIIBmDCCAT6gAwIBAgIBMTAKBggqhkjOPQQDAjAhMR8wHQYDVQQDDBZtYXNhLWNh
LmV4YW1wbGUuY29tIENBMB4XDTI0MTAyMzE3MjIyNloXDTI1MTAyMzE3MjIyNlow
HDEaMBgGA1UEBQwRMDAtRDAtRTUtRjItMDAtMDIwWTATBgcqhkjOPQIBBggqhkjO
PQMBBwNCAAQ8PLoIK4qZH5JEHC5CaWtHr74CG3sgRBFqJbr/B5F7XJ1g7obyuZxC
R7fmI8kjl08NzwQn/J5E81sRA9lzRR+Vo2wwajAcBgNVHREEFTATghEwMC1EMC1F
```

```
NS1GMi0wMC0wMjAdBgNVHQ4EFgQUxXo8btg4IsXCytOcrCUsvQL8FtkwDwYDVR0T
AQH/BAUwAwEBADAaBggrBgEFBQcBIAQObG9jYWxob3N0OjMwMDAwCgYIKoZIzj0E
AwIDSAAwRQIge7S+Fz3hggOn3oVHxB9Z6EbYP16pn02x3kiI6MwLTqwCIQCKgEyI
SHA1dTo7gCV1vRt4XFhhpDMiC19PFVQURYlF7Q==
-----END CERTIFICATE-----
```

Listing C.1: *MASA issued Pledge EE Certificate (IdevID)*

C.2 Registrar-Agent Certificate

```
Certificate:
   Data:
       Version: 3 (0x2)
       Serial Number:
           1b:1e:0b:7b:5e:7c:a0:6b:76:b0:fd:cd:
           9a:84:dd:da:32:59:4f:f1
   Signature Algorithm: ecdsa-with-SHA256
       Issuer:
           organizationalUnitName
               = Department of Computer Science
           organizationName
               = University of Applied Sciences Munich
           localityName              = Munich
           stateOrProvinceName       = Bavaria
           countryName               = DE
           commonName
               = registrar-ca.example.com Root CA
       Validity
           Not Before: Oct 23 17:22:26 2024 GMT
           Not After : Nov 20 17:22:26 2024 GMT
       Subject:
           organizationalUnitName
               = Department of Computer Science
           organizationName
               = University of Applied Sciences Munich
           localityName              = Munich
           stateOrProvinceName       = Bavaria
           countryName               = DE
           commonName
               = registrar-agent.example.com
       Subject Public Key Info:
           Public Key Algorithm: id-ecPublicKey
               Public-Key: (256 bit)
               pub:
                   04:fe:15:09:85:54:ad:a5:c6:6b:aa:8f:47:15:e2:
                   22:bb:98:0f:e0:35:b8:ba:f0:60:58:80:52:ff:ba:
                   03:06:17:f1:b5:7d:bf:46:37:60:94:03:43:cb:d3:
                   56:e7:15:ef:72:4e:9f:e6:5d:2e:17:6d:4d:27:12:
```

```
                    58:5d:ad:28:10
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:BRSKI Registrar-Agent
            X509v3 Extended Key Usage:
                TLS Web Client Authentication, Code Signing
            X509v3 Subject Key Identifier:
                7A:47:D7:BB:54:BA:BC:AA:DC:DB:E3:F8:6D:60:F2:
                F5:8A:23:C9:55
            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:af:e6:16:3f:9f:dc:e5:d3:b0:c6:7c:6e:83:
         47:ae:cb:80:33:29:0b:19:8d:87:d3:f1:ff:57:c4:2d:4f:c8:
         4e:02:21:00:ac:d6:c4:14:4a:25:e4:cb:30:92:02:c5:35:9d:
         a3:38:b7:e1:af:b8:38:d2:5e:eb:9d:f4:70:d4:76:42:64:4c
-----BEGIN CERTIFICATE-----
MIIC3jCCAoOgAwIBAgIUGx4Le158oGt2sP3NmoTd2jJZT/EwCgYIKoZIzj0EAwIw
gbQxKTAnBgNVBAMMIHJlZ2lzdHJhci1jYS5leGFtcGxlLmNvbSBSb290IENBMQsw
CQYDVQQGDAJERTEQMA4GA1UECAwHQmF2YXJpYTEPMA0GA1UEBwwGTXVuaWNoMS4w
LAYDVQQKDCVVbml2ZXJzaXR5IG9mIEFwcGxpZWQgU2NpZW5jZXMgTXVuaWNoMScw
JQYDVQQLDB5EZXBhcnRtZW50IG9mIENvbXB1dGVyIFNjaWVuY2UwHhcNMjQxMDIz
MTcyMjI2WhcNMjQxMTIwMTcyMjI2WjCBrzEkMCIGA1UEAwwbcmVnaXN0cmFyLWFn
ZW50LmV4YW1wbGUuY29tMQswCQYDVQQGDAJERTEQMA4GA1UECAwHQmF2YXJpYTEP
MA0GA1UEBwwGTXVuaWNoMS4wLAYDVQQKDCVVbml2ZXJzaXR5IG9mIEFwcGxpZWQg
U2NpZW5jZXMgTXVuaWNoMScwJQYDVQQLDB5EZXBhcnRtZW50IG9mIENvbXB1dGVy
IFNjaWVuY2UwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAAT+FQmFVK2lxmuqj0cV
4iK7mA/gNbi68GBYgFL/ugMGF/G1fb9GN2CUA0PL01bnFe9yTp/mXS4XbU0nElhd
rSgQo3YwdDAgBgNVHREEGTAXghVCUlNLSSBSZWdpc3RyYXItQWdlbnQwHQYDVR0l
BBYwFAYIKwYBBQUHAwIGCCsGAQUFBwMDMB0GA1UdDgQWBBR6R9e7VLq8qtzb4/ht
YPL1iiPJVTASBgNVHRMBAf8ECDAGAQH/AgEAMAoGCCqGSM49BAMCA0kAMEYCIQCv
5hY/n9zl07DGfG6DR67LgDMpCxmNh9Px/1fELU/ITgIhAKzWxBRKJeTLMJICxTWd
ozi34a+4ONJe6530cNR2QmRM
-----END CERTIFICATE-----
```

Listing C.2: *Registrar-Agent EE certificate*

C.3 Registrar Certificates

```
Certificate:
   Data:
       Version: 3 (0x2)
       Serial Number:
           4b:ae:29:1a:90:81:bd:cf:1d:15:83
           :15:15:40:b9:e9:b2:ed:3b:2a
   Signature Algorithm: ecdsa-with-SHA256
       Issuer:
```

```
            organizationalUnitName
                = Department of Computer Science
            organizationName
                = University of Applied Sciences Munich
            localityName              = Munich
            stateOrProvinceName       = Bavaria
            countryName               = DE
            commonName
                = registrar-ca.example.com Root CA
        Validity
            Not Before: Oct 23 17:22:26 2024 GMT
            Not After : Oct 23 17:22:26 2025 GMT
        Subject:
            organizationalUnitName
                = Department of Computer Science
            organizationName
                = University of Applied Sciences Munich
            localityName              = Munich
            stateOrProvinceName       = Bavaria
            countryName               = DE
            commonName
                = registrar-ca.example.com Root CA
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:f3:92:0d:7a:21:3a:5a:b4:02:9f:11:65:d2:e0:
                    1c:6c:7a:b0:1e:29:47:4f:c6:eb:3f:9e:56:8b:01:
                    4e:b5:81:78:22:36:62:6f:f0:bb:b7:52:b6:77:85:
                    38:a3:93:19:a1:15:90:fb:1f:f1:7b:0e:83:de:51:
                    b8:cb:08:91:a4
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:FC:3A:AD:3A:1B:61:C6:5F:06
                :A0:E7:DE:34:7B:E0:4F:FD:66:A5:37

            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Subject Key Identifier:
                FC:3A:AD:3A:1B:61:C6:5F:06:A0:
                E7:DE:34:7B:E0:4F:FD:66:A5:37
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:5b:05:f9:8d:62:1f:ad:d9:b6:5b:c5:2b:73:b7:
         34:f9:0f:e2:04:fe:50:af:4e:d5:06:f6:91:d7:22:77:56:8e:
         02:21:00:93:33:d3:d5:af:1c:b5:ff:77:19:20:12:14:de:eb:
         19:a8:03:2b:6b:38:d7:42:f2:91:6d:f9:53:e8:73:ae:e0
```

```
-----BEGIN CERTIFICATE-----
MIICzzCCAnWgAwIBAgIUS64pGpCBvc8dFYMVFUC56bLtOyowCgYIKoZIzj0EAwIw
gbQxKTAnBgNVBAMMIHJlZ2lzdHJhci1jYS5leGFtcGxlLmNvbSBSb290IENBMQsw
CQYDVQQGDAJERTEQMA4GA1UECAwHQmF2YXJpYTEPMA0GA1UEBwwGTXVuaWNoMS4w
LAYDVQQKDCVVbml2ZXJzaXR5IG9mIEFwcGxpZWQgU2NpZW5jZXMgTXVuaWNoMScw
JQYDVQQLDB5EZXBhcnRtZW50IG9mIENvbXB1dGVyIFNjaWVuY2UwHhcNMjQxMDIz
MTcyMjI2WhcNMjUxMDIzMTcyMjI2WjCBtDEpMCcGA1UEAwwgcmVnaXN0cmFyLWNh
LmV4YW1wbGUuY29tIFJvb3QgQ0ExCzAJBgNVBAYMAkRFMRAwDgYDVQQIDAdCYXZh
cmlhMQ8wDQYDVQQHDAZNdW5pY2gxLjAsBgNVBAoMJVVuaXZlcnNpdHkgb2YgQXBw
bGllZCBTY2llbmNlcyBNdW5pY2gxJzAlBgNVBAsMHkRlcGFydG1lbnQgb2YgQ29t
cHV0ZXIgU2NpZW5jZTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABPOSDXohOlq0
Ap8RZdLgHGx6sB4pR0/G6z+eVosBTrWBeCI2Ym/wu7dStneFOKOTGaEVkPsf8XsO
g95RuMsIkaSjYzBhMB8GA1UdIwQYMBaAFPw6rTobYcZfBqDn3jR74E/9ZqU3MA4G
A1UdDwEB/wQEAwIBBjAdBgNVHQ4EFgQU/DqtOhthxl8GoOfeNHvgT/1mpTcwDwYD
VR0TAQH/BAUwAwEB/zAKBggqhkjOPQQDAgNIADBFAiBbBfmNYh+t2bZbxStztzT5
D+IE/lCvTtUG9pHXIndWjgIhAJMz09WvHLX/dxkgEhTe6xmoAytrONdC8pFt+VPo
c67g
-----END CERTIFICATE-----
```

Listing C.3: *Registrar Certificate Authority certificate*

```
Certificate:
   Data:
       Version: 3 (0x2)
       Serial Number:
           15:bb:1b:a2:30:0c:6d:ab:7d:82:1c:19
           :36:ac:2b:64:c0:77:93:78
   Signature Algorithm: ecdsa-with-SHA256
       Issuer:
           organizationalUnitName
               = Department of Computer Science
           organizationName
               = University of Applied Sciences Munich
           localityName              = Munich
           stateOrProvinceName       = Bavaria
           countryName               = DE
           commonName
               = registrar-ca.example.com Root CA
       Validity
           Not Before: Oct 23 17:22:26 2024 GMT
           Not After : Oct 23 17:22:26 2025 GMT
       Subject:
           organizationalUnitName
               = Department of Computer Science
           organizationName
               = University of Applied Sciences Munich
           localityName              = Munich
           stateOrProvinceName       = Bavaria
           countryName               = DE
           commonName                = registrar.example.com
```

```
30          Subject Public Key Info:
31              Public Key Algorithm: id-ecPublicKey
32                  Public-Key: (256 bit)
33                  pub:
34                      04:fa:da:04:d6:20:8c:a0:47:c1:70:99:5b:99:7d:
35                      7e:48:0b:7a:8b:62:35:84:3f:d4:62:2d:59:0c:28:
36                      f4:4f:c3:bd:78:c8:f1:7f:fc:2a:ca:2f:08:36:ab:
37                      c0:cc:2c:c2:c2:36:b7:06:97:a0:48:fa:09:0c:12:
38                      f7:99:2c:c1:e9
39                  ASN1 OID: prime256v1
40          X509v3 extensions:
41              X509v3 Authority Key Identifier:
42                  keyid:FC:3A:AD:3A:1B:61:C6:5F:06:
43                  A0:E7:DE:34:7B:E0:4F:FD:66:A5:37
44
45              X509v3 Key Usage: critical
46                  Certificate Sign
47              X509v3 Extended Key Usage:
48                  1.3.6.1.5.5.7.3.28
49      Signature Algorithm: ecdsa-with-SHA256
50           30:45:02:20:13:a9:79:d1:38:3a:6a:b8:8f:41:9b:fb:cf:8b:
51           81:40:57:37:93:73:bd:9c:f7:8d:64:f6:ae:91:b1:8e:fa:46:
52           02:21:00:b2:ce:be:5a:2d:1c:65:31:ba:e6:d1:0d:bd:f5:d3:
53           3b:78:71:71:04:36:49:90:54:90:ef:b4:b9:8a:6a:6e:77
54  -----BEGIN CERTIFICATE-----
55  MIICqTCCAk+gAwIBAgIUFbsbojAMbat9ghwZNqwrZMB3k3gwCgYIKoZIzj0EAwIw
56  gbQxKTAnBgNVBAMMIHJlZ2lzdHJhci1jYS5leGFtcGxlLmNvbSBSb290IENBMQsw
57  CQYDVQQGDAJERTEQMA4GA1UECAwHQmF2YXJpYTEPMA0GA1UEBwwGTXVuaWNoMS4w
58  LAYDVQQKDCVVbml2ZXJzaXR5IG9mIEFwcGxpZWQgU2NpZW5jZXMgTXVuaWNoMScw
59  JQYDVQQLDB5EZXBhcnRtZW50IG9mIENvbXB1dGVyIFNjaWVuY2UwHhcNMjQxMDIz
60  MTcyMjI2WhcNMjUxMDIzMTcyMjI2WjCBqTEeMBwGA1UEAwwVcmVnaXN0cmFyLmV4
61  YW1wbGUuY29tMQswCQYDVQQGDAJERTEQMA4GA1UECAwHQmF2YXJpYTEPMA0GA1UE
62  BwwGTXVuaWNoMS4wLAYDVQQKDCVVbml2ZXJzaXR5IG9mIEFwcGxpZWQgU2NpZW5j
63  ZXMgTXVuaWNoMScwJQYDVQQLDB5EZXBhcnRtZW50IG9mIENvbXB1dGVyIFNjaWVu
64  Y2UwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAAT62gTWIIygR8FwmVuZfX5IC3qL
65  YjWEP9RiLVkMKPRPw714yPF//CrKLwg2q8DMLMLCNrcGl6BI+gkMEveZLMHpo0gw
66  RjAfBgNVHSMEGDAWgBT8Oq06G2HGXwag5940e+BP/WalNzAOBgNVHQ8BAf8EBAMC
67  AgQwEwYDVR0lBAwwCgYIKwYBBQUHAxwwCgYIKoZIzj0EAwIDSAAwRQIgE6l50Tg6
68  ariPQZv7z4uBQFc3k3O9nPeNZPaukbGO+kYCIQCyzr5aLRxlMbrm0Q299dM7eHFx
69  BDZJkFSQ77S5impudw==
70  -----END CERTIFICATE-----
```

Listing C.4: *Registrar EE certificate*

C.4 MASA Certificates

```
1       Certificate:
2       Data:
```

```
        Version: 3 (0x2)
        Serial Number:
            65:58:31:04:63:d2:57:9e:9c:
            ba:9d:4b:35:c5:b7:48:0d:75:29:42
    Signature Algorithm: ecdsa-with-SHA256
        Issuer:
            commonName
                = masa-ca.example.com CA
        Validity
            Not Before: Oct 23 17:22:26 2024 GMT
            Not After : Oct 23 17:22:26 2999 GMT
        Subject:
            commonName
                = masa-ca.example.com CA
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:07:ef:9f:3b:a1:e8:93:a7:b0:8e:22:58:30:25:
                    54:57:ad:10:ce:1a:17:ed:28:a0:b7:15:5e:bf:a6:
                    e5:cb:9e:fe:15:53:b5:32:42:11:39:25:a3:65:b1:
                    5b:b0:cc:84:46:82:f9:ef:96:63:82:6c:bc:79:c4:
                    ce:d7:4a:cd:34
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:C8:6D:77:73:AC:91:D5:E4:97
                :27:9C:A8:B6:CC:79:80:5D:E2:2F:E7

            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Subject Key Identifier:
                C8:6D:77:73:AC:91:D5:E4:97:27:9C
                :A8:B6:CC:79:80:5D:E2:2F:E7
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:ef:d0:6c:76:8b:25:3e:d7:f8:d5:4f:e6:36:
         8d:1c:26:d6:f7:ec:29:b1:68:2a:b3:a6:95:b7:ef:7e:c5:48:
         57:02:21:00:c8:0d:67:f6:cb:87:66:ee:89:a2:95:5f:75:a2:
         82:70:66:79:51:22:5c:46:c2:b3:59:f4:b1:3b:85:76:39:1b
-----BEGIN CERTIFICATE-----
MIIBqjCCAU+gAwIBAgIUZVgxBGPSV56cup1LNcW3SA11KUIwCgYIKoZIzj0EAwIw
ITEfMB0GA1UEAwwWbWFzYS1jYS5leGFtcGxlLmNvbSBDQTAgFw0yNDEwMjMxNzIy
MjZaGA8yOTk5MTAyMzE3MjIyNlowITEfMB0GA1UEAwwWbWFzYS1jYS5leGFtcGxl
LmNvbSBDQTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABAfvnzuh6JOnsI4iWDAl
VFetEM4aF+0ooLcVXr+m5cue/hVTtTJCETklo2WxW7DMhEaC+e+WY4JsvHnEztdK
zTSjYzBhMB8GA1UdIwQYMBaAFMhtd3OskdXklyecqLbMeYBd4i/nMA4GA1UdDwEB
/wQEAwIBBjAdBgNVHQ4EFgQUyG13c6yR1eSXJ5yotsx5gF3iL+cwDwYDVR0TAQH/
```

```
BAUwAwEB/zAKBggqhkjOPQQDAgNJADBGAiEA79BsdoslPtf41U/mNo0cJtb37Cmx
aCqzppW3737FSFcCIQDIDWf2y4dm7omilV91ooJwZnlRIlxGwrNZ9LE7hXY5Gw==
-----END CERTIFICATE-----
```

Listing C.5: *MASA CA certificate*

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            3f:28:fe:86:87:f9:d9:cc:71:5e:
            23:9f:cf:f5:1b:a6:bf:9d:30:2c
    Signature Algorithm: ecdsa-with-SHA256
        Issuer:
            commonName                = masa-ca.example.com CA
        Validity
            Not Before: Oct 23 17:22:26 2024 GMT
            Not After : Oct 23 17:22:26 2999 GMT
        Subject:
            commonName                = masa-ca.example.com MASA
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:a5:62:c3:2f:a9:ee:34:71:c0:c9:97:9e:22:22:
                    9c:97:22:8d:3b:e5:07:92:ed:46:77:53:9d:1e:d5:
                    ce:c2:48:6b:e0:5f:bf:5c:00:69:04:68:49:e6:59:
                    0f:a2:33:d4:23:c7:28:f1:d8:0d:00:cc:4b:e8:fe:
                    ce:9b:7b:2e:2c
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:C8:6D:77:73:AC:91:D5:E4:97:
                27:9C:A8:B6:CC:79:80:5D:E2:2F:E7

            X509v3 Subject Key Identifier:
                B5:E9:DB:D0:12:ED:D2:66:5F:DC:
                CE:7B:7E:53:DA:59:F6:AD:9B:7C
            X509v3 Basic Constraints: critical
                CA:FALSE
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:75:a7:2b:9c:3a:b9:7f:9c:b7:c5:9e:6d:f9:73:
         40:2a:37:be:93:d9:41:dc:94:56:71:2a:b1:bc:b6:de:06:94:
         02:20:3e:58:ef:14:85:22:14:f4:a4:9b:38:8f:65:07:5f:ab:
         68:48:48:dc:5e:f3:a3:02:9b:41:5a:f6:05:51:56:92
-----BEGIN CERTIFICATE-----
MIIBmjCCAUGgAwIBAgIUPyj+hof52cxxXiOfz/Ubpr+dMCwwCgYIKoZIzj0EAwIw
ITEfMB0GA1UEAwwWbWFzYS1jYS5leGFtcGxlLmNvbSBDQTAgFw0yNDEwMjMxNzIy
MjZaGA8yOTk5MTAyMzE3MjIyNlowIzEhMB8GA1UEAwwYbWFzYS1jYS5leGFtcGxl
LmNvbSBNQVNBMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEpWLDL6nuNHHAyZee
```

```
45 IiKclyKNO+UHku1Gd1OdHtXOwkhr4F+/XABpBGhJ5lkPojPUI8co8dgNAMxL6P7O
46 m3suLKNTMFEwHwYDVR0jBBgwFoAUyG13c6yR1eSXJ5yotsx5gF3iL+cwHQYDVR0O
47 BBYEFLXp29AS7dJmX9zOe35T2ln2rZt8MA8GA1UdEwEB/wQFMAMBAQAwCgYIKoZI
48 zj0EAwIDRwAwRAIgdacrnDq5f5y3xZ5t+XNAKje+k9lB3JRWcSqxvLbeBpQCID5Y
49 7xSFIhT0pJs4j2UHX6toSEjcXvOjAptBWvYFUVaS
50 -----END CERTIFICATE-----
```

Listing C.6: *MASA EE certificate.txt*

D Images

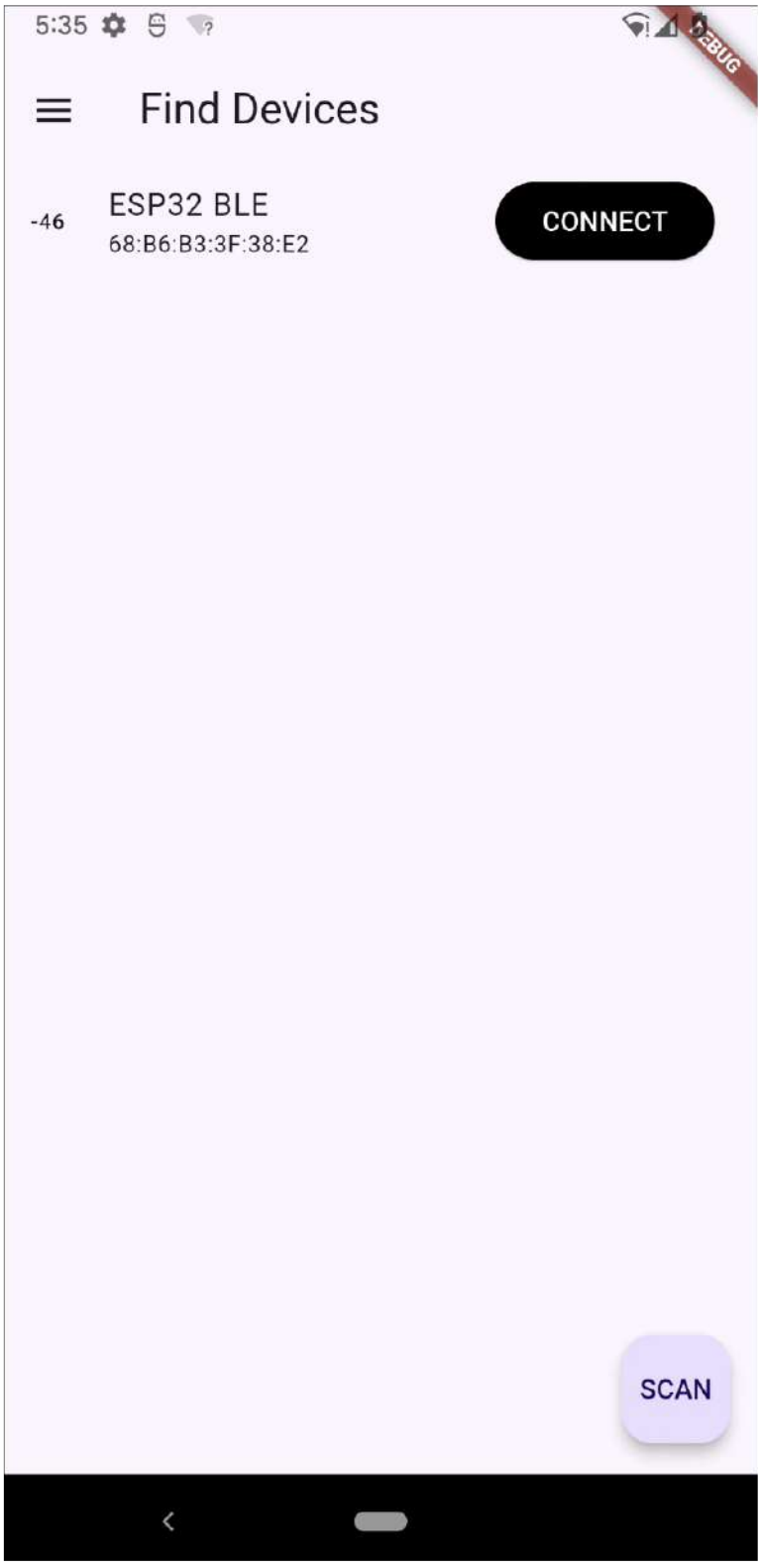

Figure D.1: *The Flutter Application showing the scan for devices screen*

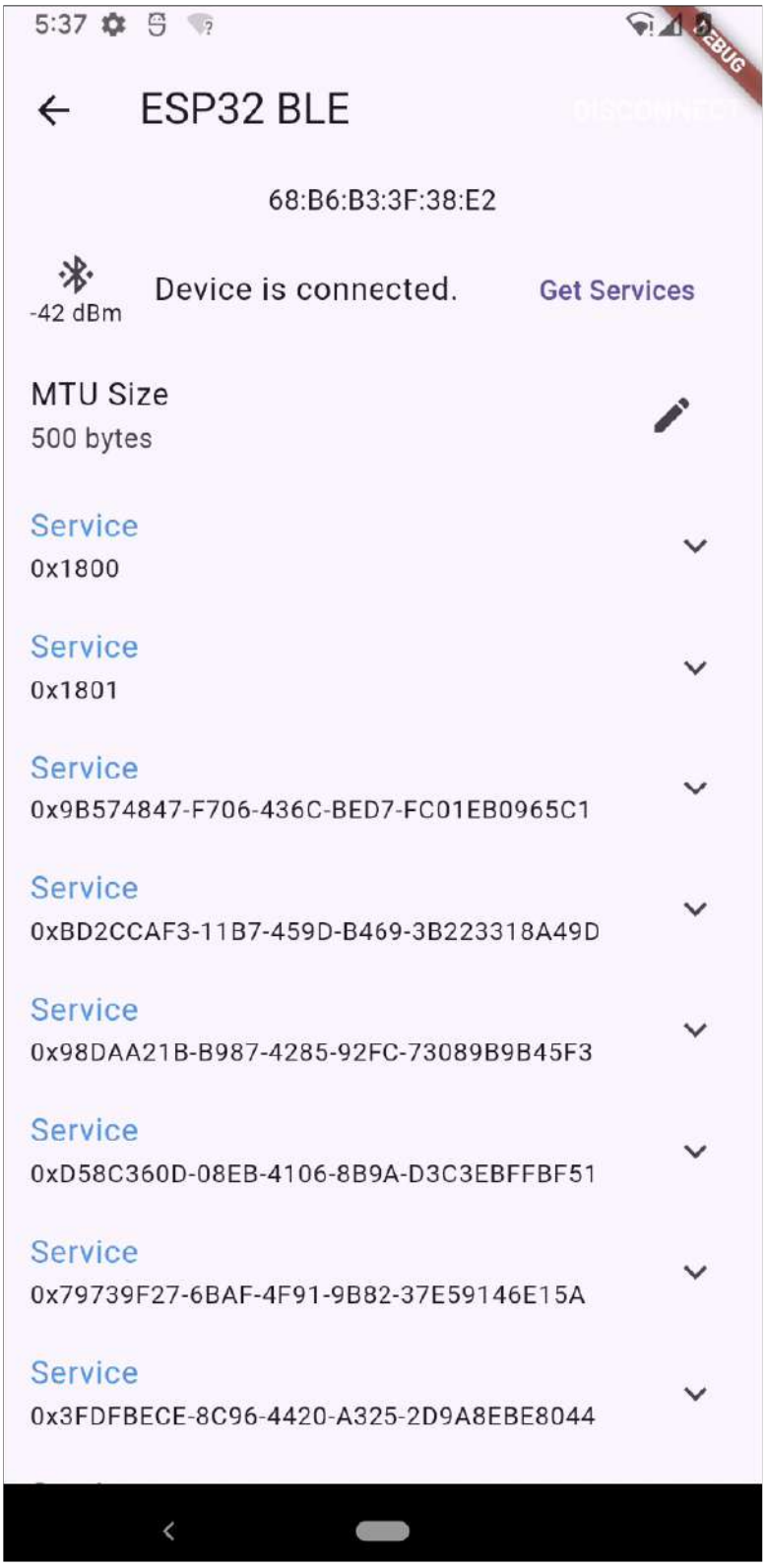

Figure D.2: *The Flutter Application showing a debug panel with device characteristics*