Secure Zero-Touch Bootstrapping of IoT devices using Blockchain Technology

Ushana Bandyopadhyay

Master of Science Thesis
31 July 2023

Supervisors:
University of Turku: Prof. Ion Petre (PhD)
Huawei Technologies: Sandeep Tamrakar (DSc. Tech), Philip Ginzboorg (DSc. Tech)

DEPARTMENT OF MATHEMATICS AND STATISTICS

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Mathematics and Statistics

USHANA BANDYOPADHYAY: Bootstrapping of IoT devices using distributed ledger network
Master of Science Thesis, 59 pages
Mathematics
July 2023

The Internet of Things (IoT) is a rapidly growing field that has found applications in various areas. Before an IoT device can be used, it needs to be bootstrapped. Bootstrapping configures the device with the credentials to connect to its user's network and communicate with other devices on the same network. Additionally, bootstrapping is an essential foundation for the security of the IoT network and its operation. Therefore, ensuring that the correct device connects to the correct network is crucial during bootstrapping. However, the bootstrapping process should also be user-friendly to promote widespread adoption and reduce the risk of errors caused during manual user bootstrapping.

Several companies have introduced Zero-Touch Bootstrapping (ZTB) mechanisms to address these requirements. In ZTB, all the user needs to do is power on their newly bought IoT device, which automatically gets bootstrapped to their IoT network. However, the currently available ZTB mechanisms do not protect against misbinding. Misbinding occurs when a device connects to an unintended endpoint, potentially leading to malicious attacks.

In this thesis, we propose a novel approach to ZTB where we leverage a blockchain-based network infrastructure to enhance its security. Our approach creates a unique bootstrapping network per device per user. The parameters for creating such a bootstrapping network are cryptographically shared between the device and its legitimate user.

In summary, our approach securely transfers bootstrapping parameters, immutably records ownership transfers on the blockchain, and enables zero-touch bootstrapping based on verified credentials.

Keywords: IoT, security, misbinding, bootstrapping, blockchain, distributed ledger technology.

Acknowledgements

I wish to thank my supervisors at Huawei Technologies Oy, Sandeep Tamrakar (DSc.) and Philip Ginzboorg (DSc.), my supervisor from the University of Turku, Prof. Ion Petre (PhD.), and my colleague at Huawei Technologies Oy, Sampo Sovio.

Their valuable guidance and constructive critiques elevated my understanding of the subject matter and helped me write this thesis with utmost precision and depth. I contributed to an ongoing area of research by implementing a prototype design that could be used in the future in the area of IoT device security and management.

Terminology

Asymmetric-Key Encryption: An encryption scheme in which there are encryption/decryption transformation sets, $(E_e,\ D_d)$, which represent the public keys of two communicating parties with e and d as private keys. The property of asymmetric or public key cryptography is that if E_e is known, it is computationally infeasible to find a message m from a random ciphertext, c such that $E_e(m) = c$. [64]

Blockchain: A distributed ledger technology across a network of nodes that records transactions taking place on it in the form of a sequence of blocks. These transactions are recorded when a smart contract is executed and are available to all the nodes in encrypted form. The authenticity of each transaction, however, can be verified by the parties who took part in that transaction. The blocks are validated and ordered by the nodes on the network on the basis of a pre-configured consensus algorithm before they are appended to the chain. [47]

Bootstrapping: In IoT, for a device to be operational in a network, i.e., under its owner's control, it needs to be bootstrapped. The device is initially configured with generic factory settings. After bootstrapping, the device is provisioned with credentials (identities, addresses, security parameters, etc.) that allow the device to be accessible by other devices in that network.[77]

Digital Signature: It is a tag that is generated by transforming a message and some secret information held by an entity. The digital signature binds the said entity's identity to the message in order to ensure authenticity, authority and non-repudiation of that message. [64]

Distributed Ledger: An immutable record of transactions with a local copy of itself present and regularly updated on every participating device in the network (node). This prevents duplication of transaction records which avoids unnecessary effort and inconsistency often faced in traditional business ledgers. [47]

Galois Field (GF): Also known as finite field, is a field over the curve E used in Elliptic Curve Cryptography and is defined by the Weierstrass equation, $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, where $a_1, a_2, a_3, a_4, a_6 \in GF$ and $\Delta \neq 0$, where Δ is the determinant of E. [63]

Hash Function: Hash functions take a message as input and produce an output referred to as a hash or a hash digest. The function h maps bit-strings of arbitrary finite length to strings of fixed length, say, n bits. [64]

Hyperledger: A blockchain-based technology that is a part of the Hyperledger foundation which is an open-source community that is focused on developing frameworks, tools and libraries that work with the enterprise-grade Hyperledger blockchain. Unlike the Bitcoin and Ethereum blockchains, Hyperledger does not have an inbuilt currency and is mainly used for B2B implementations. [40]

Internet of Things (IoT): A collective network of interconnected physical devices (home appliances, vehicles, etc.) with embedded electronics that can connect and communicate with each other across a shared network. It works through real-time collection and exchange of data [58].

Message Authentication Code (MAC): is a hash function that takes two functionally distinct inputs and computes an n-bit output [64].

Man-in-the-middle attack: is a form of cyber attack where the attacker secretly intercepts and possibly alters the communication between two parties who believe they are directly communicating with each other. [80]

Nonce: A time-varying and almost non-repeating value that is freshly generated for each use such as a timestamp, a sequence number, or some combination of these. [11]

Replay Attack: An attack in which the attacker records and replays a portion of or an entire communication session between two parties at some later point in time. [64]

Root Of Trust: A reliable hardware module that includes firmware and software components that perform specific, critical security functions such as generating and verifying digital signatures, encrypting and decrypting data, etc. They are secure by design and inherently trusted. [70]

Smart Contract: A self-executing contract with the terms of the agreement written in the form of lines of code. They are implemented on blockchain platforms in order to enforce the rules and conditions of the agreement. They enable trusted transactions and agreements to be carried out between parties without the need for intermediaries while ensuring transparency, immutability, and tamper resistance. [6]

Symmetric-Key Encryption: An encryption scheme with the encryption/decryption key-pair, (e, d), is said to be symmetric-key if it is "computationally easy" to determine e from d and vice versa. In most of these schemes, $e = d$ and, therefore, they are called symmetric keys. [64]

Telemetry Data: Data collected as a result of automated communications processes by which measurements made by IoT devices (or any device with sensors) are transmitted to receiving equipment for monitoring, display, and recording. [81]

Trusted Execution Environment (TEE): A hardware-assisted, secure execution environment (e.g., Intel SGX and ARM TrustZone) that guarantees isolated execution of authorized security software, called Trusted Applications (TAs), loaded in it. Some useful features of a TEE include trusted storage of data, cryptographic functions such as random number generation, and isolated execution of trusted code. GlobalPlatform is a technical standards organization that develops and maintains specific standards for TEE technology. [52]

Trusted Platform Module (TPM): A computer chip or a microcontroller device that that is embedded in a platform (such as a laptop or an IoT device) that can securely store artifacts (such as encryption keys and certificates) which are used to authenticate the said platform. [83]

Zero Touch Bootstrapping: Also referred to as Secure Zero-Touch Provisioning (SZTP), is a bootstrapping strategy where a network-connected device (such as an IoT device) can be securely provisioned with its new credentials with no input from its user except physically placing and powering it on [87]. An example where this strategy has been implemented is Zero-Touch Setup mode in Amazon's Frustration-Free Setup (FFS) [5].

List of Abbreviations

AES	Advanced Encryption Standard
AKE	Authenticated Key Exchange
AP	Access Point
API	Application Programming Interface
AWS	Amazon Web Services
BADH	Badly Authenticated Diffie-Hellman
BCS	Barcode Setup
CA	Certifying Authority
CLI	Command Line Interface
DH	Diffie-Hellman
DHKE	Diffie-Hellman Key Exchange
DID	Decentralized Identifier
DLT	Distributed Ledger Technology
DPP	Device Provisioning Protocol
DPS	Device Provisioning Service
DSS	Device Setup Service
EAP	Extensible Authentication Protocol
EBSI	European Blockchain Services Infrastructure
ECC	Elliptic Curve Cryptography
ECDHE	Elliptic Curve Diffie Hellman (Ephemeral)
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
EVM	Ethereum Virtual Machine
GCM	Galois/Counter mode
GCP	Google Cloud Platform
GF	Galois Field
HSSL	Helsinki System Security Lab
IETF	Internet Engineering Task Force
IoT	Internet of Things
KA	Key Agreement
KDF	Key Derivation Function
KE	Key Exchange
MAC	Message Authentication Code
OOB	Out of Band
OS	Operating System
PBFT	Practical Byzantine Fault Tolerance
PoA	Proof of Authority
PoET	Proof of Elapsed Time
PoS	Proof of Stake
PoW	Proof of Work
RoT	Root of Trust
RSA	Rivest-Shamir-Adleman
SCITT	Supply Chain Integrity, Transparency and Trust
SDK	Software Development Kit

SDO	Secure Device Onboard
SGX	Software Guard Extensions
SIGMA	Sign-n-MAC
SSID	Service Set Identifier
STS	Station-To-Station
TA	Trusted Application
TEE	Trusted Execution Environment
TLS	Transport Security Layer
TPM	Trusted Platform Module
UI	User Interface
VC	Verifiable Credentials
VPC	Virtual Private Cloud
YAC	Yet Another Consensus
ZT	Zero-Touch
ZTB	Zero-Touch Bootstrapping
ZTP	Zero-Touch Provisioning
ZTS	Zero-Touch Setup

Contents

1 Introduction

IoT is a growing field of technology that has become quite ubiquitous. It has found applications in various areas, including homes, healthcare, transportation, waste management, and retail. For large companies, IoT in their infrastructure has made it more efficient by speeding up processes, reducing error, preventing theft, and incorporating complex and flexible organizational systems [60]. According to a report published by Statista, over 13 billion IoT devices were installed globally as of 2022, and this number is projected to reach 29 billion by 2030 [85].

An IoT device needs to be "bootstrapped" to become operational, i.e., under its owner's control. The bootstrapping process configures the IoT device with the credentials necessary to connect to its user's network and communicate with other devices connected to that network [77]. At the bootstrapping stage, it is prudent to ensure that a device connects to its intended network. Otherwise, an insecurely bootstrapped device is left vulnerable to security breaches. Nevertheless, it is also essential that a high-security implementation in IoT device bootstrapping does not require complicated user interaction, consequently curbing widespread adoption of it [71].

To meet both of these above criteria – making the bootstrapping process more user-friendly while also combatting the security risks associated with it – several companies have come up with Zero-Touch Bootstrapping/Zero-Touch Provisioning (ZTB/ZTP) mechanisms for their IoT devices. Some of these are Amazon's Frustration-Free Setup (FFS) [5] and Microsoft Azure IoT Hub Device Provisioning Service (DPS) [65]. ZTB solutions usually rely on cloud services. For example, Amazon's FFS uses a Device Setup Service (DSS). The DSS is a cloud service responsible for automatically provisioning an IoT device with credentials necessary for it to be bootstrapped to its user's WiFi network.

Making this process "zero-touch" (fully automatic) not only aids in a better user experience but also reduces the chances of security breaches caused by human error. However, ZTB does not protect against misbinding, where a device connects to an endpoint different from the originally intended one. Misbinding could happen accidentally, such as a person's Amazon Smart TV connecting to their neighbour's Amazon Echo. However, it could also be done intentionally by an attacker for malicious reasons, such as a man-in-the-middle attack, which could cause substantial damage in a network where many IoT devices containing sensitive information are involved, such as in hospital infrastructure. Furthermore, presently available ZTB mechanisms are vendor-specific. In other words, ZTB requires that both the IoT and the bootstrapping devices are manufactured or certified by the same ZTB solution provider. For example, an Amazon IoT device can only be ZT-bootstrapped by a device that has FFS capabilities, such as an Amazon Echo.

To counter these limitations, we would employ device-specific, single-use bootstrapping networks meant to be known only by the device and its intended bootstrapping device belonging to its user. In order to implement this, we will be using blockchain technology. The unique identities of the device and its intended user will be verified using the blockchain during ZTB. Encrypted transfer of the device's parameters occurs with every ownership transfer from its point of manufacture to its next owner. Each of these transfers is recorded on the blockchain, which can later be used for verification during bootstrapping, as mentioned above.

We used the Hyperledger Blockchain to replace the centralized cloud service where the

IoT device usually records its data. Blockchain technology uses an immutable distributed ledger maintained and validated cryptographically by a distributed network of nodes. The records on a blockchain are audited and synchronized by these nodes utilizing a consensus protocol. A local copy of the ledger exists on every node, which makes it computationally infeasible to tamper with [47].

This thesis was written during an internship at the Helsinki System Security Lab (HSSL) of Huawei, Finland, from 6th February 2023 to 31st July 2023. It is part of ongoing research in the area of IoT device management. The three theses that precede the work done in this thesis are:

1. Bourdoucen's thesis [14] in securing communication channels in IoT in a home WiFi network using an Android smartphone while also maintaining/improving user experience;

2. Akgün's thesis [3] in the area of secure device pairing and lifecycle management of IoT devices using the LwM2M protocol and;

3. Boire's thesis [13] in the area of authentication protocols, where a new Extensible Authentication Protocol (EAP) was proposed for bootstrapping a group of IoT devices in an extensive campus WiFi network.

We extended the scope of these previous works by designing a solution that uses a blockchain-based network infrastructure using Hyperledger Sawtooth to implement the following features:

1. Secure transfer of bootstrapping parameters during change of ownership of an IoT device;

2. Creation of a secure bootstrapping network connection between a particular device and its intended bootstrapping device using the uniquely generated and securely transferred parameters and authentication from the blockchain;

3. Authenticated and immutable recording of each transfer of ownership on the blockchain using a smart contract.

The proposed solution enhances the security of zero-touch bootstrapping IoT devices. HSSL has previously filed a patent for the solution, and in this thesis, we designed the prototype for it and demonstrated the efficacy of the security measures.

2 Structure of the Thesis

- **Chapter 3: Motivation**

  In this chapter, we have described the goals of this thesis and our motivations behind achieving these goals. We started by stating why automated bootstrapping is more secure than manual bootstrapping and then proceeded to point out a security flaw in ZTB – misbinding – that is the main issue that we have aimed to resolve in this thesis. Five research questions have been formulated from the goals. We have also given example usage scenarios to give a general idea to the readers about how our implementation is expected to work in real life. Finally, we have discussed the novelty behind this research and implementation since it is based on a patent filed by HSSL of Huawei Technologies Oy., Finland.

- **Chapter 4: Background**

  This chapter starts with the work that has already been done or is ongoing in IoT, Blockchains and ZTB. We explain the main issue we are trying to resolve in this thesis, "misbinding" – the different types of misbinding and the ones that pose a threat in currently available ZTB solutions. Then, we briefly introduce blockchain technology and describe why Hyperledger Sawtooth ended up being our chosen framework for our project's implementation. After this, we describe Sawtooth's architecture and explain mathematically how consensus in blockchains works and then compare the consensus protocols used in Hyperledger Sawtooth.

- **Chapter 5: Design**

  In this chapter, we have described the architecture of our implementation, starting from a high-level view down to the intricate technical details and the two design options for implementing the concept. Following this, we have explained the cryptographic primitives behind our implementation and how it ensures that our design is indeed secure.

- **Chapter 6: Implementation**

  This chapter is centred around our prototype and discusses the code that makes it work. We based our code on a pre-existing GitHub repository, the workings of which are discussed first. After this, the additions and alterations made to the original code to make it fit our design are discussed.

- **Chapter 7: Discussion**

  This is the final chapter that answers the research questions that were stated in the Motivation chapter. We have also discussed some drawbacks of our solution and where it needs further improvement. Furthermore, we have described the challenges faced while trying to build our prototype and the constraints of the prototype. Lastly, we have mentioned some areas where there is scope for future work and given a conclusion of what we have learned from this thesis.

3 Motivation

We have established that the use of IoT is prevalent in almost every industry in today's world [60] [85]. It also includes mission-critical IoT used in healthcare, transportation, power supply and similar industries. In these areas, the secure execution of IoT applications is crucial not only for the functioning of an organization but also for the safety of human lives. If a safety-critical IoT network is compromised, it could potentially lead to disastrous consequences. For example, the Triton Malware attacked the network-connected safety instrumented systems of a petrochemical plant in Saudi Arabia in 2017 [62]. The code written in the malware allowed it to access the systems remotely. It could have led to the release of toxic hydrogen sulphide gas, which would have put several human lives at risk at the plant and the surrounding areas. It was concluded that the attackers had intercepted an employee's login credentials who had access to the network, which, in turn, gave them unauthorized access.

This incident also serves as an example of human error in credential management. As the size of IoT networks keeps growing in infrastructures, manually setting up every single device on the network gives rise to similar vulnerabilities caused by human error. Furthermore, this is a tedious process that needs to be more scalable. Manual bootstrapping may also require manually updating the device firmware, which, if not done regularly, could bereave the IoT devices of much-needed security patches in their software [74].

In order to prevent such compromises from occurring in the future, it is vital to ensure that an IoT network, as a whole, and the communication channels between devices in that network are always secure and checked from time to time for vulnerabilities. Since an IoT device becomes operational only after its bootstrapping, it is natural that securing this first step is of utmost importance. On top of all this, it is also evident that the bootstrapping procedure should be as automated as possible for better security, scalability, and usability.

3.1 Our goals

We have already given a summary of Zero-Touch Bootstrapping (ZTB) as a solution for the automated bootstrapping of IoT devices. We have also mentioned how the problem of misbinding persists in the ZTB solutions available in the market today. Misbinding occurs when the IoT device gets bootstrapped to a different network by connecting to the wrong bootstrapping device. Sometimes this can happen accidentally, like a user's Amazon IoT device connecting to their neighbour's Alexa-enabled smart speaker such as an Amazon Echo. However, misbinding can sometimes be done intentionally by any unauthorized person through a man-in-the-middle attack. Upon successful execution of such an attack, the attacker can gain unauthorized access to the network and control and access data on the connected devices. Current ZTB methods usually use predefined and generic parameters for creating bootstrapping networks. The IoT devices are registered on and verified by a cloud server. An attacker can easily intercept and connect to any of these bootstrapping networks or even broadcast similar networks to which any IoT device can connect. We discuss this in detail in Section 4.3.3.

In this thesis, we, first, aim to implement ZTB more securely to reduce the chances of misbinding. Furthermore, we aim to extend this solution such that our implementation does not have to be vendor-specific, unlike the presently available ZTB solutions.

In order to achieve the first goal, we intend to create unique single-use bootstrapping networks, each specific to an IoT device and its intended owner's bootstrapping device. Next, we must ensure that the bootstrapping process continues to be zero-touch regardless of the safety measures we implement. The user-friendliness of ZTB is essential for security for two reasons:

1. So that mass adoption of the solution is not curtailed and security is available to all. Even technical laypeople can use the solution and would not be discouraged by the complicated procedures required to set up the device. In other words, to prevent security from being compromised for usability;

2. To reduce vulnerabilities caused by human errors.

Therefore, minimizing the responsibility of the users in the bootstrapping process also minimizes the number of potential security vulnerabilities.

Our approach to achieving these goals is by recording the entire supply chain life-cycle of the device from the point of manufacture to every time it gets a new owner. These records help verify the device and the endpoints it wishes to connect to whenever it attempts to get bootstrapped. However, these records must be anonymous and encrypted so that no information about the device or the owner is ever breached. The solution to this is to use a blockchain that uses Distributed Ledger Technology (DLT) to record immutable yet anonymous and encrypted transactions. Furthermore, each record is verified by multiple participating nodes on the blockchain of its validity before getting recorded on the blockchain. Every transfer of the IoT device across the supply chain is recorded as a transaction on the blockchain and helps keep track of the latest owner of the IoT device.

During the supply-chain transfers, we also transfer the parameters required to bootstrap the device in a cryptographically secure way so that any third party cannot intercept them. The bootstrapping parameters are encrypted and signed by the current valid owner of the device and transferred to the next valid owner. Both the recording of the transfer of ownership of the device and the transfer of its bootstrapping parameters is done via a smart contract deployed on the blockchain.

Finally, the records stored on the blockchain can be used by devices regardless of the manufacturer and how the bootstrapping parameters are generated. We aim to implement a universal smart contract logic for this purpose.

3.2 Research Questions

The above defining goals of our thesis have helped us narrow down our research questions which are the following:

Q1. *How do we create a device-and-user-specific unique network for bootstrapping to prevent misbinding?*

Q2. *How do we share parameters for this device-and-user-specific bootstrapping network between the IoT and bootstrapping devices?*

Q3. *How do the endpoints (IoT device and bootstrapping device) mutually authenticate each other before the bootstrapping of the IoT device can begin?*

Q4. *Can the bootstrapping process continue to be zero-touch even after applying all these extra layers of security?*

Q5. *Can this process be applied to any IoT device regardless of its manufacturer?*

3.3 Example Usage Scenarios

Our implementation focuses on an automated bootstrapping procedure called zero-touch bootstrapping – the user only needs to power on their IoT device in the presence of their bootstrapping device, and it automatically gets bootstrapped to the user's network. In the scenarios described below, we aim to show how our implementation is supposed to work.

Scenario 1: Registering oneself on WebStore and adding an IoT device.

IoT devices can be registered, bought and sold on WebStore. WebStore is a user interface that can send data to and retrieve data from the blockchain. WebStore can be accessed either from its website or from its mobile application. WebStore has a unique username and a unique device name, respectively, for each user, and each IoT device registered on it. Their real identities are never revealed. A local list of all IoT devices being sold is available to every user on WebStore.

A user with the username Bob orders a device named device1 from the WebStore mobile app on his phone. The device, device1, is now displayed as "reserved" on WebStore. After Bob picks up the device or has it physically delivered to his home, Bob has to click on a "device received" button on his app. This action updates the blockchain with device1 being removed from the market and Bob as its new owner. After this, Bob has to power on and place device1 at a suitable location near a bootstrapping device, say his Amazon Echo smart speaker. Now, device1 automatically gets connected to Bob's home network and can be controlled by his Amazon Echo. Bob can also set a nickname for his device, e.g., "Bob's Smart Bulb". The process of ZT-Bootstrapping device1 will not work with any other bootstrapping device belonging to anyone else.

Scenario 2: Ownership transfer of an IoT Device.

Bob no longer wishes to use device1 and has now decided to put it on sale. He clicks on a "sell device" button, which updates WebStore's blockchain and device1 can now be viewed on the list of devices that are currently being sold.

A user with the username Alice wishes to buy device1. She orders device1 on WebStore's website since she does not wish to use the mobile app. Bob then accepts the order, which leads to device1 now being displayed as "reserved" on WebStore. Once Bob delivers device1 to Alice, she must click on the website's "device received" button.

This action updates the blockchain with device1 being removed from the market and Alice as its new owner. After this, Alice has to power on and place device1 at a suitable location in her home near a bootstrapping device, say her Google Nest Audio smart speaker. Now, device1 automatically gets connected to Alice's home network and can be controlled by her Google Nest Audio. Alice can also set a nickname for her device, e.g., "Alice's Smart Bulb". The process of ZT-Bootstrapping device1 will not work with any other bootstrapping device belonging to anyone else.

3.4 The Novelty Behind our Work

Several projects (discussed in Section 4.1), including those in IoT, have employed blockchains for supply-chain tracking and have mentioned or implemented certain use cases for IoT security and attestation using blockchains. However, as per our knowledge, relevant research or projects have yet to directly implement ZTB of IoT devices using blockchain technology as of today.

We aim to ensure that even if ZTB is performed on an IoT device, it connects to the correct network every time, and there are almost no chances of misbinding. We have, thus, explored and attempted a way to generate unique and secure single-use bootstrapping networks per-user-per-IoT device. The parameters specific to a bootstrapping network are shared between the device and its owner. The network parameters change with every change of ownership of the IoT device, which is recorded every time on the blockchain.

To further reduce the chances of misbinding, we want not only the bootstrapping device to be able to authenticate the IoT device but also for the IoT device to authenticate the bootstrapping device. This requirement implies that we find a way to perform secure and confidential delivery of the bootstrapping parameters with every transfer of ownership of the device along its lifecycle. The device's new owner is the only one with access to these parameters through their bootstrapping device, allowing it to authenticate the IoT device. The IoT device's ownership status is verified from the transaction records on the blockchain to generate attestation evidence required for its bootstrapping.

While several novel approaches have been undertaken to handle secure zero-touch bootstrapping of IoT devices, our approach is independent of vendor-specific IoT bootstrapping protocols. Since we use Hyperledger Sawtooth transaction processors to define the business logic, the code can be customised to implement any authentication protocol required to initiate ZTB (discussed in Section 4.5.1). Our implementation also focuses on creating a unique bootstrapping network per-device-per-user. This implies that any bootstrapping mechanism can be implemented once a secure connection between the two endpoints has been established.

Furthermore, IoT devices that can be bootstrapped using the ZTB method are also usually vendor-specific, implying that only the devices manufactured or certified by the ZTB solution provider (such as Amazon) can be bootstrapped in this manner. In our implementation, any IoT device can automatically establish a secure connection with its intended bootstrapping device regardless of each of its vendors. This follows from the previously mentioned approach of leveraging blockchain technology and it allows small businesses or private entities to apply ZTB on their devices.

Finally, our implementation allows ZTB of a device even after a direct change of its ownership. If person A sells a bootstrapped IoT device to person B, person B can directly apply ZTB to the device in their network. This cannot be done with the ZTB methods available today. For example, Amazon devices can only implement ZTB if they are bought from the official website by the owner rather than if it has been gifted or bought from a third party.

4 Background

4.1 Related Work

We have researched works that have (i) analyzed the possible attacks in zero-touch (ZT) or other automated bootstrapping procedures, (ii) integrated blockchain into IoT, and (iii) implemented automatic bootstrapping.

First, we explored the paper "Misbinding Attacks on Secure Device Pairing and Bootstrapping" [78]. It included identity misbinding in cryptographic protocols, misbinding in cuckoo attacks during device pairing, and misbinding during bootstrapping over a WiFi network or an Out-of-Band (OOB) channel. In their paper [78], Sethi et al. suggest "asset tracking" as a solution for mitigating these kinds of attacks. This is similar to our approach. Although Sethi et al. recommended using a cloud database to implement asset tracking, we are extending the security by using the decentralized architecture of a blockchain.

Several ongoing open-source initiatives have developed solutions that leverage blockchain based systems. The Trillian project [82] is a tamper-evident logging system. It uses Certificate Transparency [20], a blockchain-like Merkle Tree that behaves as an ecosystem for the issuance of transparent and verifiable certificates. The latter replaces the need for a certifying authority (CA) – a trusted third party that issues digital certificates to bind a public key to an entity. ZTB relies heavily on such certificates for IoT device verification and attestation. Over 9.9 billion certificates have been issued using the certificate transparency ecosystem, proving that DLT is gaining traction as a reliable tool to enhance security in a setting with several participating entities, much like an IoT network.

Internet Engineering Task Force's (IETF) Supply Chain Integrity, Transparency and Trust (SCITT) working group is focused on IoT device supply chain tracking using blockchain technology [35]. They standardize protocols, interfaces, and data models for tracking IoT devices across the supply chain. Their approach utilizes Verifiable Credentials (VCs), Decentralized Identifiers (DIDs), and also Certificate Transparency.

The European Blockchain Services Infrastructure (EBSI) provides a blockchain platform that hosts a distributed ledger (DL) across a pan-European network of nodes. EBSI provides Application Programming Interfaces (APIs) for developers to leverage the EBSI blockchain for their applications. These APIs can be used inside smart contracts written on other blockchain frameworks. So far, EBSI only has an API for the implementation of verifiable credentials, which does not come into the scope of our project. The API for supply chain tracking is yet to be released. In the future, EBSI could potentially be used as a standard blockchain framework if our project were to be adopted and implemented across Europe. [88]

In the paper, "Distributed IoT Attestation via Blockchain" [54], Jenkins et al. have introduced a concept called "Distributed Attestation Network (DAN)" that uses blockchain technology as a distributed storage system as well as to implement an attestation protocol utilizing this stored information. The attestation is done by gathering verifiable evidence about an IoT device from the information about the state or properties of that device that is shared across all the nodes on that network. While our project primarily focuses on creating unique bootstrapping networks, this attestation protocol can be used in conjunction to complete the bootstrapping process.

Trustonic has developed the concept of "digital holograms" for IoT devices [84]. These

days, most IoT devices are identified by an inbuilt Root of Trust (RoT) that is embedded in its hardware at the point of manufacture (mentioned in section 4.2.1). However, from its point of manufacture, the IoT device goes through several other intermediaries that configure the device with software and hardware modules. It is vital to ensure that genuine entities have performed these configurations for security purposes. A digital hologram is an authentication, signed by the RoT, that confirms that an IoT device has been modified by a valid entity. Each digital hologram is generated by the device's Trusted Execution Environment (TEE) and chained together in a block. This block and the RoT are then verified by Trustonic's Virtual Private Cloud (VPC) before it is provisioned or bootstrapped. However, this mode of verification is also vendor-specific.

In the article "Zero-Touch Bootstrap of a Network Connected Device" [33], a solution is proposed for the challenges faced in WiFi Alliance's Device Provisioning Protocol (DPP). The methodology for bootstrapping using DPP is closely related to Microsoft Azure's Device Provisioning Service (DPS) as described in Section 4.2.2. Our implementation is closely related to the solution that is described in the article mentioned above, except, we use a distributed ledger technology, the Hyperledger Blockchain, to make our implementation more secure and vendor-non-specific. In the next section, we give some general insight into the presently available solutions for automatic bootstrapping of IoT devices.

4.2 Currently Available ZTB protocols

For an IoT device to be bootstrapped to a network, it must be configured with custom credentials to establish secure communication channels with other devices on that network. This bootstrapping process can be zero-touch (ZT). In this case, the device owner needs to power on the IoT device within the range of an active bootstrapping device, like an Amazon Echo or a smartphone. The user does not need to input custom credentials manually.

In order to understand how current ZTB protocols function, we studied the mechanisms offered by the companies that have claimed to have employed ZTB functionality in their IoT solutions. The top four companies that have ZT support in their IoT devices are :

- Google - KORE's Zero-Touch Provisioning [56].
- Intel - Secure Device Onboard [50].
- Microsoft - Azure IoT Hub Device Provisioning Service [65].
- Amazon - Frustraton-Free Setup (Zero-touch setup mode) [5].

Google Cloud Platform (GCP) IoT core that offers a cloud suite for IoT solutions is set to be discontinued and migrated to the hands of KORE. Not much information about the mechanism for zero-touch provisioning was found and is therefore not discussed in detail in this thesis. Among the mechanisms that were studied, for Microsoft and Amazon IoT devices, ZT bootstrapping or ZT provisioning is preceded by pre-binding. Pre-binding registers only the IoT devices on the IoT platform's cloud service before they are sold. Any bootstrapping device or interface with ZTB capabilities compatible with that of the given IoT device, can later connect to the device and then configure it with the credentials necessary to bootstrap it to its user's network.

4.2.1 Intel's Secure Device Onboard

In the case of Intel's IoT devices, they use what is called Late Binding [50]. Here similar devices can be manufactured in batches and are later auto-provisioned to any platform that offers IoT services, such as Amazon and Microsoft Azure. Each device is configured with an ownership voucher and a hardware-generated RoT key that is stored in a Trusted Platform Module (TPM) embedded on the device; both of these are used to authenticate the IoT device before binding them. The ownership voucher helps keep track of the device throughout the supply chain. It also decides how the device will be configured based on the requirements of the IoT platform with which it later wishes to operate.

This mechanism, however, is not precisely ZTB. However, the method of automatically identifying and provisioning the device (without user intervention) is almost the same as in ZTB, as described in Section 4.2.3.

4.2.2 Microsoft Azure's IoT Hub Device Provisioning Service

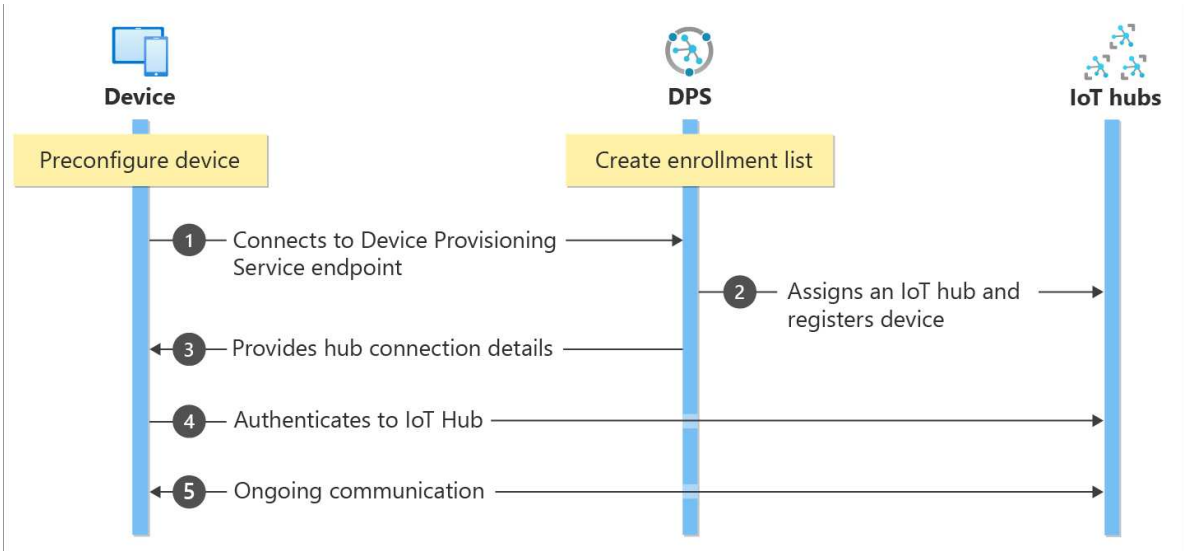


Figure 1: Architecture of Microsoft Azure's IoT Hub Device Provisioning Service [65]

Microsoft Azure's IoT Hub is an integrated public cloud service set including a Device Provisioning Service (DPS) [65]. It is the DPS that helps the IoT hub in making ZTP possible in Azure IoT devices. The steps for this workflow are given below and follow the workflow shown in Figure 1:

1. Before device provisioning can begin, the manufacturer manually configures the device with its authentication credentials, DPS ID, and endpoint.

2. The manufacturer then prepares a DPS instance with individual enrollments and enrollment groups that can identify valid devices and decide how they should be provisioned. These two consecutive steps comprise pre-binding.

3. Next, when the device's new user powers it on, it connects to a DPS endpoint with its DPS ID and then sends its authentication credentials to it. This function is demonstrated in step (1) in Figure 1.

4. The DPS instance checks for the device's DPS ID in its enrollment list, and if it is verified, the device is connected to its respective IoT Hub and is registered there. This function is demonstrated in step (2) in Figure 1.

5. The IoT hub then sends the device's personal ID and registration information back to the DPS instance, which, in turn, sends this information back to the device. This function is demonstrated in step (3) in Figure 1.

6. Now, the device can connect directly to the IoT Hub and authenticate itself. This function is demonstrated in step (4) in Figure 1.

7. The DPS instance is no longer required, and the device can directly communicate with the IoT Hub. This function is demonstrated in step (5) in Figure 1.

An Azure IoT device connects to its respective IoT hub at the end of this procedure. It is referred to as "ZT provisioning" in Microsoft's documentation [65]. Just like Intel's SDO, even though ZTP has characteristics similar to ZTB implementation-wise, it differs from ZTB because its goal is to connect an IoT device to the correct IoT hub. The IoT hub is a cloud-hosted backend mainly responsible for providing a secure communication channel for IoT devices to send and receive telemetry data. We shall see how this differs from ZTB by comparing it with Amazon's Frustration Free Setup (FFS) in the next section, Section 4.2.3.

4.2.3 Amazon's Frustration Free Setup

Amazon's FFS has two modes: Zero-Touch Setup (ZTS) and Barcode Setup (BCS) [5]. ZTS is the most relevant example of ZTB for us – a person with an FFS-enabled Access Point (AP) need only buy an Amazon IoT device and turn it on in the presence of this AP. This action automatically configures the device with the credentials to connect to the user's network. Amazon's ZTS consists of the following steps [5]:

1. A user orders an Amazon-certified IoT device from Amazon's official website, amazon.com. Now, the device gets prebound with the user's Amazon account; the device is registered on the Device Setup Service (DSS) and identified by a unique identifier such as a serial number or a certificate.

2. The Amazon account has to have the following prerequisites:

    - The user's home WiFi credentials through Amazon WiFi locker.
    - An Amazon Alexa skill linked to it.
    - "Log in with Amazon" enabled.

3. When the user powers on the device, it broadcasts a "need for setup" message with its identity information.

4. Access points (APs), such as an Alexa or a non-Amazon router with FFS capabilities, continually scan for such broadcasts and, upon detecting one, connect to the device. Then the AP forwards the message to Amazon's DSS. Similar to Azure's DPS, this cloud service aids FFS.

5. The DSS sends a challenge to the device to confirm its identity.

6. The device generates a response by signing the generated challenge with its private key.

7. The DSS confirms the correct response to this challenge by verifying the device's signature with its public key associated with its unique identifier.

8. After the challenge is confirmed, the DSS sends the configuration parameters to the device, including the user's WiFi credentials.

If ZTS fails, the BCS mode can be employed. BCS is mainly used when the device is not prebound with the user's Amazon account. For example, if the user received the device as a gift. In this case, the user needs to scan the barcode on the device with an app on their mobile phone.

If neither mode works, the IoT device can still be bootstrapped. For example, an Amazon Echo smart speaker with Alexa capabilities may still detect an IoT device in its vicinity and connect to it. Alexa can then alert the user, who can manually configure it with their account preferences.

Since ZTS is the most relevant implementation of ZTB in our thesis, we shall, henceforth, be referring to it in most of our examples.

4.3 Misbinding

4.3.1 Identity Misbinding

In a network, it is necessary to have a secure communication channel between two endpoints on that network so that the communication between them cannot be intercepted and/or altered. In a public network, it might be a good idea to use an asymmetric key encryption scheme (such as RSA [2], ElGamal [34], Diffie-Hellman [25], etc.) that involves both parties (endpoints) to have a public-private key pair each that can be used to encrypt and decrypt the messages that they send to each other. Asymmetric cryptography, in general, – not just encryption but also signing and key-exchange protocols (ECDSA, ECDH, etc.) – are useful since their usage simplifies key management.

When n parties use symmetric key encryption, and each pair has its own key, the number of needed keys is the same as the number of pairs: $n \cdot (n-1)/2$. On the other hand, in asymmetric key encryption, n keys are needed for n parties.

However, asymmetric key encryption is computationally expensive and time consuming [21]. This can be a problem for endpoints that wish to have continuous communication across a channel in a network such that in an IoT network, multiple devices need to continuously share data with other devices on that network.

In this case, it is less computationally expensive to create a channel with symmetric key encryption (such as AES [29], ChaCha20 [69], etc.). But to create such a channel, a shared secret key is required that both communicating parties agree upon without it being breached. In order to agree upon this shared secret, these two parties first start with an asymmetric scheme such as the Diffie-Hellman Key Exchange (DHKE), where they use their respective public-private key pairs to generate a shared secret on each side which can then be used to encrypt their secure channel of communication.

However, the basic Diffie-Hellman (DH) protocol does not provide authentication; the communicating peers do not know to whom they establish the shared secret for subsequent communication. This makes the protocol susceptible to misbinding attacks. In simple terms, misbinding occurs when an endpoint connects to another endpoint that is not the one that it had initially intended to connect to. Therefore we need a secure key-exchange algorithm that meets the following three criteria[57]:

- **Authenticity:** ensures that the two parties in the KE are legitimate and an adversary cannot impersonate them or tamper with the key exchange process [86].
- **Secrecy:** the property of keeping the exchanged keys confidential from unauthorized entities [25].
- **Consistency:** the property when two parties establish a common session key (shared secret), they must have a consistent view of who these two parties are in the session.

For the next few attack examples, we will assume the Dolev-Yao adversarial model [26], where the attacker:

- has full control of the communication channel between the two parties in consideration;
- can intercept, delay the delivery of, modify, and inject their own messages into the communication between the two parties;
- can activate either of the two parties to initiate and respond to KE sessions.

Let us call this attacker Eve.

Let us first take an example of an authenticated DH protocol [57].

Let there be two parties, Alice and Bob. In Figure 2, g^a and g^b are called the DH exponentials of Alice and Bob, respectively; a and b represent Alice and Bob's respective private keys. The $SIG_{sigKey}()$ function is used to represent the digital signature of each party under its private signing key, $sigKey$. This signature is used as proof of freshness in order to avoid replay attacks. Freshness refers to the ability of a party to verify that the session key received from another party is new and not replayed from an old session [16]. Both Alice and Bob then end up with the same shared key that is g^{ab}. This shared key, also called the session key, is what both Alice and Bob use to identify the session. Here, the session is defined by each execution of the protocol between two parties.

In this case, there is no way to authenticate that the exchanging parties are indeed the ones that had originally agreed on exchanging the secret in the first place. This is because the pair of keys that is used for signing every time is not bound to the party's identity, and neither party is sure who the owner of the peer public key is. Creating new ephemeral key pairs during each round of protocol execution provides perfect forward secrecy and, thus, does not compromise past or future keys. However, this leaves the possibility of an identity misbinding attack occurring from a third party, Eve. The cause of this is that this protocol in Figure 2 does not satisfy the consistency requirement [57].

As we can see from the protocol above, Eve lets the first two exchanges of messages (in steps 2 and 3) between Alice and Bob go unchanged. Eve then replaces the third message (in step 4) with their own signature. Henceforth, Alice believes that they are sending the

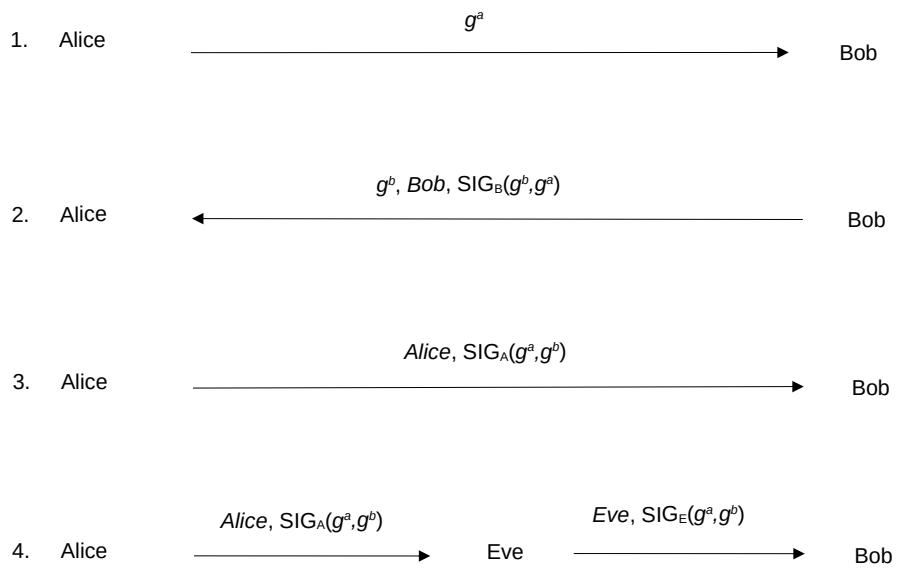


Figure 2: Example of how an identity misbinding attack takes place

next messages to Bob when it is actually Eve who is intercepting the messages. Bob, on the other hand, receives messages signed by Eve, who they believe to be Alice since they have no way of verifying whether the signature is from Alice. This gives the protocol the name, Badly Authenticated Diffie-Hellman (BADH) [57].

In this attack, however, Eve cannot read or alter the message that is being exchanged between Alice and Bob. Therefore, there is no breach of secrecy in this attack. However, this attack can cause other problems such as in the case of a bank transfer. If Bob represents the bank and Alice is trying to make a transaction with Bob, Eve can intercept in between and have Bob approve the transaction such that all the money that Alice wishes to withdraw goes into Eve's account.

This can also be applied to the case of IoT devices which is our use case. Suppose Alice buys an IoT device, say, a smart lock for the main entrance to their house. Alice owns a bootstrapping device (say, Amazon Echo) that can connect to Alice's IoT device for bootstrapping. However, Eve, Alice's neighbour, owns another Echo that connects to Alice's said IoT device. Now Eve gets control over Alice's IoT device and can use it to unlock the main entrance to their house.

Several protocols have been formulated over this to combat identity misbinding, starting from Station-To-Station (STS) protocols to Sign-and-MAC (SIGMA) protocols [57]. STS protocols use digital signatures for authentication, however, they are still susceptible to identity misbinding attacks because of the lack of sufficient binding of the DH key to the signatures.

The SIGMA protocols counter the problem of identity misbinding by including a

digital signing algorithm that can validate the identities of both parties involved in the key exchange. The SIGMA protocols have enough evidence to support that they do not suffer the same misbinding attacks that the STS protocols and the other KE protocols before them, did. We see how the basic SIGMA protocol (without identity protection) works in Figure 3.

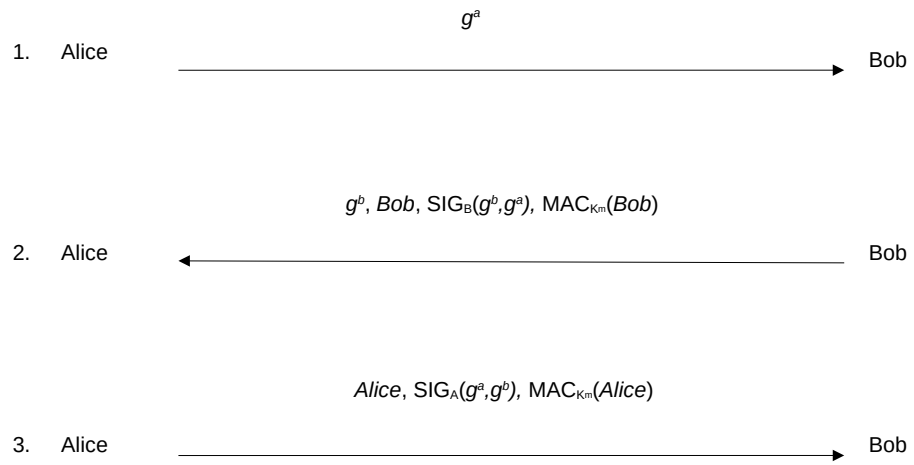


Figure 3: The basic SIGMA protocol [57]

The output is the same as before, a session key, K_s that is derived from the DH value, g^{ab}. K_m is also derived from the DH value but it is essential to note that K_s and K_m are computationally independent or in other words, neither K_s nor K_m can be derived from each other. SIGMA's security is guaranteed independently of how the identity of each party (Alice and Bob) is encrypted. This is why we are not going to go into further detail about the variations of the SIGMA protocol that employs identity protection.

In Figure 3, $Alice$ and Bob represent the identities of Alice and Bob, such as public-key certificates. $MAC_{K_m}(Bob)$ represents a Message Authentication Code (MAC) on Bob's identity using K_m. A MAC is a hash function that takes two functionally distinct inputs, in this case, Bob's identity Bob and K_m, and computes an n-bit output [64].

The DH exponentials chosen by Alice and Bob each are protected from being altered by an attacker by applying their signatures to their respective exponentials. Furthermore, in the ideal case, it is important to note that the DH exponentials, g^a and g^b are freshly chosen for each session. g^a and g^b are deleted as soon as g^{ab} is computed. Another option is to generate fresh nonces for each peer for the signing function. This is done to prevent replay attacks.

Next, applying the MAC function binds the session key, K_s, to the identity of each

peer. This not only acts as proof of possession of K_s but also satisfies the consistency requirement that the BADH protocol failed to.

Modern security protocols include measures against identity misbinding. Let us take as an example TLS 1.2 [24], which is used, among other things, for secure communication in the case of ZTP of IoT devices in Microsoft Azure [65]. Some of the commonly used cipher suites for TLS handshakes are:

- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

Let us take the example of the cipher suite TLS_ECDHE_RSA_WITH_AES_256 _GCM_SHA384 [49]:

- Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) is the protocol used for encrypted key exchange. The term "ephemeral" represents single-use cryptographic key pairs that are generated only during and then deleted after the exchange.
- Rivest-Shamir-Adleman (RSA) scheme is used for digital signatures during peer authentication.
- The AES-256-GCM is the encryption algorithm. The shared secret derived from ECDHE is used to derive the encryption and authentication keys for Advanced Encryption Standard (AES) in Galois/Counter mode (GCM).
- SHA-384 is used to compute hash values of the transmitted data. The hash values are used to verify the integrity of the received data.

To understand the security implications of this, let us consider the simplified model of a TLS handshake in Figure 4.

Alice and Bob use private keys a and b, respectively. These private keys are used to generate the public keys, g^a and g^b. The session key,

$$K_{AB} = MAC_{PMK}(N_A, N_B) \tag{1}$$

where PMK stands for pre-master key or pre-master secret which is calculated as

$$PMK = g^{ab} \tag{2}$$

N_A and N_B are random nonce values chosen by Alice and Bob, respectively. MAC_{PMK} is a MAC algorithm that uses the PMK to create a MAC of the two nonces that were exchanged between Alice and Bob.

$E_B(PMK)$ is the encryption function on the PMK using Bob's public key. The $SIG_A(MessSeq1)$ is Alice's digital signature on the first message sequence which is a hash of all the messages exchanged between Alice and Bob up until then. In this case,

$$MessSeq1 = H(N_A, N_B, E_B(PMK)) \tag{3}$$

where $H()$ is a hashing function.

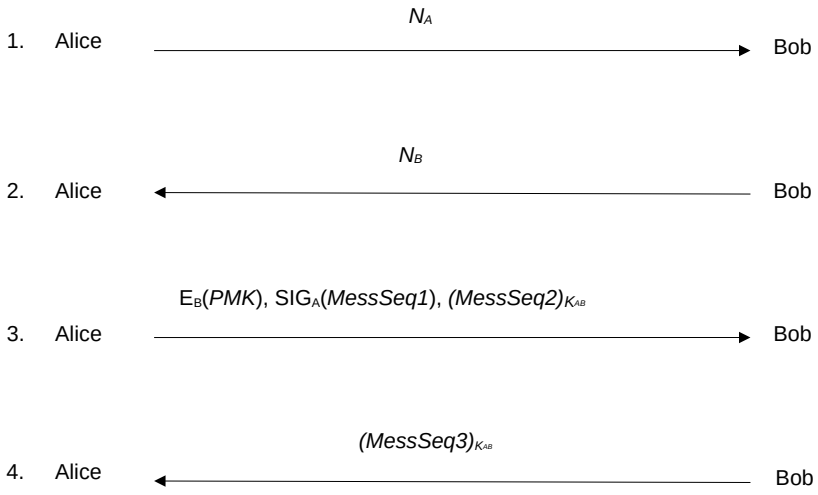


Figure 4: Simplified diagram of a TLS key-exchange protocol [16]

Let us compare this TLS 1.2 suite with the basic SIGMA protocol[57].

$$E_B(PMK), SIG_A(MessSeq1), (MessSeq2)_{K_{AB}} \tag{4}$$

$$g^b, Bob, SIG_b(g^b, g^a), MAC_{K_m}(Alice) \tag{5}$$

Equation 4 in Figure 4 similar to Equation 5 in Figure 3 since we can observe the following:

- $SIG_A(MessSeq1)$ and $SIG_b(g^b, g^a)$ are analogous functions. The digital signatures on the DH exponentials of both parties prevent them from getting altered. The DH exponentials are N_a and N_b in the case of TLS and g^a and g^b in the case of SIGMA. Furthermore, N_a and N_b are nonces, meaning freshness is preserved to prevent replay attacks.

- $(MessSeq2)_{K_{AB}}$ and $MAC_{K_m}(Alice)$ are also analogous. This is not analogous at first, but if you look at the equation on message sequences given in Equation 3, it is a hash function, so $(MessSeq2)_{K_{AB}}$ is essentially a MACing function using K_{AB}. This is where the consistency requirement is satisfied.

Therefore, the TLS handshake includes the sign and MAC measures against identity misbinding that were introduced in the SIGMA protocol.

4.3.2 Cuckoo Attack

Authenticated Key Exchange (AKE) protocols give us a basis for how misbinding attacks occur in a traditional sense. IoT devices communicate with each other over short-range channels such as Bluetooth, WiFi, and Zigbee. Before these channels can be established, the IoT devices require device pairing, which cannot apply AKEs because the devices to be paired do not have prior knowledge of each other's public keys or other identifiers [78].

Many IoT device manufacturers rely on trusted computing for verifying or attesting a device during bootstrapping based on the device's software or hardware components. In this case, the IoT device has a Trusted Platform Module (TPM) that can calculate the cumulative hash of the installed software. In more advanced cases, the devices can also have a Trusted Execution Environment (TEE), such as Intel SGX, which allows otherwise isolated software to be launched only after the device has been bootstrapped. It can also attest the device's integrity to an external verifier [78]. Intel's Secure Device Onboarding (SDO) utilizes this TEE feature to implement their concept of late binding [50].

Trusted computing can, however, be susceptible to a cuckoo attack [78], where a user connects to a "Cuckoo" device in their proximity but is unaware of this. The cuckoo device itself is not a "trusted device" but can forward the communication to another device with the capabilities for trusted attestation. In this case, incorrect endpoints can also end up connecting, making the system vulnerable to misbinding attacks.

This process can be countered by limiting the time taken for request round-trips to determine the proximity of the device being paired with [32] or manual confirmation by the user during bootstrapping. The latter solution cannot be applied to ZTB where the only amount of user interaction is just powering on the device. Furthermore, the attestation in trusted computing takes place only after the two devices connect to each other. The vulnerability that this creates is discussed in the next section, Section 4.3.3.

4.3.3 Misbinding in ZTB

It is evident that modern ZTB protocols, by themselves, are not cryptographically flawed. The problem lies in establishing a secure bootstrapping channel between the IoT and bootstrapping devices.

In Section 4.2.1, it is worth noticing that in the mechanism, the IoT device can connect to any bootstrapping device before it is authenticated and then "onboarded". Furthermore, before getting provisioned, the device cannot check if it is getting "binded" with the correct IoT cloud solution provider. Similarly, in Section 4.2.2, the IoT device can connect to any DPS endpoint and send its authentication credentials without authenticating the DPS endpoint itself. Even here, the IoT device cannot confirm if the DPS endpoint is valid or if the IoT hub it is getting connected to is the right one. These two problems persist in Amazon's ZTS, as described in Section 4.2.3. The "need for setup" message broadcasted by the IoT device is generic, and any device broadcasting such a message can connect to a bootstrapping device with FFS capabilities. The credentials that the bootstrapping device sends to the IoT device are not validated by the IoT device as authentic either. In conclusion, the IoT and bootstrapping devices do not mutually authenticate each other before beginning the ZTB process.

Let us imagine a scenario where a user, Alice, bought an FFS-enabled IoT device from Amazon. It has already been registered on Amazon's DSS. Alice brings the device home

and turns it on, expecting it to be connected to their home network by their Amazon Echo. However, Alice's neighbour Bob also has an Amazon Echo with FFS capabilities installed in his house. Due to unfortunate circumstances, Bob's Echo happened to detect Alice's device first. Now Bob's Echo performs ZTB on Alice's device, and Alice's device gets provisioned with Bob's WiFi credentials and, consequently, ends up connecting to Bob's network. This happens because all FFS-capable devices worldwide access the DSS, and the network broadcasted by an IoT device (or, in some cases, a bootstrapping device) is generic. Furthermore, Alice's device cannot check if the credentials being supplied to it are authentic and will use them regardless. In other words, Alice's specific IoT device ought to have connected to Alice's specific Echo and not Bob's.

In the above scenario, Alice would probably reset the device and start the ZTB process all over again, or, to be extra safe this time, rely on BCS mode and manually scan the QR code on their device with their phone.

Suppose Alice has another neighbour, Eve, who is secretly a malicious hacker. Eve knows that Alice recently had an Amazon smart security camera delivered to their house. Alice powered up the security camera, expecting their Amazon Echo to perform ZTB. However, Eve brought a rogue AP with FFS capabilities and connected to the camera before Alice's Echo could detect it. The AP injected malicious code into the security camera with which Eve got backdoor access. Alice noticed that the security camera was not connected to their WiFi and unsuspectingly resets it, and this time their Amazon Echo successfully performed ZTB on it. However, Alice is unaware of the fact that Eve now has control over her security camera and can gather and manipulate footage recorded by it.

An attack can also take place the other way around. In this case, Eve managed to gain unauthorized access to Alice's Amazon account and its associated IoT devices. Eve configured a rogue IoT device that broadcasts an SSID that Alice's Amazon Echo thinks is from one of these devices. The Echo connects to the rogue device, which injects it with malicious code that gives Eve backdoor access to the Echo. Eve can now track telemetry data from Alice's IoT devices and maybe even access Alice's credentials stored on the Echo.

Thus we can re-iterate that in ZTS, the IoT device connects to a WiFi AP with a pre-defined generic SSID. This SSID is not specific to the device or its user. Attackers can exploit this vulnerability to gain access to telemetry data, the user's WiFi credentials, cameras, smart locks, and the like. Thus, not only do we need to authenticate the IoT device that wishes to be bootstrapped to a network, but we also need to ensure the IoT device can verify the network it is attempting to get bootstrapped to.

4.4 Blockchain Technology

A blockchain is a continuously growing chain of blocks, each containing a set of cryptographically secured transactions. Blockchain technology comprises a network of nodes (or computers) and a digital Distributed Ledger (DL) that records transactions taking place between these nodes. All the nodes possess a copy of the ledger in order to have a synchronized view of it. These records need not be of just financial transactions but also the exchange of other kinds of information. This fosters the network's transparency, immutability, and decentralization [12].

Every blockchain transaction produces a unique hash value that cannot be replicated.

We can leverage this to create unique device-and-its-corresponding-user-specific bootstrapping networks every time the device is to be bootstrapped.

These mechanisms and features are implemented by means of a consensus protocol, a data storage mechanism such as a distributed file system, and smart contracts. [28]

Each of these concepts is discussed in further detail in the following sections.

For the sake of brevity, we will use the term "blockchain" instead of "blockchain technology" in the rest of the thesis.

4.4.1 Types of blockchains

There are multiple blockchain platforms today. Based on public access to the distributed network, blockchains can be divided into the following types [67]:

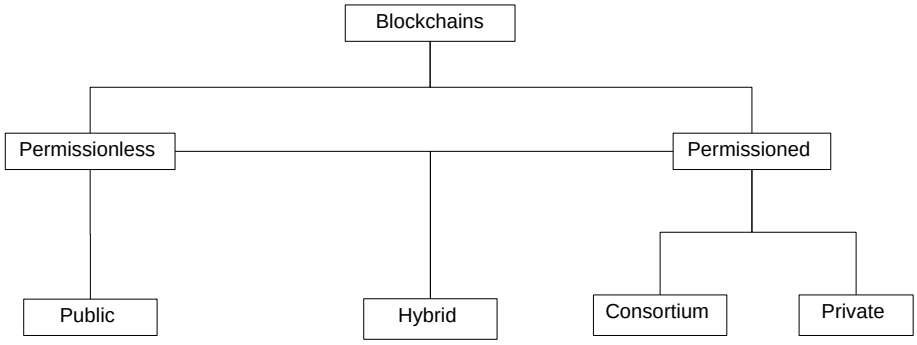


Figure 5: Types of Blockchains

1. **Public Blockchains**, such as Bitcoin and Ethereum, are also called trustless blockchains. These are accessible by anyone and, therefore, are completely decentralized and transparent. No single entity is in control of the entire network.

2. **Private Blockchains** such as Quorum[22] and Corda [73] are typically controlled by a single private enterprise. Unlike public blockchains, only authorized entities are allowed to participate in the network. They are also referred to as enterprise blockchains.

3. **Consortium Blockchains** such as the Marco Polo Network [61] and the Energy Web Foundation [30] are also called federated blockchains and are controlled by a pre-selected group of entities or organizations. These are useful in cases where inter-organizational transactions are involved, such as a country's intercity railway system.

4. **Hybrid Blockchains**, or semi-private blockchains, such as Dragonchain [27] and IBM food trust [48], have features of both private and public blockchains. This means that some portions of the network are private while the rest are public. For

example, the transaction records are not public but any public entity on the network can validate transactions through smart contracts. These kinds of blockchains are, thus, more customizable.

Let us say, for example, that anyone (public) can buy an IoT device and, therefore, can join the network and validate their transactions. But if we want to track the movement of a device across the supply chain, that should only be available to the company that has manufactured the said device. This kind of customisability can only be provided by a hybrid blockchain. We can see that the most suitable type of blockchain for our project would be a hybrid one.

Furthermore, since our project has a strong business involvement i.e., ownership transfer of IoT devices and recording of a device's life-cycle in the supply chain, we could use an enterprise-grade blockchain for our implementation.

A consortium-like blockchain that is controlled by a group of entities – in our case, different IoT device manufacturers – might also be a good choice. However, whether the control should lie only in the hands of the device manufacturers or should also be distributed amongst IoT device users is not looked into and, thus, this aspect has been disregarded here.

4.4.2 Hyperledger Frameworks Currently in Use

We have looked into the Hyperledger blockchain which is enterprise-grade. It also has the most number of projects developed on it and updated and available documentation which makes development work easier. While Hyperledger at its core is a permissioned blockchain, it has different kinds of projects with different levels of permissioning which we will look into in the next section.

The five main projects that have been developed using the Hyperledger blockchain are: [38]

1. **Hyperledger Fabric Project:** The most renowned project that has a modular architecture. Here, consensus and membership functions can be added to applications built on it in plug-n-play style.

2. **Hyperledger Indy Project:** Primarily focuses on decentralized digital identities. It is interoperable across domains, applications, and also other blockchains.

3. **Hyperledger Iroha Project:** It has a simple and modular construction that can easily be incorporated into infrastructure and IoT projects with an emphasis on client application development.

4. **Hyperledger Sawtooth Project:** It is also modular and flexible but is most notable for having separate core system and application domains which makes the writing of smart contracts easier.

5. **Hyperledger Besu Project:** It is an Ethereum Client with an extractable Ethereum Virtual Machine (EVM) implementation. It is enterprise-friendly and can be used for both public as well as permissioned network projects.

Sawtooth has the following useful features that we can leverage for our project:

- It supports parallel transaction execution which boosts performance.
- The application layer is separate from the core layer so smart contracts can be written in multiple languages.
- Custom transaction processors can be designed for different applications with unique requirements.

While Hyperledger Fabric is the most common amongst developers to build on due to its versatility, we still prefer Hyperledger Sawtooth. This is because Hyperledger Fabric provides a private network whereas Hyperledger Sawtooth is more customisable in this aspect and can be used as a hybrid blockchain network.

Hyperledger Besu could also be considered for the implementation of this project. Besu uses the Ethereum Virtual Machine (EVM) which allows smart contracts to be written in Solidity, the most commonly used language for writing Ethereum smart contracts. Therefore developers can leverage existing Ethereum tools and libraries in their projects. This could be advantageous, for example, in the case of Amazon Managed Blockchain [4] that connects to the Ethereum Mainnet and utilize dedicated Ethereum nodes. Also, Hyperledger Iroha has an architecture that is specifically designed for IoT-based project implementations.

The consensus mechanisms available in Besu are mostly based on Proof of Authority (PoA) or Proof of Stake (PoS) [39]. PoA is useful in the case of permissioned networks but we cannot fully rely on PoA if we want to deploy our project on a hybrid blockchain. The other option, PoS requires that the participating nodes have staked some crypto-currency (in this case, ETH) in order to participate in the PoS protocol.

Furthermore, the consensus algorithms supported by Sawtooth are Practical Byzantine Fault Tolerance (PBFT) and Proof of Elapsed time(PoET) both of which are better compared to the algorithm used in Iroha which is Yet Another Consensus (YAC). All of these algorithms are Byzantine Fault Tolerant [1] but YAC is not crash-fault tolerant by design and also slower and less efficient [37] [66].

Consensus is described in detail in section 4.6

4.5 The Architecture of Hyperledger Sawtooth

In this section, we will describe the main components of Hyperledger Sawtooth, an open-source Hyperledger platform for building distributed ledger applications. Sawtooth follows a modular design philosophy, allowing for pluggable consensus algorithms, transaction families, and other components. This modularity provides flexibility, enabling developers to customize and extend the platform. At the core of Sawtooth's architecture is a blockchain-based distributed ledger (DL), which consists of a series of blocks (called batches in Sawtooth). Each batch represents an ordered set of valid transactions. The ledger is maintained by a network of validator nodes that collectively agree on the state of the blockchain through one of the consensus algorithms mentioned in section 4.6.1.

Figure 6 shows the structure of a validator node in the Hyperledger Sawtooth blockchain from its official guide [45].

The DL distributes consistent copies of itself in each node. The state in figure 6 represents a single instance of the DL in a single validator node. These states are combined

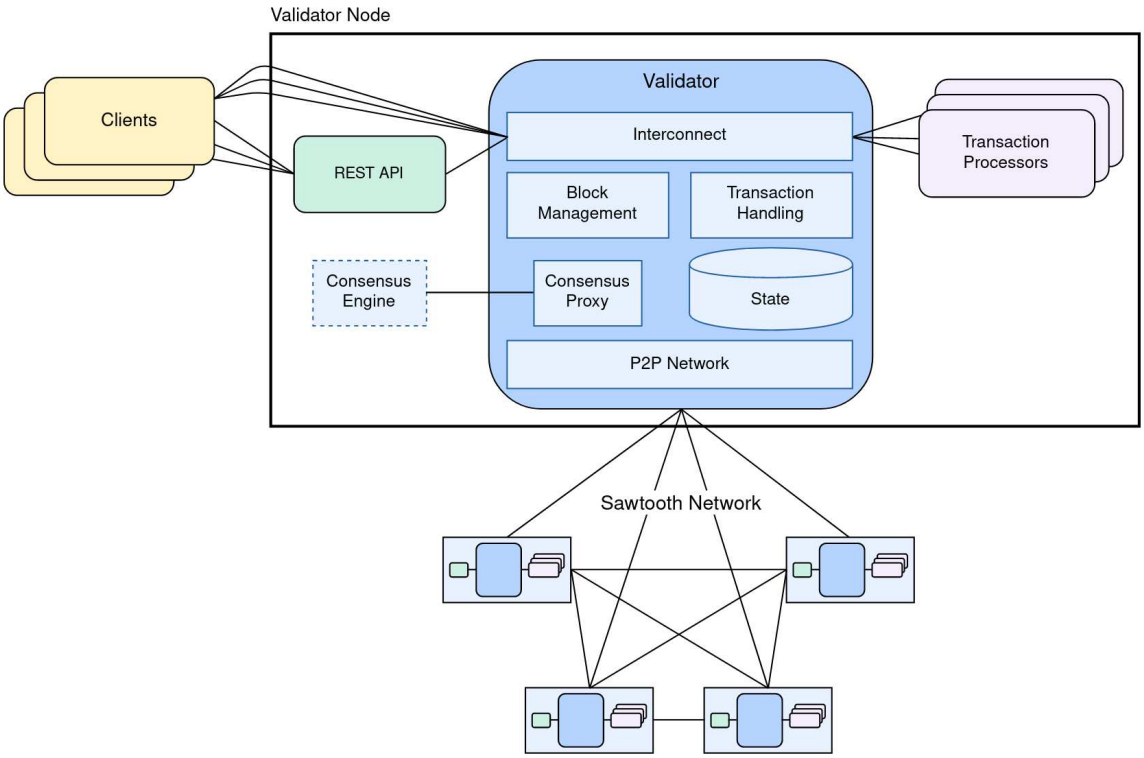


Figure 6: Overview of a validator node in the Hyperledger Sawtooth network [76].

and synchronized across all the nodes on the blockchain to create a consistent global state of the DL. The state is divided into namespaces which allows defining different transaction families that can share the same global state data.

A transaction family defines the structure, semantics, and validation rules for a specific type of transaction. This approach allows different transaction families to coexist on the same blockchain, each serving a specific purpose. For example, there could be separate transaction families for asset transfers, identity management, and supply chain operations. Transaction families enable Sawtooth to separate the application level from the core system level. This allows applications to be written any language that the developer chooses.

A transaction family is composed of three parts:

- **Client**: the users that interact with the transaction family and initiate transactions
- **Transaction Processor:** a piece of code that contains the business logic behind the execution of transactions and is equivalent to a smart contract in blockchain technology.
- **Data Model:** a piece of code that defines how the data that is entered into the blockchain is recorded and stored.

4.5.1 Transaction Processors

Smart contracts are self-executing pieces of code that are deployed on blockchain platforms that perform predefined actions when certain conditions are met. The execution of a smart

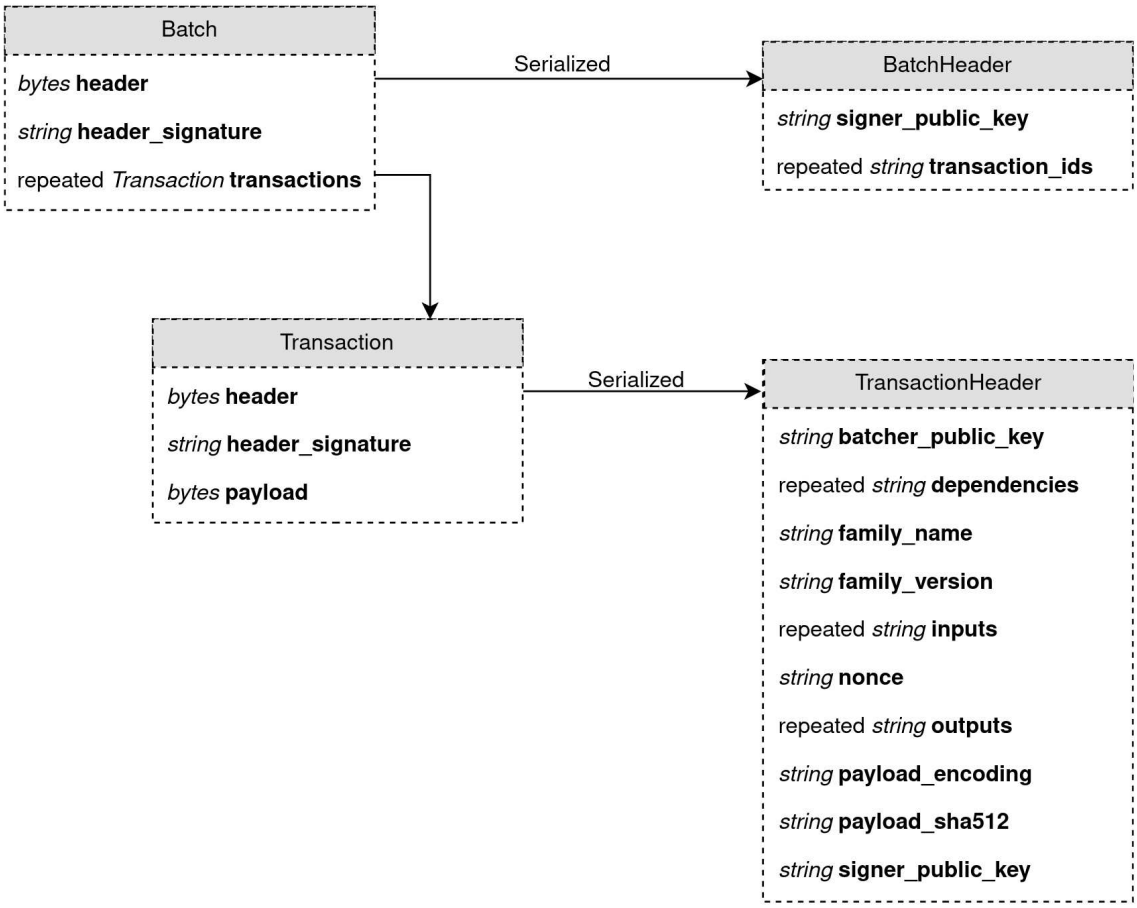


Figure 7: Serialization of Transactions and Batches [76]

contract generates an output representing a transaction that eventually gets added into a block (containing multiple such transactions) and then the blockchain ledger.

In Sawtooth, smart contracts are called Transaction Processors. The business logic behind an application written in Sawtooth is defined in its transaction processor. Figure 6 shows how all the validator nodes access a transaction processor.

As mentioned, a Sawtooth transaction processor can be written in multiple programming languages. Sawtooth also provides SDKs and development tools to simplify the creation and deployment of transaction processors.

A client initiates a transaction and submits it to the validator through a REST API, as shown in Figure 6. This API allows clients to submit transactions, query the blockchain's global state, and receive updates. The validator then calls the transaction processor. Just like other smart contracts, the execution of a transaction processor code results in a transaction. A certain number of transactions are wrapped inside of a batch, and this batch is eventually committed to a state which results in a change in the global state of the DL. A batch is, therefore, the atomic unit of state change.

Each transaction is serialized into a transaction header that includes the signature and public key of the client that initiated it, which is verified by the validator nodes. The transaction header also includes a payload which is the encoded transaction family-specific information related to the transaction, like the clients involved in the transaction and the

data being transferred. Last but not least, the transaction header also has to include the public key of the signer of the batch that it is wrapped in. Including this prevents a transaction from being omitted from the chain and repackaged into a new batch.

Similarly, the batches are also serialized into batch headers that include the client's signature and public key. Serialization converts the transactions and batches into a format that can easily be stored, transmitted and processed in the blockchain. The serialization occurs through protocol buffers [17]. Figure 7 shows how the transactions and batches are serialized.

All the transactions are executed in an isolated environment called an "execution context." Each execution context operates independently in separate instances. This isolation helps provides data isolation and enhanced security and privacy.

Sawtooth employs parallel transaction processing. The platform partitions the global state into subsets, allowing multiple transactions to be processed concurrently. This parallelism enables high throughput and reduces the potential for transaction conflicts and bottlenecks.

4.5.2 Data Models

The fundamental concept of a DL is that there is no centralized storage of data. Blockchains employ the concept of decentralized storage by utilizing the free storage space on all the nodes that comprise it. The data could also be stored on a distributed file system such as Interplanetary File System (IPFS) [53].

In Sawtooth, the data is stored as copy-on-write Merkle-Radix Trees. Copy-on-write means that each tree only creates a copy of itself when changes are made. A characteristic feature of a Merkle tree is that it is created from the bottom to the top. The tree comprises hashes of successive nodes from leaf to root. A single instance of this Merkle-Radix tree in each validator represents the state. For every state transition (addition of a new batch), a single root hash is generated that points to that version of the tree. The tree's root hash consists of the cumulative hashes of all the data and child node pointers on that tree.

Each child node in the tree points to a transaction batch header. The child node is identified by an address which is prefixed by the namespace of the transaction family to which that batch belongs. The root hash is included in the batch header, which all the validators can compare to gain consensus on the expected version of the state and the order of the chain of batches (blocks). This structure ensures data consistency on the ledger and Byzantine Fault Tolerance (discussed in Section 4.6).

The protocol buffers (as mentioned in Section 4.5.1), responsible for serializing the transactions and batches, are also responsible for deserializing them. Deserialization is required when the data is to be retrieved and verified.

Therefore, the data model is defined by the protocol buffers. It is the responsibility of the developer to write the protocol buffer code in such a way that it is (i) concurrent with the Merkle-Radix tree data structure of the global state and (ii) takes into account how the distributed storage of data is to be done.

4.6 Consensus in Blockchains

In a blockchain, the transactions are chained together as blocks. The participating nodes validate each of these blocks on the basis of a consensus protocol. They are then encrypted

and appended to the chain. Without a central authority, a distributed ledger operates in a trustless environment. The consensus protocol plays a crucial role in maintaining the integrity and security of the blockchain, ensuring that the participating nodes always agree on the validity of the transactions and the state of the ledger.

Once the nodes verify and append a block, it is computationally infeasible to modify or delete it, making the blockchain immutable. The consensus algorithm is designed with the assumption that the blockchain network is trustless and unreliable with adversarial users [9]. Various consensus algorithms exist, such as Proof of Work (PoW), Proof of Stake (PoS), and Practical Byzantine Fault Tolerance (PBFT).

A consensus protocol is expected to have the following properties [31]:

- **Safety:** All nodes must receive the same set of inputs in the same order to produce the same valid output (or machine state) that all the nodes can verify.
- **Liveliness:** The protocol is said to have been executed when all the valid nodes reach a consensus on the output.
- **Fault Tolerance:** The network functions as expected even when there are node failures

In a trustless network, ensuring fault tolerance is of utmost importance. Two kinds of fault tolerance are taken into account while designing consensus protocols:

- **Crash Fault Tolerance:** Some nodes in the network might crash or fail to perform due to failures in their hardware or software. In a crash fault-tolerant consensus algorithm, a head, or leader node, chosen by some mechanism, is responsible for appending a validated block to the chain. If the head node turns out to be faulty, a new one can be chosen. This algorithm is expected to work as long as $t < n/2$, where n is the total number of nodes in the network, and t is the number of faulty nodes. [79]
- **Byzantine Fault Tolerance:** This is a property of a distributed system which prevents it from being susceptible to possible system failures arising from the Byzantine Generals' Problem (BGP) [59]. In the context of blockchains and other DLTs, BGP is a dilemma in which there needs to be enough reliable and honest nodes so that even if data exchanged between some of these nodes gets tampered with (by Byzantine nodes) or destroyed midway; the nodes can still reach a consensus on a common decision (e.g., adding a transaction record to the blockchain) [1].

Understanding the consensus protocol to be used in any blockchain implementation is essential. The consensus protocol ensures that the records on the DL are trustworthy and immutable.

4.6.1 Consensus Protocols Available in Sawtooth

The following consensus protocols are described in the Sawtooth documentation [45]:

1. **Practical Byzantine Fault Tolerance (PBFT):** A byzantine fault-tolerant, voting-based consensus protocol that does not support forking. The algorithm requires at least four nodes to operate. Each node must install and run the PBFT consensus engine package and the PBFT consensus engine, respectively.

2. **Proof of Elapsed Time (PoET):** A Nakamoto-style consensus algorithm where instead of relying on the computationally resource-expensive "proof of work", a trusted execution environment (TEE) randomly generates a wait time based on which a node gets to add a block to the ledger chain. Intel originally developed it for Hyperledger Sawtooth.

3. **Raft:** A leader-based, crash fault tolerant (can withstand system failures on up to $n/2$ nodes) consensus protocol that does not allow forking of the blockchain network. This protocol is suitable for small networks that have restricted ownership.

4. **Devmode:** A consensus protocol that uses a simplified random-leader algorithm to decide the head node that will add the next transaction record to the block. It is intended to be used for testing a transaction processor with a single Sawtooth node.

While Sawtooth supports PBFT and PoET, Raft is still in the prototype version and cannot be used in production. Devmode cannot be used in production either since it can only be implemented on a single-node Sawtooth network for testing purposes. So, the choice comes down to PBFT and PoET. At the time of this writing, the latest version of the framework, sawtoothv1.2, only has support for PBFT and not PoET. However, we will use sawtoothv1.0 to implement this project, which supports both PoET and PBFT.

4.6.2 PBFT

In a blockchain network of nodes, some nodes may be faulty and behave arbitrarily. These are called Byzantine nodes. Furthermore, it is possible for messages to be duplicated, delayed, dropped or delivered out of order, even in the absence of faulty nodes.

The Practical Byzantine Fault Tolerance (PBFT) consensus protocol is said to preserve the properties of safety and liveness as long as the maximum number of faulty nodes in the network are $\lfloor \frac{n-1}{3} \rfloor$ where n is the total number of nodes in the network [18]. Assuming that we are operating in a network where there is at least one Byzantine node, the minimum number of nodes required for PBFT to work is $n = 4$.

Cryptographic techniques are used to secure the delivery of messages so that even in the presence of Byzantine nodes and attackers in the network, the information contained within a message is confidential. For this purpose, the messages should contain:

- Message digests produced by collision-resistant hash functions (such as SHA256);
- Message Authentication Codes (MACs);
- Public-key signatures (each node knows the public keys of every other node).

If the Byzantine nodes in the network are under the control of an attacker, Eve, it is assumed that Eve cannot:

- Delay communication between the nodes indefinitely;
- Compute information from the message digests;
- Find two messages with the same digests;

- Produce a valid signature on a non-Byzantine node.

It is the state or the consistent copies of the DL that is available to every non-Byzantine node that ensures that the Byzantine nodes cannot cause direct damage to the ledger data. This upholds the safety property of the consensus. However, the safety property itself does not guard against faulty nodes. The liveness property ensures that the Byzantine nodes cannot delay messages indefinitely. This means that a client always receives a reply to their messages as long as delay(t) does not grow faster than t indefinitely; delay(t) is the duration between time t when the message is sent for the first time (the message keeps getting retransmitted until a response is received) up until a response is received for it.

The algorithm is described in the following steps:

1. The nodes move through a succession of configurations called views. Upon receiving a message from a client each node runs the same set of instructions and contains a copy of the latest state of the network. The copy of the state on each node contains a message log that has all the messages that the node has accepted in its current view and an integer denoting the view number.

   Let us denote the set of nodes by R and each node is denoted by an integer in the range of $[0, |R - 1|]$.

2. In a view, one node is the primary node and the rest are backups. The primary node p is decided using the formula, $p = v \mod |R|$, where v is the view number.

3. The client sends a message with a request of the form $(REQUEST, o, t, a)_{\sigma_c}$ to p where, o is the requested operation, t is a timestamp, each of which is ordered in such a way that later timestamps are higher than previous ones, c is the client's identity, and the function $()_{\sigma_c}$ represents the digest signed by the client. The primary node p broadcasts the client's request to the backups.

4. The primary node p multicasts the message to all the backup nodes. The backup nodes execute the requested operation and sends a response directly to the client. The response is of the form $(REPLY, v, t, a, i, r)_{\sigma_i}$ where v is the view number for the client to keep track of p, i is the node number and, r is the the result of executing o.

   The nodes generate proofs that their copies of the state are correct after the completion of x operations, where x is a constant whole number. In other words, when a client request with sequence number divisible by x is executed, a proof of the validity of the latest state is generated which results in a new state called a checkpoint.

5. The client awaits $f + 1$ responses from the nodes with the same t and r and valid signatures. This is what ensures that r is valid and can be accepted since at most f nodes can be Byzantine.

6. In case the client does not receive responses within a given time-frame, their message is broadcasted to all the nodes. All the nodes are expected to remember their last response to any given client. Thus, if the response to the last message has already been processed, all the nodes simply resend their responses back to the client.

If the response to the last message is not processed, all the non-primary nodes relay the request in the message to the primary node p, and p is expected to multicast this request again to all the nodes like in step 4 and the ensuing steps continue. However, if p does not multicast the message, all the other nodes suspect it to be faulty and this calls for a view change with a new p.

The protocol behind the view change is what gives PBFT its liveness property. If a timer for a backup-node i expired in v, it means that the the p in v has failed and a view change to view $v + 1$ is triggered.

Now i multicasts a view change message, $(VIEW - CHANGE, v + 1, n, C, P, i)_{\sigma_i}$ where, n is the sequence number of the last stable checkpoint, s known to i, C is the set of $2f + 1$ valid checkpoint messages that prove the validity of s, and P is a set of logs related to each message m with a sequence number that is greater than n that was sent to i.

When the new primary, p' in view $v + 1$ receives $2f$ valid view-change messages, it multicasts a view change message to all the other backup nodes in that state and decides the latest stable checkpoint for them.

The proof of correctness of the safety and liveness properties of this algorithm is given in [18].

4.6.3 PoET

The Proof of Elapsed Time (PoET) consensus protocol has no known formal mathematical proof of its security or efficacy. PoET relies on the security provided by a Trusted Execution Environment (TEE) [15]. The PoET protocol is both crash-fault tolerant as well as Byzantine fault tolerant. It is similar to the Proof of Work (PoW) consensus protocol that was first used in the Bitcoin blockchain [68].

Just like most crash-fault tolerant consensus protocols, both PoW and PoET use a leader-based strategy to add blocks to the DL. In PoW, the leader is the node that can solve a complex cryptographic puzzle the fastest. This depends on the node's inherent computational power and has been criticized to be environmentally harmful as it involves large energy consumption and consequently high CO_2 emissions [75].

PoET employs a "lottery-based" system to elect a leader node. This is done by the hardware-assisted Trusted Execution Environment (TEE), e.g., Intel SGX [51] or ARM TrustZone [7], which generates random wait times t_i in each node. The i^{th} node with the shortest t_i is elected as the head node that gets to add the next block to the DL.

A node builds a block that would extend the chain in the DL and its TEE computes a t_i for it. This t_i is the duration for which the node has to wait until it can claim leadership to add this block to the chain. When the time duration is completed, the node broadcasts a message that contains its proposed block and a proof of the correctness of its t_i which can be verified by all the nodes on the network.

In PoW, the correctness of the leader election is checked by verifying the correct solution to the cryptographic puzzle. In PoET, the correctness is checked based on attestation by the TEE. The TEE vouches for the correct generation and enforcement of t_i.

When a new block is received, each node checks (i) if the addition of the block results in a valid longer chain in the DL, and (ii) if the wait time t_i has elapsed since the creation of the block. If both of the above criteria are met, the block gets added to the chain and the nodes query their respective TEEs to receive a new t_i.

Intel's PoET protocol employs a rate-limiting function called "z-test" to prevent an attacker who has compromised a TEE in a device from systematically "winning" the competition for smallest t_i [15]. This countermeasure also prevents any one TEE from dominating the network, which would break the very foundation of the decentralized architecture.

In PoW, a node with more computational power than others on the network can end up dominating the network.

4.6.4 PoET vs PBFT

PBFT is best for small, consortium-style networks that do not require open membership [45]. This is because the PBFT protocol is communication-intensive. We can see this from the algorithm behind the protocol as described in section 4.6.2. Therefore, for large trustless networks that we expect our project to use, using the PBFT consensus mechanism could slow down the process of making records on the DLT.

On the other hand, PoET relies on TEEs. This means that the participating nodes in the network need to have TEE capabilities for the algorithm behind the PoET protocol to be reliable enough.

A lot of modern IoT devices contain TEEs like Intel-SGX [51] and ARM-TrustZone [7]. Furthermore, our prototype would require a TEE in order to function as it is supposed to. Therefore, we can say that choosing PoET as our consensus algorithm would be most suitable.

5 Design

5.1 Architecture

Zero-touch bootstrapping (ZTB) is a way of bootstrapping an IoT device such that when powered on, it automatically gets connected (bootstrapped) to the user's network and can be controlled by controller devices on the same network. As discussed in section 4.2, in modern ZTB solutions, the bootstrapping device connects to the IoT device using a generic network. It then attempts to validate the device's identity and supply it with the credentials needed to connect to its user's WiFi network. These credentials are retrieved from a centrally-controlled server belonging to the ZTB solution provider, such as Amazon's WiFi Locker [5]. This mechanism comes with two main problems:

1. It does not prevent the device from connecting to the wrong endpoint (misbinding) and consequently getting bootstrapped to the wrong network (described in Section 4.3.3).

2. It can only be performed using vendor-specific devices by "prebinding" the devices with their users at the time of purchase, e.g., an Amazon Echo can only perform ZTB on an Amazon IoT device.

To counter problem number 1, we have used a Distributed ledger technology (DLT), in this case, the Hyperledger Sawtooth blockchain. The transaction processor generates unique bootstrapping network parameters (such as an SSID) for a particular device intended to be used by a particular user. These parameters are encrypted and transferred to the next user of the IoT device upon successful execution of the transaction processor code. This implies that the ownership transfer and the transfer of parameters take place only after being validated by the validator nodes in the network. Now an IoT device can only connect to a bootstrapping device that it is intended to since the parameters are unique to both of them, thus reducing the chances of misbinding.

Furthermore, we also audit the lifecycle of each IoT device, allowing it to verify its ownership status – that it is getting bootstrapped by the bootstrapping device of its rightful current owner – after the bootstrapping device has successfully connected to the device using the unique network parameters that it received as part of the transaction.

Coming to problem number 2, traditional methods of prebinding the IoT device for ZTB, such as those explained in Section 4.2, have the following pre-requisites:

- The IoT service provider should have management access to the IoT users' accounts on a cloud-based framework.

- Relationships between users and service providers must be established before purchase.

These requirements are necessary because the device verification for ZTB is done per data stored on a centralized server controlled by the ZTB solution provider. However, IoT devices are bought from different sources in real life, and ZTB through pre-binding does not work in these cases.

The authentication mechanism we provided can produce an output that could be supplied to any ZTB solution provider as a sufficient authentication mechanism to directly

supply the user's credentials from their servers and perform ZTB. We have described this in detail in Section 5.2

In summary, here is how blockchain helps with ZTB:

- It does not need to prebind the device with its owner at the time of device purchase.
- Allows confidential delivery of device-specific bootstrapping network parameters from the current owner to the next owner.
- Creates the necessary parameters for single-use bootstrapping networks per-device-per-user.
- Blockchain acts as a trust anchor to validate every interaction recorded on it employing its consensus protocol.

5.1.1 Overview

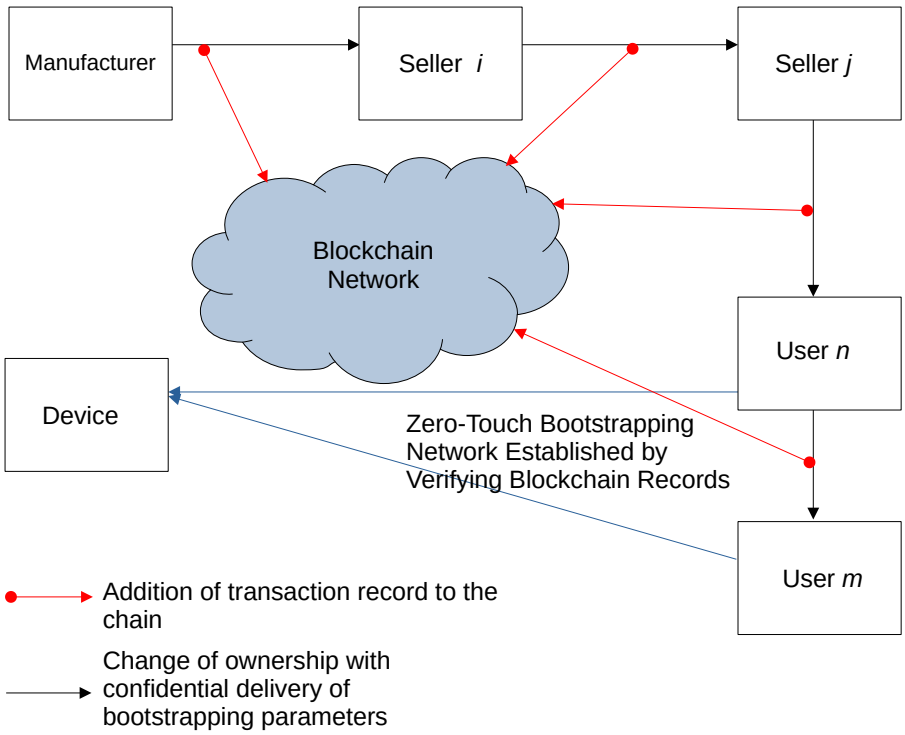


Figure 8: Overview of the zero-touch boostrapping architecture – with transfer of ownership and ZTB

In Figure 8, we have shown the overview of the architecture of ZTB using blockchain technology. The steps involved are described as follows:

1. The manufacturer adds a new IoT device with a unique public address to the blockchain. This device now has some default bootstrapping parameters known only to the manufacturer.

2. The device is then transferred to its next owner, seller i, and this change of ownership is recorded on the blockchain. The device generates new bootstrapping parameters that are securely delivered in encrypted form to seller i.

3. After this, every change of ownership of the device gets added as a transaction to the blockchain. New bootstrapping parameters are generated and securely transferred with every such transaction.

4. Upon reaching a user m, ZTB takes place with the help of the bootstrapping parameters and proof of ownership retrieved from the blockchain, which is further described in Section 5.1.3.

5.1.2 Change of Ownership of the IoT Device

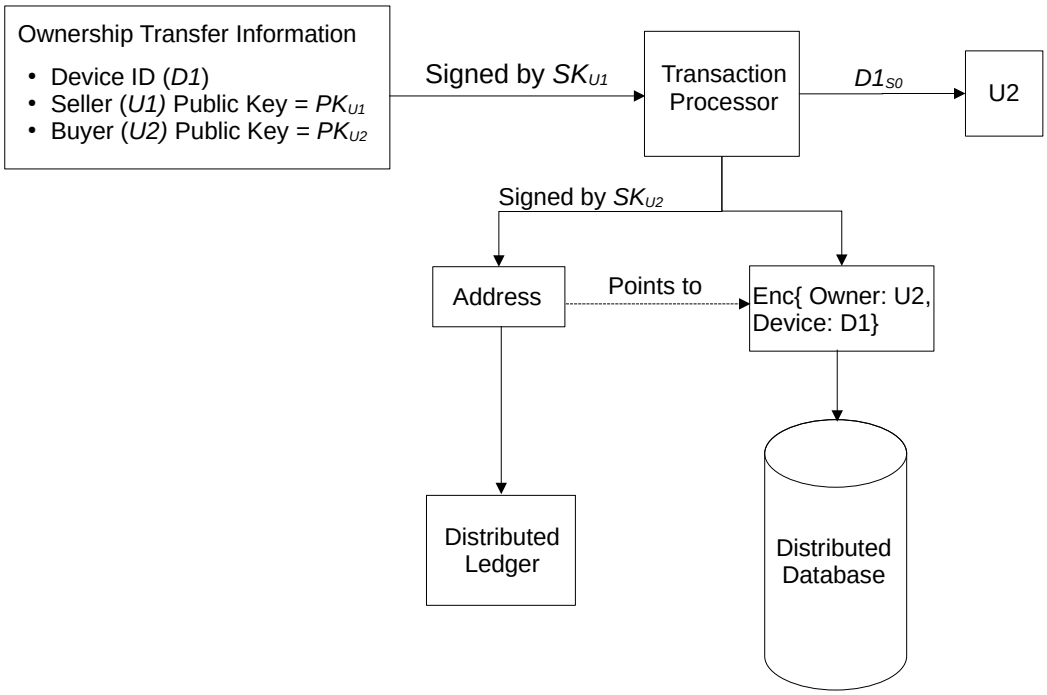


Figure 9: Recording a transfer of Ownership on the Blockchain.

As mentioned in Section 5.1.1, every change of ownership of the IoT device is recorded on the blockchain. Figure 9 shows how the transaction is recorded on the DL and where the associated data is stored.

The current user, or the seller, $U1$, initiates an ownership transfer of their device with a unique ID, $D1$, to a buyer $U2$. The ownership transfer request includes the public keys of PK_{U1} and PK_{U2}, and $D1$.

When $U2$ accepts the transfer, the transaction processor generates a unique bootstrapping network parameters (such as an SSID). Let these parameters be $SSID_{D1}$. These

parameters are encrypted using ECIES encryption to generate $D1_{S0}$ which is transferred to $U2$. The transaction outputs the following:

- An encoding of device $D1$ and its association with its new user, $U2$. This is stored on the distributed databased
- A hexadecimal address that points to the location of where the above encoded information is stored. This address is signed by $U2$'s private key, SK_{U2}. The validator nodes validate this signature before adding the transaction to the DL

It is important to note that the "Distributed Database" in Figure 9 might refer to a distributed file system (such as IPFS [53]) where chunks of the data are stored across the nodes on the network or on a decentralized cloud database (such as Amazon Web Services (AWS) [8]). The DL and the distributed database along with the participating nodes comprise the blockchain.

5.1.3 Design Options for ZT-Bootstrapping

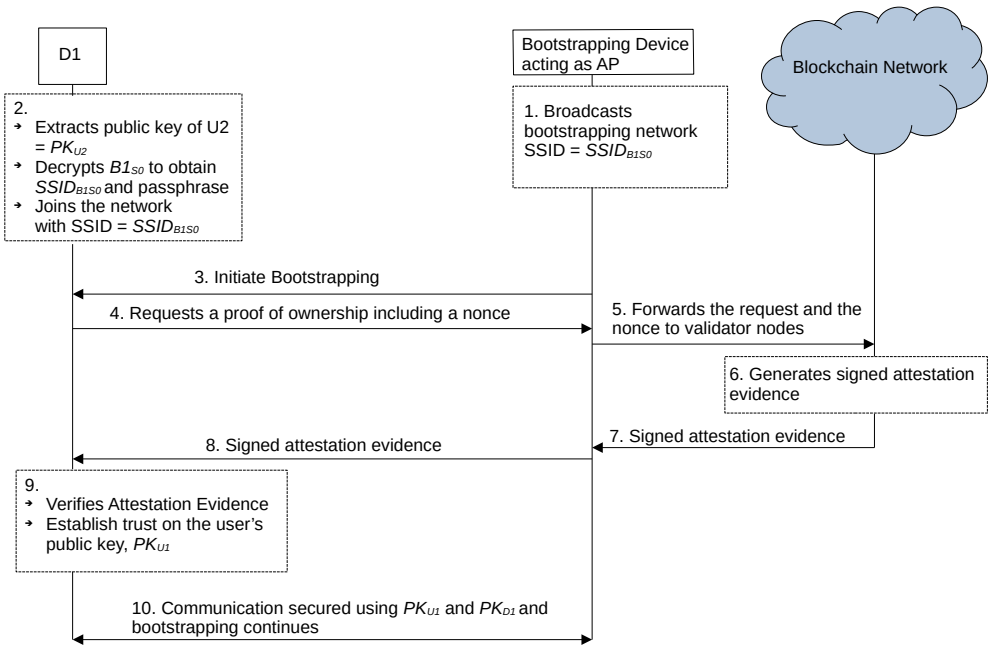


Figure 10: Architecture of Option 1 – Making the Bootstrapping Device Act as an Access Point

We have the following two design options for the bootstrapping procedure: in Option 1 the bootstrapping device acts as an access point; in Option 2 the IoT device acts as an access point.

Option 1: *Making the Bootstrapping device act as an access point.* The design is shown in figure 10. The bootstrapping device once turned on, acts as an access point (AP) and broadcasts its network. The bootstrapping device can enter bootstrapping mode periodically whenever it expects a new device (e.g. when it receives a blockchain block indicating that the user has purchased a new device.)

IoT device $D1$ joins the bootstrapping network by using the parameters that it generated as part of the change of ownership.

Next, $D1$ sends a request for proof of ownership to the bootstrapping device along with a cryptographic nonce (to avoid replay attacks). The request and the nonce are signed by the long-term public key of the user and sent to the validator nodes. The validator nodes then verify if the user initiating the bootstrapping is the rightful owner according to the records on the blockchain.

If the bootstrapping device returns valid attestation evidence from the blockchain in response to the challenge sent by $D1$, a secure channel is established between $D1$ and the bootstrapping device and bootstrapping can continue on this network.

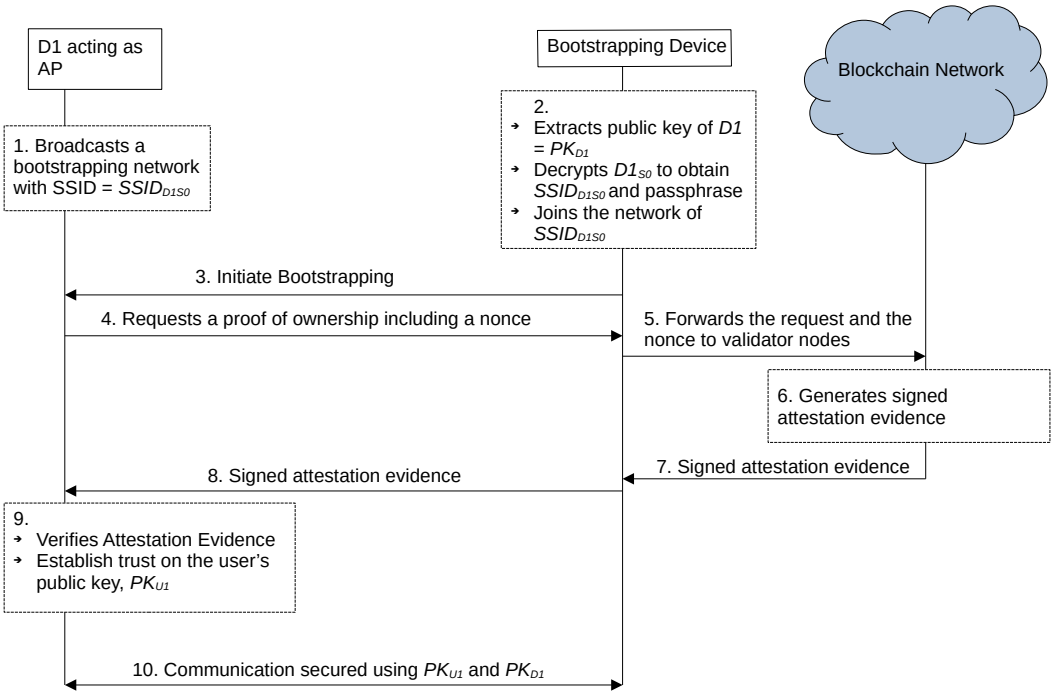


Figure 11: Architecture of Option 2 – Making the IoT Device Act as an Access Point.

Option 2: *Making the IoT device act as an access point.* The design is shown in figure 11. In contrast to Option 1, here, $D1$ acts as the AP for the bootstrapping device to connect to. The user's private key that is available to their bootstrapping device is used to decrypt $D1_{S0}$ and get the unique bootstrapping network parameters. Now the bootstrapping device is connected to $D1$'s unique bootstrapping network.

Now, once again, the same steps as design option 1 are repeated. $D1$ sends a request for proof of ownership to the bootstrapping device along with a cryptographic nonce (to avoid replay attacks). The request is sent to verify that the user initiating the bootstrapping is the rightful owner, according to the records on the blockchain.

If the bootstrapping device returns valid attestation evidence from the blockchain in response to the request sent by $D1$, a secure channel is established between $D1$ and the bootstrapping device, and bootstrapping can continue on this network.

5.2 Cryptographic Primitives

In this section, we list the cryptographic functions that we have used for (i) the generation of unique bootstrapping parameters per-device-per-user, (ii) confidential delivery of bootstrapping parameters to the new owner of the IoT device, and (iii) proof of ownership of the IoT device.

5.2.1 Generation of unique bootstrapping parameters

As discussed previously, the bootstrapping parameters should be unique per-device-per-user. Thus, the bootstrapping parameters need to be generated in such a way that it is dependent on the unique ID of the device as well as the unique ID of its current owner. However, these two requirements alone are not sufficient. Assuming that the algorithm for generating the unique parameters is public, it would be easy for an attacker who knows the unique ID of the device and its next user to generate the same parameters on their own and spoof the access point with a rogue device. Even if the algorithm is not public, an attacker can intercept the unique parameters while a bootstrapping network is being established and can later broadcast the same unique network with their rogue AP and have the IoT device connect to it.

Therefore, each time an IoT device needs to establish a unique bootstrapping connection with a bootstrapping device, it must use a fresh set of bootstrapping parameters. To preserve the freshness of the parameters, it is essential to also input a randomly or pseudorandomly generated value in the parameter generation function. Let the function be f. We can define the function as:

$$f : f(x, y, z) = p_i \tag{6}$$

where, x = unique ID of the IoT device, y = unique ID of the current owner, z = pseudorandom value such as a cryptographic nonce and p_i is the set of uniquely generated parameters for every i^{th} time the device needs to be bootstrapped.

This function is included in the smart contract. This is because the smart contract is deployed on the blockchain, so the code is immutable. Therefore, the legitimacy of the parameters is preserved just as much as the legitimacy of the transfer of ownership of the device.

5.2.2 Confidential delivery of bootstrapping parameters

We have used the Elliptic Curve Integrated Encryption Scheme (ECIES) in order to securely transfer the IoT device's bootstrapping parameters to the bootstrapping device [63].

As an encryption scheme, ECIES uses the following functions:

- **Key Agreement (KA) function** to generate a shared secret between the two communicating parties;
- **Key Derivation Function (KDF)** to produce a set of keys based on some inputted parameters;
- **Symmetric Key Encryption Algorithm**;

- **Message Authentication Code (MAC)** to authenticate messages;
- **Hash Function** that is used by the KDF and MAC functions.

An ephemeral public-private key pair is first generated for the seller of the IoT device. This key pair is used to encrypt the IoT device's bootstrapping parameters. This key pair is generated using Elliptic Curve Cryptography (ECC). The elliptic curve E in ECC is calculated over a finite field or a Galois Field (GF). A Galois field, also called a finite field, is a field over the curve E used in Elliptic Curve Cryptography and is defined by the Weierstrass equation, $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, where $a_1, a_2, a_3, a_4, a_6 \in GF$ and $\Delta \neq 0$, where Δ is the determinant of E [63]. The private key is a random scalar value represented by u and the public key is calculated as $U = u \cdot G$ where G is the generator of E. This ephemeral key pair is created specifically for this encrypted transfer and is discarded once the decryption of it is complete.

Let the long-term public and private keys of the buyer be v and V, respectively. The KA function is used to calculate the value of a shared secret which is equal to $u \cdot V$.

The shared secret, $u \cdot V$ is used by the KDF to calculate a concatenated output of a symmetric encryption key, k_{enc} and a MAC key, k_{MAC}.

Ciphertext $D1_{S0}$ is generated by encrypting the bootstrapping parameters $SSID_{D1_{S0}}$ with k_{enc}.

Next, a tag is generated by applying the MAC function on $D1_{S0}$ using k_{MAC}.

Now the cryptogram $(U \mid tag \mid D1_{S0})$ is sent to the bootstrapping device to be decrypted.

The shared secret can now be calculated by the new user of the IoT device using their own private key and the received ephemeral key of the IoT device as $v \cdot U$ which is the same as $u \cdot V$. This is based on the Diffie-Hellman Key exchange protocol [25].

This shared secret is then used to produce k_{enc} and k_{MAC}.

The k_{MAC} is used to produce tag^* which can be compared with tag received in the cryptogram. If they are both equal, the MAC is verified which proves that the message received from the IoT device is authentic.

Now the bootstrapping device can decrypt $D1_{S0}$ using k_{enc} and get the bootstrapping parameters, $SSID_{D1_{S0}}$.

The ECIES transfer is defined in and carried across the transaction processor (or smart contract). This is to ensure that the transfer of parameters takes place at the end of a valid transfer of ownership of the IoT device. Since the transfers are validated by all the validator nodes on the blockchain by a consensus algorithm, it acts as an extra step to ensure that the bootstrapping device belonging to the correct user receives the encrypted parameters.

5.2.3 Proof of Ownership of the IoT Device

We have shown in the design of the change of ownership what is recorded on the blockchain in figure 9.

All the transactions are encoded (hashed) and the digest is signed by the private key of the seller (or the client that initiated that transaction). This signature can be verified by the validator nodes on the network with the signer's public key that is included in the TransactionHeader.

The signature verification is done using the Elliptic Curve Digital Signature Algorithm (ECDSA). Both the buyer's well as the seller's key pairs are ECDSA key pairs that are generated using the secp256k1 elliptic curve [19].

The steps involved in ECDSA signing and verification are as follows [55]:

1. The algorithm requires domain parameters, p, q and g. q is a 160-bit prime number and p is a 1024-bit prime number such that $q \mid p - 1$. g is a generator of a unique cyclic group of the order $q \in \mathbb{Z}_p^*$ and can be computed using the formula, $g = h^{(p-1)/q} \mod p$ where $h \in \mathbb{Z}_p^*$ and $g \neq 1$.

2. Let the private key of the seller be a pseudorandom integer, x such that $1 \leqslant x \leqslant q-1$. The public key y is calculated using x as $y = g^x \mod p$.

3. A random integer k is selected such that $1 \leqslant k \leqslant q - 1$. This is used to compute $X = g^k \mod p$ from which we can calculate $r = X \mod q$. If $r = 0$, then a new k is chosen.

4. Then we compute $s = k^{-1}(e + xr) \mod q$ where e is the hash of the message that is being signed. If $r = 0$, then a new k is chosen and both r and s are recalculated. Otherwise, the signature of the seller is produced which is (r, s).

5. Now, before the seller can add their transaction to the blockchain, the seller's signature, (r, s), needs to be verified by all the validator nodes. The first step is to ensure that both r and s are in the interval $[1, q - 1]$.

6. Then e is computed using the same hashing function followed by the computation of $w = s^{-1} \mod q$. Using these, $u_1 = ew \mod q$ and $u_2 = rw \mod q$ are computed. Finally, $X = g^{u_1} y^{u_2} \mod p$ and $v = X \mod q$ are calculated. The signature is accepted if and only if $v = r$.

Once this signature has been verified by all the validator nodes this signed transfer record is immutably stored on the blockchain. When a bootstrapping device has successfully connected to the unique bootstrapping network of an IoT device, the IoT device sends a challenge to the bootstrapping device with its unique address and a cryptographic nonce (to avoid replay attacks). The address of the device and its rightful owner are extracted from the signed record which points to the location of that data. The public key of the user who has initiated the bootstrapping is compared with that of the rightful owner as recorded on the blockchain. If these two values match, the device is bootstrapped.

6 Implementation

We have forked a GitHub repository that implements an application to transfer assets across the Hyperledger Sawtooth blockchain, called transfer-chain-js [23]. We then edited the code and designed our prototype. The new repository that implements our design is called ZTB-Hyperledger-Sawtooth [10]. Since the example usage scenarios described in section 3.3 include a webstore, we wanted to implement a web UI for adding and transferring the devices. The transfer-chain-js app was a convenient starting point since we only needed minor code changes to implement this prototype.

6.1 Transfer Chain

This app belongs to the transfer-chain-js repository. It leverages Hyperledger Sawtooth to create users with public and private key pairs who can add devices to transfer between each other. It has two parts – the processor and the client.

6.1.1 Processor

The processor is where we define the logic for the transaction processor of our application. The processor directory contains two files: `handler.js` and `index.js`. The `handler.js` file works as the transaction processor.

The `getAddress()` method contains an encoding function:

```
const { createHash } = require('crypto')

const getAddress = (key, length = 64) => {
    return createHash('sha512').update(key).digest('hex').slice(0, length)
}
```

The `createHash()` function creates a hash object for the SHA512 hashing algorithm in this method. The `update()` method updates this hash object with additional data specified in the key parameter. The `digest()` function computes and retrieves the hash digest or output as a buffer object that can be converted into a hexadecimal hash encoding. Finally, the `slice()` function extracts this hash's portion in the range specified in its parameters.

The `getAssetAddress()` and `getTransferAddress()` functions listed below use the `getAddress()` function to generate addresses that will eventually point to data in the distributed system. All the data is stored in encoded form.

```
const FAMILY = 'transfer-chain'
const PREFIX = getAddress(FAMILY, 6)

const getAssetAddress = name => PREFIX + '00' + getAddress(name, 62)
const getTransferAddress = asset => PREFIX + '01' + getAddress(asset, 62)
```

The `handler.js` file has four main functions:

1. `createAsset()`: This function creates an asset with a unique user-defined name and generates and returns a hash of the asset and the address of its owner that can be updated in the state. The devices addresses generated using the `getAddress()` function are ensured to be unique for each device because of the following piece of code that throws an error if the asset name inputted is already in use:

```
const entry = entries[address]
   if (entry && entry.length > 0) {
       throw new InvalidTransaction('Asset name in use')
     }
```

2. `transferAsset()`: This function initiates an asset transfer. The current owner of the asset signs the transfer, and this signature is verified with the public key of the owner of the asset as retrieved from the ledger. A transfer address is generated using the `getTransferAddress()` function. The transfer address is updated to the state and points to the encoded data about the asset and its owner who initiated the transfer.

```
const address = getTransferAddress(asset)
    .
    .
    .

    if (signer !== decode(entry).owner) {
      throw new InvalidTransaction('Only an Asset\'s owner may
      transfer it')
    }


        return state.set({

    //address that points to the initiated transfer of an asset by
    //its current owner
        [address]: encode({asset, owner})
      })
```

3. `acceptTransfer()` - This function allows new owners to accept the asset being transferred to them. The signature of the last owner is verified before the state is updated with the device's new owner.

4. `rejectTransfer()` - This function allows the user to whom the asset is being transferred to reject the transfer if they do not want it.

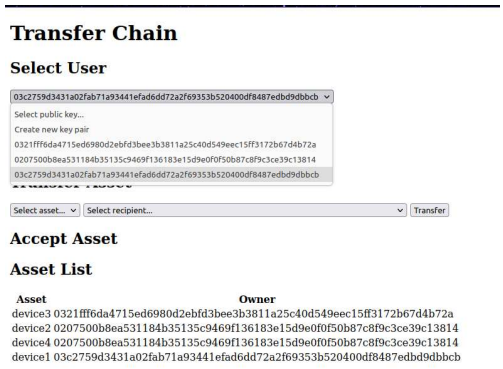
The `JSONHandler` class extends the `TransactionHandler` constant, which accesses the sawtooth-sdk's processor node module. `JSONHandler` is a handler for JSON-encoded payloads. It conveys the change that should be applied to the ledger's state.

Finally, the `index.js` file imports the processor module of the `sawtooth-sdk` into the `TransactionProcessor` and the `handler.js` file into `JSONHandler`. These are used to

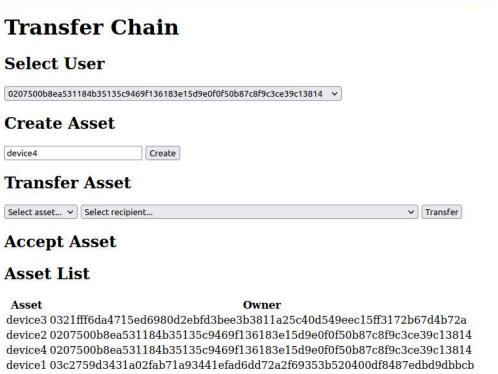
initialise a dockerised Sawtooth validator launched at port 4004 that will follow the logic defined in the `TransactionProcessor`.

The data of the users and their respective assets are stored using JavaScript localStorage, which stores data on the browser that is not deleted when the browser is closed, and the system is rebooted. This kind of local storage is suitable for our project since its purpose is to serve as a simple demonstration that does not require large amounts of data to be stored. For practical purposes, to store user and device data, either a database like PostgreSQL [72] or an off-chain decentralized peer-to-peer file storage system like Inter-planetary File System (IPFS) could be used [53].

6.1.2 Client

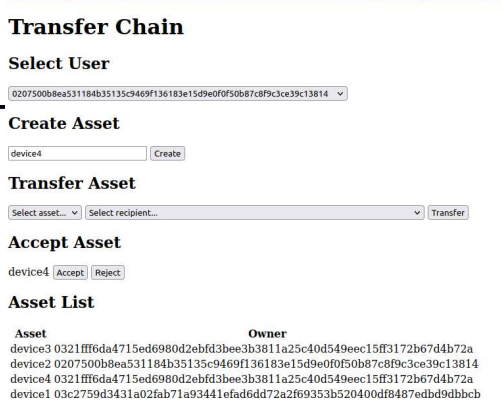


(a) Select user or create new key pair.



(b) The selected user creates an asset - device4.

(c) The selected user transfers one of their assets, device4, to another user.



(d) The selected user can accept or reject the transfer of the asset to them - device4

Figure 12: The user interface of transfer-chain-js.

The client has four main files - `apps.js`, `components.js`, `state.js` and `index.html`.

The `app.js` file defines the logic and functionality behind the web application, the frontend of which is written in `index.html`. The `components.js` file has three functions

for the `index.html` file – `addOption()`, `addRow()`, and `addAction()`. These functions are used to add a new select option, a new table row, and a div block with accept/reject buttons, respectively.

The frontend allows one to select a user or create a new ECDSA public and private key pair for the user (see figure 12a). The users are referenced by their respective public keys.

Once a user is selected, they can add an asset or transfer one of their old assets to another user's address (see Figures 12b and 12c). They can also accept or reject an asset being transferred to them (see Figure 12d).

Finally, an asset list of an asset name and its respective owner is constantly updated with every transfer.

Every action carried out by the transaction processor is logged on the console.

```
Initializing JSON handler for Transfer-Chain
Connected to tcp://localhost:4004
Registration of [transfer-chain 0.0 application/json] succeeded
Received TpPing
Handling transaction:  create > device1  :: 03c2759d...
Handling transaction:  create > device1  :: 03c2759d...
Handling transaction:  create > device2  :: 0207500b...
Handling transaction:  create > device2  :: 0207500b...
Handling transaction:  create > device2  :: 0321fff6...
InvalidTransaction: Asset name in use
    at /home/ushana/work/coding/enc/transfer-chain-js/processor/handlers.js:30:15
Handling transaction:  create > device2  :: 0321fff6...
InvalidTransaction: Asset name in use
    at /home/ushana/work/coding/enc/transfer-chain-js/processor/handlers.js:30:15
Handling transaction:  create > device3  :: 0321fff6...
Handling transaction:  create > device3  :: 0321fff6...
Handling transaction:  create > device4  :: 0321fff6...
Handling transaction:  create > device4  :: 0321fff6...
Handling transaction:  transfer > device4 > 0207500b...  :: 0321fff6...
Handling transaction:  transfer > device4 > 0207500b...  :: 0321fff6...
Handling transaction:  accept > device4  :: 0207500b...
(node:44349) [DEP0005] DeprecationWarning: Buffer() is deprecated due to security and
Handling transaction:  accept > device4  :: 0207500b...
Received TpPing
Handling transaction:  transfer > device4 > 03c2759d...  :: 0207500b...
Handling transaction:  transfer > device4 > 03c2759d...  :: 0207500b...
Handling transaction:  reject > device4  :: 03c2759d...
Handling transaction:  reject > device4  :: 03c2759d...
```

Figure 13: Transfer logs and Transaction Processor Handling shown in the console.

The `state.js` file connects the client to the validator's state record through the REST API. It uses the client module of the sawtooth-sdk. It generates, saves, and retrieves key pairs to and from the blockchain. It also sends data inputted at the frontend to `handler.js` through the validators. Finally, it submits signed transactions to the validator and wraps them into batches.

6.1.3 Docker compose file

The docker-compose file, `docker-compose.yaml` is used to containerize the different components of the application. The docker-compose file runs containers for the following services:

- The **validator container** uses the `hyperledger/sawtooth-validator` [46] docker image and runs four validators that use the PoET consensus engine to add blocks to the ledger.

- The **settings-tp container** runs the `hyperledger/sawtooth-tp_settings:0.8` docker image [43] to use the default settings-tp transaction processor, which is required for modifying the on-chain settings for transactions. All the validators on the network share these settings.

- The **rest-api container** runs the `hyperledger/sawtooth-rest_api:0.8` docker image [42]. This container acts as an interface between the client and the processor.

- The **shell container** runs the `hyperledger/sawtooth-shell:0.8` docker image [44] that contains the CLI interface for configuring and interacting with the validators and the consensus engine.

6.2 ZTB-Hyperledger-Sawtooth

On top of all the functionalities of the transfer chain-js, the ZTB-Hyperledger-Sawtooth app is capable of performing the following additional operations:

1. Secure transfer of Bootstrapping parameters during the change of ownership of the IoT device;

2. Proof of ownership of the device as retrieved from the blockchain.

We are using the eccrypto node module [36] to apply the Elliptic Curve Integrated Encryption Scheme (ECIES).

6.2.1 Architecture of the prototype

In Figure 14, we can see how different code sections are connected to run the business logic. One can also correlate it with the Hyperledger Sawtooth Architecture in figure 6.

- The IoT device is connected to the client through the webpage (UI). This webpage represents the webstore mentioned in the example usage scenarios in section 3.3. Our prototype did not use actual IoT devices, so they are referenced simply by their names.

- Just like the IoT devices, no actual bootstrapping devices were used to implement the prototype, either. The console acts as the bootstrapping device which gets inputs from the client.

- The client is connected to the validators through the REST API run in a dockerized container.

- Each validator contains a copy of the state it gets from the Sawtooth Blockchain. Each validator also runs a consensus engine proxy embedded inside it.

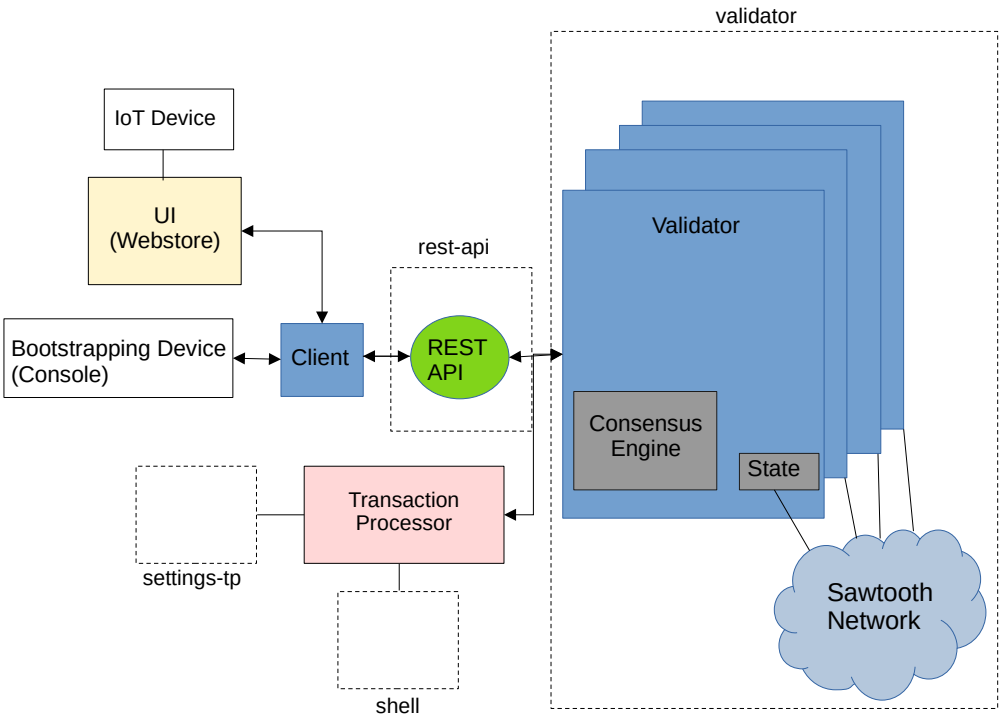


Figure 14: The architecture behind our implementation. (dotted rectangles represent docker containers)

- The transaction processor, where the business logic for our prototype is defined, is connected to the validators. The processor and the client interact with each other indirectly through the validators.

Just like in transfer-chain-js, the four validators that create the Sawtooth blockchain network are implemented using docker containers, so the docker file remains the same in our implementation.

Each validator runs a consensus engine proxy (PoET consensus engine) and has a copy of the global state of the blockchain. The consensus engine helps the validators decide the order in which transaction batches get added to the chain. The copies of the global state help maintain the distributed ledger.

The user interface consists of the webpage that the Sawtooth client hosts. The users interact with the webpage, and the client records these interactions. The client connects to the validators across a REST API running as a docker container, just like in transfer-chain-js. The validator takes the inputs from the client to execute the logic written in the transaction processor. Upon successful execution of this logic, the state of each validator is updated with the necessary changes. The publicly available outputs are returned to the client and reflected on the webpage.

6.2.2 Flowchart of the prototype

Figure 15 shows how our prototype works with the Sawtooth blockchain, from a very technical point of view. Here, the term ´´state" refers to an instance of the global state of the blockchain's ledger, a copy of which is available to every validator node that is being run in the docker container (refer to Figure 14).

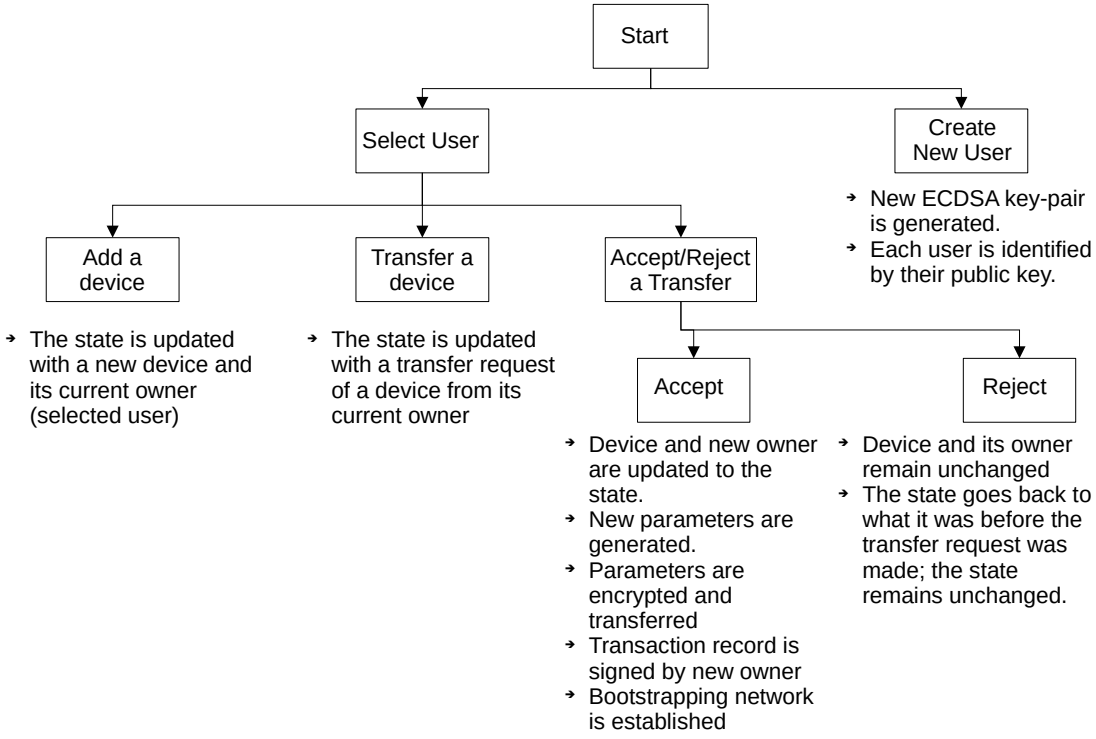


Figure 15: Flowchart of the Prototype

6.2.3 Code

Since our implementation involves mutual authentication between the IoT device and its user's bootstrapping device, the device's unique parameters must be encrypted and transferred from one owner to the next throughout the device's lifecycle. Each of these transfers is recorded on the blockchain. When the bootstrapping network connection between the IoT device and the bootstrapping device is established, the current owner of the device is verified from the records on the blockchain.

The architecture for the transfer of parameters was shown in 9. We have added the following in the `handler.js` file that contains the transaction processor logic:

1. The function below generates parameters for a device:

```
const getAssetParameters = (asset, owner) => {
var nonce = require('crypto').randomBytes(16).toString('base64');
```

```
const parameters = String(getAddress(asset, 62)+String(owner)+nonce);
return parameters;
}
```

We call this function, f and it generates unique parameters per-device-per-owner. The `getAddress()` function always generates unique addresses, as shown in the `handler.js` file of the transfer-chain-js application in section 6.1.1. The owner is also referenced by their public key, derived from a uniquely generated ECDSA key pair for each owner. The nonce is pseudorandomly generated and added for the freshness of the parameters. These steps ensure that unique parameters are always generated for each device.

2. We changed the `createAsset()` function to include the `getAssetParameters()` function:

```
const createAsset = (asset, owner,  state) =>
  {
  const address = getAssetAddress(asset)
  return state.get([address])
  .then(entries => {
    const entry = entries[address]
    if (entry && entry.length > 0) {
    throw new InvalidTransaction('Device name in use')
    }
    console.log('Device Created. \nDefault Device Parameters: ')
  // new parameters generated for the device
    console.log(getAssetParameters(asset))
    return state.set({
  // this address maps to device and owner
    [address]: encode({name: asset, owner})
    })
  })
}
```

When the device is added, the new parameters are displayed on the console to the current owner of the device. The state gets updated with the new asset and its owner, which can be verified if and when the user wishes to bootstrap the device. We are assuming that ZTB is not performed at this step.

3. Inside the `acceptTransfer()` function, we have included the code for the encrypted transfer of parameters. This is where the eccrypto library [36] is used. This step is performed just before the state change is applied. The parameters are displayed in the console to the latest owner for prototype demonstration.

```
//ephemeral public and private key pair
var privateKeyBuyer = eccrypto.generatePrivate();
```

```
var publicKeyBuyer = eccrypto.getPublic(privateKeyBuyer);

// Encrypting the message for the buyer.
eccrypto.encrypt(publicKeyBuyer, parameters).then(function(encrypted)
{

// Buyer decrypting the message.
eccrypto.decrypt(privateKeyBuyer, encrypted).then(function(plaintext)
{
console.log("Parameters:", plaintext.toString());
 }
}
```

This part generates ephemeral public-private key pairs to encrypt and transfer the unique parameters. Once the next owner of the device receives them, these parameters are decrypted. The ephemeral keys are not stored anywhere, and new ephemeral key pairs will be generated when the device is transferred again.

This encryption function has been placed in the `acceptTransfer()` method so that is included as a part of a valid signed transfer.

4. Next we have defined the function to initiate the broadcast of the bootstrapping network followed by retrieving the proof of ownership of the device. Once the ownership has been proven, we generate a console prompt that says that the bootstrapping network connection has been established.

```
const bootstrap = (asset, signer, state) => {

  const assetAddress = getAssetAddress(asset)
  console.log("Broadcasting Bootstrapping Network")
  return state.get([assetAddress])
    .then(entries => {
      const entry = entries[assetAddress]

    //matches the signature of the record with the device's owner

    if (signer !== decode(entry).owner) {
    throw new InvalidTransaction('Only Device\'s
    owner may bootstrap it')
      }
    console.log("Secure Bootstrapping Network Established");
    })
}
```

5. We made a change in the `app.js` file in the client folder. The `update()` function is meant to submit updates to the `state.js` file to be converted into a signed transaction and updated into the state to be used by handler.js. A "count" variable tracks when

the accept action is invoked from the frontend. This automatically sends a bootstrap action update to the state which executes the `bootstrap()` function in `handler.js`.

```
app.update = function (action, asset, owner) {
    if (this.user) {
        if (count === 1)
        { count = 0 //ZTB once the device is accepted
        submitUpdate(
        { action, asset, owner },
        this.user.private,
        success => success ? app.update('bootstrap', asset) : null
        )
        }
        submitUpdate(
        { action, asset, owner },
        this.user.private,
        success => success ? this.refresh() : null
        )
    }
}
```

6. Minor changes were made to the other files in the client application. These are mainly for changes in the frontend of the webpage and, therefore, are insignificant to our project.

Figure 16 shows the outputs in the console and Figure 17 shows the webpage or the UI of our prototype.

It is important to take the following things into consideration:

- For simplicity, we have not made major changes to the original UI. The user interacting with the client through the frontend can be changed with the drop-down menu that says "Select User". This is because we have to play the roles of different users for testing purposes. This is shown in figure 17.
- Since the devices and their respective owners are available to be viewed publicly on the webpage, the credentials are shown only on the console to show that this information is hidden from the general public.
- All the interactions such as the creation of a device, transfer of a device etc. are displayed on the same console as shown in 16. This is because users can be switched on the same webpage. Additionally, this also helps us compare the device's parameters (to show that they are the same before and after being transferred). In the ideal case, the user should only be able to see, on the console, the interactions made only by them.

```
Initializing JSON handler for Transfer-Chain
Connected to tcp://localhost:4004
Registration of [transfer-chain 0.0 application/json] succeeded
Handling transaction:  create > device4  :: 038e850f...
Device Created.
Default Device Parameters:
d051fb25a986ceab02531253b214c5cae7f9c74e53dba89229db49e21b7f58undefinedyKfv1wuoA8FFSYR4l+XPCQ==
Handling transaction:  create > device4  :: 038e850f...
Device Created.
Default Device Parameters:
d051fb25a986ceab02531253b214c5cae7f9c74e53dba89229db49e21b7f58undefinedc9TK3HFZSJdLKAA9wqz55Q==
Handling transaction:  transfer > device4 > 0233134b...  :: 038e850f...
Initiating transfer
Handling transaction:  transfer > device4 > 0233134b...  :: 038e850f...
Initiating transfer
Handling transaction:  accept > device4  :: 0233134b...
transferring parameters
Parameters: d051fb25a986ceab02531253b214c5cae7f9c74e53dba89229db49e21b7f58undefinedBPqJ9lU7TqjOXU8c8zawEg==
(node:59362) [DEP0005] DeprecationWarning: Buffer() is deprecated due to security and usability issues. Pleas
Handling transaction:  accept > device4  :: 0233134b...
InvalidTransaction: Asset is not being transfered
    at /home/ushana/work/coding/finale/ZTB-Hyperledger-Sawtooth/processor/handlers.js:107:15
Handling transaction:  accept > device4  :: 0233134b...
transferring parameters
Parameters: d051fb25a986ceab02531253b214c5cae7f9c74e53dba89229db49e21b7f58undefined5qD7a5kYLGa4p1XeUv7Tmg==
Handling transaction:  bootstrap > device4  :: 0233134b...
Broadcasting Bootstrapping Network
Secure Bootstrapping Network Established
Handling transaction:  bootstrap > device4  :: 0233134b...
Broadcasting Bootstrapping Network
Secure Bootstrapping Network Established
```

Figure 16: The bootstrapping process displayed in the console.

Zero-Touch Bootstrapping using Hyperledger Sawtooth

Select User

0233134b9ad7ed5460bed59d3fd7683b12350459faf723b475ed9001a20daad8b1

Add Device

device4 Create

Transfer Device

Select device... Select recipient... Transfer

Accept Device

device4 Accept Reject

Device List

Device	Owner
device3	038972110f534ac39af73ad27f9483f0e6a87043fdb93a6d646a5ab34e3c25f1ea
device2	03022cb219437905da7e493e80b49bb675b2977078b7865fb3307b3f3e0751409b
device4	038e850fd9b303a7ce4faffa51403daaa67a2ad2c706ce31af8c6879fa2449af98
device1	0258d53c21d1c9a9258ed2e7359759e0999255cffe871594477681dda784c4ff7a

Figure 17: The user interface webpage in our prototype.

7 Discussion

7.1 Answers to Research Questions

Q1. *How do we create a device-and-user-specific unique network for bootstrapping to prevent misbinding?*

A1. Unique bootstrapping parameters are generated when (i) a new device is added to the blockchain, (ii) when an already bootstrapped device is transferred from its old user to its new user, or (iii) when the current user resets the device. The new parameters are generated by a function f that takes the unique device identity and the unique identity of its next user as inputs. The function f also includes a nonce to generate the output to avoid replay attacks. Previously generated parameters cannot be used to connect to the bootstrapping network.

Q2. *How do we share parameters for this device-and-user-specific bootstrapping network between the IoT and bootstrapping devices?*

A2. Once the new user accepts a device transfer, the function f generates fresh bootstrapping parameters, encrypts them using the new user's ephemeral public key (based on the ECIES encryption algorithm), and transfers them to the new user. In our implementation, the function f is embedded in the smart contract (transaction processor). However, it does not matter how the function is implemented – whether the parameters are generated inside or outside the IoT device – as long as the parameters are included in the smart contract logic. The new user receives the encrypted parameters only after a valid transfer of the device that has been agreed upon by the blockchain nodes based on the chosen blockchain consensus protocol.

Q3. *How do the endpoints (IoT device and bootstrapping device) mutually authenticate each other before the bootstrapping of the IoT device can begin?*

A3. The mutual authentication is done in two steps. First, only the new owner can decrypt the bootstrapping parameters using their ephemeral private key. Using these parameters, the bootstrapping device and the IoT device establish a connection, after which the device sends a request for proof of ownership to the bootstrapping device. The bootstrapping device forwards this request to the validators on the blockchain. The validators generate an output that includes the public key of the latest owner of the blockchain. This output is signed with the user's private key in the bootstrapping device. Upon receiving this public key, the IoT device uses it to match the signature on the output. These two steps allow the IoT and bootstrapping devices to authenticate each other mutually.

Q4. *Can the bootstrapping process continue to be zero-touch even after applying all these extra layers of security?*

A4. Yes. If the bootstrapping device acts as the access point (AP), it can start broadcasting its bootstrapping network once its user accepts an IoT device transfer. The bootstrapping network connection can be established when the IoT device is brought into its vicinity and powered on. Once the connection is established, any ZTB protocol can configure the IoT device.

Q5. *Can this process be applied to any IoT device regardless of its manufacturer?*

A5. Yes, as mentioned before, the procedure is not vendor-specific. However, based on the implementation, the IoT device or the bootstrapping device might need to have capabilities to perform public key cryptography. This capability is required if either device has to (i) decrypt the bootstrapping network parameters, and (ii) check the digital signature of its latest user.

7.2 Drawbacks of our solution

Similar to other ZT bootstrapping methods, our solution requires an internet connection during the bootstrapping process. Specifically, IoT device ownership can only be verified when the bootstrapping device is connected to the internet.

The IoT device needs to have the capabilities required to perform public-key cryptography as well as cryptographic key storage. An IoT device with a TPM or a TEE would be ideal for this purpose however the hardware aspect has not been researched in depth in this thesis.

Our implementation has not considered the case where the ZTB feature fails. Just like Amazon's FFS has a BCS option in case ZTS fails, our design should also have such a "fallback" bootstrapping option. Regardless of which bootstrapping method is used, the process should still be able to generate unique bootstrapping connections per-device-per-user that satisfy all the cryptographic primitives specified in Section 5.2.

Our solution relies on a deployed blockchain-based network infrastructure. However, several large-scale blockchain-based projects are currently under development, as mentioned in Section 4.1. Availability of such infrastructures may facilitate the adoption of our solution.

Our solution needs some form of standardization for interoperability across different platforms and devices. Standardization efforts could contribute to seamless integration and enhance the compatibility of our solution with existing IoT ecosystems.

Our solution, currently, does not protect against malicious IoT devices from joining the user's home network. For instance, if an attacker modifies an IoT device before it reaches the end user, the compromised device can still undergo bootstrapping and join the user's home network. To address this threat, we recommend integrating device attestation into the solution. Device attestation would verify the integrity and authenticity of IoT devices during the bootstrapping procedure, mitigating the risk of compromised devices.

7.3 Limitations in the implementation

7.3.1 Challenges faced

The main issues that were encountered while designing the prototype revolved around the Sawtooth framework. The developer documentation was incomplete in several places. I personally felt that the documentation available at this time was not clear enough, which took me extra time to understand the framework's design. Furthermore, the current versions of the available SDKs for application development are outdated, and several technical errors were encountered, hindering the time to finish the project.

Nevertheless, as we had previously discussed in Section 4.4.2, Sawtooth has an edge over other frameworks for our particular project. Therefore, even though there are other

frameworks and blockchains that could be used for this implementation, our project was built on Sawtooth to serve as a foundation and precursor for more upcoming projects to be developed on Sawtooth and, consequently, better resources for its use.

7.3.2 Constraints in the prototype

Our current implementation primarily focuses on the backend, specifically the network infrastructure aspect of ZTB. To create a more comprehensive solution, our implementation could be extended to include an actual IoT device and bootstrapping device, thus, enabling end-to-end testing and validation of the bootstrapping procedure.

In our implementation, anyone can add a new device to the blockchain as well as bootstrap it. However, in real-world scenarios, only authorized participants, such as manufacturers, should have the privilege to add new devices to the chain and only end users should be able to bootstrap them. Addressing this limitation would involve implementing access controls and authentication mechanisms.

In cases where a user resets their device, new parameters need to be generated for bootstrapping it. Although this feature has not been implemented in our current solution, it can be included as a future enhancement to ensure the accuracy and integrity of device information within the system.

7.4 Scope for future work

Our solution could be extended to include attestation protocols involving blockchain-based verification. We have stated in Section 5 that the established secure connection between the IoT device and the bootstrapping device, along with the proof of ownership, could produce an output that any ZTB solution provider could use to supply the IoT device with the credentials that it needs to connect to its designated network. This proof could be formally verified to serve as an attestation protocol.

The implementation can be improved overall for a production-grade application by adding more comprehensive handling of error situations, like dropped messages.

Future work can also include crypto transactions since we already use a blockchain in our implementation. In other words, buyers and sellers can trade IoT devices for crypto-currencies. This could also involve comparing the performance of the design with other Hyperledger frameworks or other blockchains in general.

From an environmental perspective, support for recycling at the end of a device's lifecycle could be added. It can be implemented by enabling seamless change of ownership without the need for bootstrapping parameters and including valid recycling stations as participating nodes in the blockchain.

Finally, ongoing research and development efforts should focus on addressing the limitations identified in this study to refine further and optimize our solution.

7.5 Conclusion

In Zero-Touch Bootstrapping (ZTB), the user powers on a new IoT device in the vicinity of a bootstrapping device such as a smart speaker or a mobile phone. The IoT device then automatically receives the credentials of its user's network from a cloud-based backend

to connect to its user's local network. For example, in Amazon's Frustration-Free Setup (FFS) [5], this information is supplied to the IoT device by an Amazon Echo smart speaker that belongs to the owner of the IoT device. ZTB is highly effective in terms of usability and security at home and in large infrastructures.

One of the issues in ZTB, however, is the threat of misbinding, where an IoT device is vulnerable to getting connected to the wrong endpoint and ZTB cannot be performed as intended. Furthermore, current ZTB solutions are vendor-specific, which means that the IoT device and the bootstrapping device must be manufactured or certified by the same ZTB solution provider. In most cases, IoT and bootstrapping devices are bought from different sources, making vendor-specific bootstrapping solutions infeasible.

We have designed a novel blockchain-based prototype using Hyperledger Sawtooth [41] to address these issues. Here, the device transfers are recorded immutably on a blockchain, and at any given time, the IoT device and its current owner can be verified from the blockchain. The system can generate and securely transfer unique bootstrapping parameters for an IoT device to establish a connection only with its intended bootstrapping device, thus preventing misbinding. Furthermore, establishing this secure connection, combined with proof of ownership obtained from the blockchain, can be used by a ZTB solution provider to perform ZTB on any IoT device regardless of its source.

This project creates scope for future work in IoT device management. Furthermore, as of today, several projects are developing blockchain-based systems for use cases in IoT. These include tamper-evident logs (Google Trillian) [82], valid certificate lists (Certificate Transparency) [20], and IETF SCITT [35]. Our project, thus, also adds to this list of use cases that leverage blockchain technology for IoT solutions.

References

[1] Binance Academy. *Byzantine Fault Tolerance Explained*. Binance Academy. URL: https://academy.binance.com/en/articles/byzantine-fault-tolerance-explained.

[2] Ronald Rivest Adi Shamir and Len Adleman. "A method for obtaining digital signatures and public-key cryptosystems." In: *Comm. ACM* (1978).

[3] Tolgahan Akgün. "Secure Lifecycle Management for Internet of Things Devices". Master's thesis, Master of Science in Technology. MA thesis. Aalto University, 2020.

[4] Amazon. *Amazon Managed Blockchain*. https://aws.amazon.com/managed-blockchain/.

[5] Amazon. *Understanding Frustration-Free Setup*. https://developer.amazon.com/docs/frustration-free-setup/understanding-ffs.html.

[6] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018.

[7] ARM. *TrustZone for Cortex-M*. URL: https://www.arm.com/technologies/trustzone-for-cortex-m.

[8] AWS. *Start Building on AWS Today*. URL: https://aws.amazon.com/.

[9] Seyed Mojtaba Hosseini Bamakan, Amirhossein Motavali, and Alireza Babaei Bondarti. "A survey of blockchain consensus algorithms performance evaluation criteria". In: *Expert Systems with Applications* 154 (2020). ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113385. URL: https://www.sciencedirect.com/science/article/pii/S0957417420302098.

[10] Ushana Bandyopadhyay. *Secure Zero-Touch Bootstrapping of IoT Devices using Hyperledger Sawtooth*. GitHub repository. 2023. URL: https://github.com/UBgood/ZTB-Hyperledger-Sawtooth.

[11] Elaine Barker. *Recommendation for Digital Signature Timeliness*. Special Publication 800-102. Computer Security Division, Information Technology Laboratory: National Institute of Standards and Technology, Sept. 2009.

[12] Imran Bashir. *Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications*. Packt Publishing, 2018.

[13] Sebastien Boire. "Credential Provisioning and Peer Configuration with Extensible Authentication Protocol: Secure Configuration and Management of Connected Devices". Master's thesis, Master's Programme in Security and Cloud Computing. MA thesis. Aalto University, 2021.

[14] Amel Bourdoucen. "Securing Communication Channels in IoT using an Android Smart Phone". Master of Science (Tech.) Thesis. University of Turku, 2020.

[15] Mic Bowman et al. "On Elapsed Time Consensus Protocols". In: *Progress in Cryptology–INDOCRYPT 2021: 22nd International Conference on Cryptology in India, Jaipur, India, December 12–15, 2021, Proceedings 22*. Springer. 2021, pp. 559–583.

[16] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[17] Protocol Buffers. *Protocol Buffers*. URL: https://protobuf.dev/.

[18] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. castro,liskov@lcs.mit.edu. Laboratory for Computer Science, Massachusetts Institute of Technology. New Orleans, USA, Feb. 1999.

[19] Certicom Research. *Standards for Efficient Cryptography 2 (SEC 2): Recommended Elliptic Curve Domain Parameters*. Certicom Corp. Version 2.0. Contact: Daniel R. L. Brown (dbrown@certicom.com): Certicom Research, Jan. 2010.

[20] *Certificate Transparency*. https://certificate.transparency.dev/.

[21] Sourabh Chandra et al. "A comparative survey of Symmetric and Asymmetric Key Cryptography". In: Nov. 2014. DOI: 10.1109/ICECCE.2014.7086640.

[22] Consensys. *Quorum*. 2022. URL: https://consensys.net/quorum/.

[23] Zach Delventhal. *Transfer Chain JS*. GitHub repository. URL: https://github.com/delventhalz/transfer-chain-js.

[24] Rescorla Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC8446. URL: https://www.ietf.org/rfc/rfc5246.txt.

[25] Whitfield Diffie and Martin Hellman. "An Efficient Key Exchange Protocol". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[26] Danny Dolev and Andrew Yao. "On the Security of Public Key Protocols". In: *Transactions on Information Theory* 29.2 (1983), pp. 198–208.

[27] Dragonchain. *Dragonchain*. URL: https://dragonchain.com/.

[28] Daniel Drescher. *Blockchain Basics: A Non-Technical Introduction in 25 Steps*. Apress, 2017.

[29] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. Federal Inf. Process. Stds. (NIST FIPS) 197. Accessed: July 12, 2023. Gaithersburg, MD: National Institute of Standards and Technology, 2001. URL: https://doi.org/10.6028/NIST.FIPS.197.

[30] Energy Web. *Energy Web*. URL: https://www.energyweb.org/.

[31] Md Sadek Ferdous et al. *Blockchain Consensus Algorithms: A Survey*. 2020. arXiv: 2001.07091 [cs.DC].

[32] Russell A. Fink et al. "Catching the Cuckoo: Verifying TPM Proximity Using a Quote Timing Side-channel". In: *Proceedings of the International Conference on Trust and Trustworthy Computing*. Springer-Verlag. Berlin, Heidelberg, 2011, pp. 294–301.

[33] Owen Friel et al. "ZERO-TOUCH BOOTSTRAP OF A NETWORK CONNECTED DEVICE". In: *Technical Disclosure Commons* (Jan. 2019). URL: URL%20of%20the%20Document.

[34] Taher El Gamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *CRYPTO 84 on Advances in Cryptology*. Springer-Verlag New York, Inc., 1985, pp. 10–18.

[35] Henk Birkholz, Antoine Delignat-Lavaud, Cedric Fournet, Yogesh Deshpande. *An Architecture for Trustworthy and Transparent Digital Supply Chains*. https://datatracker.ietf.org/doc/draft-ietf-scitt-architecture/. Accessed: July 7, 2023. 2023.

[36] Kagami Hiiragi. *eccrypto*. GitHub repository. URL: https://github.com/bitchan/eccrypto.

[37] Hyperledger. *Hyperledger Iroha Security Audit Results*. Feb. 2019. URL: https://www.hyperledger.org/blog/2019/02/04/hyperledger-iroha-security-audit-results.

[38] Hyperledger. *Use Cases*. Hyperledger. Accessed May 19, 2023. URL: https://www.hyperledger.org/use.

[39] *Hyperledger Besu Consensus protocols*. Accessed: 2023-07-14. URL: https://besu.hyperledger.org/23.4.0/private-networks/how-to/configure/consensus.

[40] Hyperledger Foundation. *Hyperledger Foundation*. 2021. URL: https://www.hyperledger.org/.

[41] Hyperledger Foundation. *Hyperledger Sawtooth*. URL: https://sawtooth.hyperledger.org/.

[42] *Hyperledger Sawtooth REST API Docker Image*. https://hub.docker.com/r/hyperledger/sawtooth-rest_api.

[43] *Hyperledger Sawtooth Setting Transaction Processor Docker Image*. https://hub.docker.com/r/hyperledger/sawtooth-tp_settings.

[44] *Hyperledger Sawtooth Shell Docker Image*. https://hub.docker.com/r/hyperledger/sawtooth-shell.

[45] *Hyperledger Sawtooth v 1.2 System Administrator’s Guide – About dynamic consensus*. Accessed: 2023-03-09. URL: https://sawtooth.hyperledger.org/docs/1.2/sysadmin_guide/about_dynamic_consensus.html.

[46] *Hyperledger Sawtooth Validator Docker Image*. https://hub.docker.com/r/hyperledger/sawtooth-validator.

[47] IBM. *Blockchain*. IBM. URL: https://www.ibm.com/topics/blockchain.

[48] IBM. *IBM Supply Chain Intelligence Suite - Food Trust*. URL: https://www.ibm.com/products/supply-chain-intelligence-suite/food-trust.

[49] IETF Network Working Group. *RFC 5289: Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)*. https://datatracker.ietf.org/doc/html/rfc5289. RFC, Informational. Aug. 2008.

[50] Intel. *Intel Secure Device Onboard*. https://www.intel.com/content/dam/www/public/us/en/documents/idz/iot/briefs/sdo-product-brief.pdf.

[51] Intel. *Intel Software Guard Extensions (Intel SGX)*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[52] *Introduction to Trusted Execution Environments*. GlobalPlatform, May 2018.

[53] *IPFS*. Accessed: 2023-07-18. URL: https://ipfs.tech/.

[54] Ira Ray Jenkins and Sean W. Smith. *Distributed IoT Attestation via Blockchain*. Tech. rep. {jenkins, sws}@cs.dartmouth.edu. Hanover, New Hampshire, USA: Dartmouth College, 2020.

[55] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Journal of Cryptology* 14.3 (2001). Published online: July 27, 2001, pp. 209–218. DOI: 10.1007/s00145-001-0002-3.

[56] KORE. *How to Manage (And Even Thrive In) The Google Cloud Platform Migration*. https://www.korewireless.com/news/how-to-manage-and-even-thrive-in-the-google-cloud-platform-migration.

[57] Hugo Krawczyk. "SIGMA: the 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols". In: (June 2003). Technical Report.

[58] Kuruva Lakshmanna and et. al. Shafiq. "The Rise of "Internet of Things": Review and Open Research Issues Related to Detection and Prevention of IoT-Based Security Attacks". In: *Wireless Communications and Mobile Computing* (2022). ISSN: 1530-8669. DOI: 10.1155/2022/8669348. URL: https://doi.org/10.1155/2022/8669348.

[59] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* (July 1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/.

[60] Somayya Madakam and et. al. Ramaswamy. "Internet of Things (IoT): A Literature Review". In: *Journal of Computer and Communications* 3 (2015), pp. 164–173. DOI: 10.4236/jcc.2015.35021. URL: http://www.scirp.org/journal/jcc.

[61] Marco Polo Network. *Marco Polo Network*. URL: https://marcopolonetwork.com/.

[62] Martin Giles, MIT Technology Review. "Triton is the world's most murderous malware, and it's spreading". In: (Mar. 2019).

[63] Víctor Gayoso Martínez, Luis Hernández Encinas, and Carmen Sánchez Ávila. "A Survey of the Elliptic Curve Integrated Encryption Scheme". In: (2010).

[64] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[65] Microsoft. *What is Azure IoT Hub Device Provisioning Service?* https://learn.microsoft.com/en-us/azure/iot-dps/about-iot-dps.

[66] Fedor Muratov et al. "YAC: BFT Consensus Algorithm for Blockchain". In: (Sept. 2018). fyodor, andrei, nikolai, bulat, takemiya@soramitsu.co.jp. arXiv: 1809.00554v1 [cs.DC]. URL: https://arxiv.org/abs/1809.00554v1.

[67] Mustafa Bedawala. *A comprehensive overview of enterprise blockchain*. https://usa.visa.com/solutions/crypto/enterprise-blockchain.html. Accessed: June 23, 2023. 2023.

[68] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. Accessed: Month Day, Year. 2008.

[69] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 7539. May 2015. DOI: 10.17487/RFC7539. URL: https://www.rfc-editor.org/info/rfc7539.

[70] NIST - Information Technology Laboratory, Computer Security Resource Center. *roots of trust*. https://csrc.nist.gov/glossary/term/roots_of_trust. Accessed: Jul 14, 2023.

[71] Aleksi Peltonen et al. "Enterprise Security for the Internet of Things (IoT): Lightweight Bootstrapping with EAP-NOOB". In: *Sensors* 19.22 (2019). ISSN: 1424-8220.

[72] *PostgreSQL*. Accessed: 2023-07-26. URL: https://www.postgresql.org/.

[73] r3. *Corda*. URL: https://corda.net/.

[74] Lukas Reinfurt et al. "Internet of Things Patterns for Device Bootstrapping and Registration". In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPLoP '17. Irsee, Germany: Association for Computing Machinery, 2017. ISBN: 9781450348485. DOI: 10.1145/3147704.3147721. URL: https://doi.org/10.1145/3147704.3147721.

[75] Nishant Sapra, Imlak Shaikh, and Ashutosh Dash. "Impact of Proof of Work (PoW)-Based Blockchain Applications on the Environment: A Systematic Review and Research Agenda". In: *Journal of Risk and Financial Management* 16.4 (2023). ISSN: 1911-8074. URL: https://www.mdpi.com/1911-8074/16/4/218.

[76] *Sawtooth Architecture Guide*. Accessed: 2023-06-09. URL: https://sawtooth.hyperledger.org/docs/1.2/architecture/.

[77] Manoj Sethi. "Security for Ubiquitous Internet-Connected Smart Objects". Ph.D. Thesis. Espoo, Finland: Aalto University, 2016.

[78] Mohit Sethi, Aleksi Peltonen, and Tuomas Aura. "Misbinding Attacks on Secure Device Pairing and Bootstrapping". In: *Proceedings of the 14th ACM ASIA Conference on Computer and Communications Security (AsiaCCS '19)*. NomadicLab, Ericsson Research, Finland and Aalto University, Finland. Auckland, New Zealand: ACM, July 2019, pp. 123–134.

[79] Jonas S. Søgaard et al. "Improving Trust in a (Trans)National Invoicing System: The Performance of Crash vs. Byzantine Fault Tolerance at Scale". In: *Applied Sciences* 13.12 (2023). ISSN: 2076-3417. DOI: 10.3390/app13126941. URL: https://www.mdpi.com/2076-3417/13/12/6941.

[80] William Stallings. *Cryptography and Network Security: Principles and Practice (7th Edition)*. Pearson, 2017.

[81] The Editors of Encyclopaedia Britannica. *telemetry*. https://www.britannica.com/technology/telemetry. Encyclopedia Britannica, Accessed: July 17, 2023.

[82] *Trillian*. https://transparency.dev/.

[83] *Trusted Platform Module (TPM) Summary*. Accessed: 2023-07-20. URL: https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/.

[84] Trustonic. *Establishing a Root of Trust with Digital Holograms*. 2018. URL: https://www.trustonic.com/opinion/digital-holograms-2/.

[85] Lionel Sujay Vailshery. *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts to 2030*. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. Nov. 2022.

[86] D. Wallner, E. Harder, and R. Agee. "A Secure Key Exchange Protocol for Real-Time Applications". In: *ACM SIGCOMM Computer Communication Review* 29.3 (1999), pp. 17–26.

[87] Kent Watsen, Mikael Abrahamsson, and Ian Farrer. *Secure Zero Touch Provisioning (SZTP)*. RFC 8572. Apr. 2019. DOI: 10.17487/RFC8572. URL: https://www.rfc-editor.org/info/rfc8572.

[88] *What is EBSI?* Accessed: 2023-07-18. URL: https://ec.europa.eu/digital-building-blocks/wikis/display/EBSI/What+is+ebsi#how-it-works.