Journal of
Sensor and Actuator Networks

Article

A Zero-Trust Multi-Processor Reporter-Verifier Design of Edge Devices for Firmware Authenticity in Internet of Things and Blockchain Applications

Ananda Maiti [1,*] and Alexander A. Kist [2]

[1] School of IT, Deakin University, Waurn Ponds, VIC 3216, Australia
[2] School of Electrical Engineering, University of Southern Queensland, Toowoomba, QLD 4350, Australia; kist@ieee.org
* Correspondence: anandamaiti@ieee.org

Abstract: Firmware authenticity and integrity during upgrades are critical security factors in Internet of Things (IoT) applications in the age of edge artificial intelligence (AI). Data from IoT applications are vital for business decisions. Any unintended or malicious change in data can adversely impact the goals of an IoT application. Several studies have focused on using blockchain to ensure the authentication of IoT devices and the integrity of data once the data are in the blockchain. Firmware upgrades on IoT edge devices have also been investigated with blockchain applications, with a focus on eliminating external threats during firmware upgrades on IoT devices. In this paper, we propose a new IoT device design that works against internal threats by preventing malicious codes from device manufacturers. In IoT applications that monitor critical data, it is important to ensure that the correct firmware reporting honest data is running on the devices. As devices are owned and operated by a small group of application stakeholders, this multiprocessor design extracts the firmware periodically and checks whether it matches the signatures of the expected firmware designed for the business goals of the IoT applications. The test results show that there is no significant increase in code, disruption, or power consumption when implementing such a device. This scheme provides a hardware-oriented solution utilizing processor-to-processor communication protocols and is an alternative to running lightweight blockchain on IoT edge devices.

Keywords: internet of things; blockchain; firmware; ISCP; microprocessor; hash; edge AI

Academic Editor: Jordi Mongay Batalla

Received: 18 February 2025
Revised: 17 March 2025
Accepted: 24 March 2025
Published: 31 March 2025

Citation: Maiti, A.; Kist, A.A. A Zero-Trust Multi-Processor Reporter-Verifier Design of Edge Devices for Firmware Authenticity in Internet of Things and Blockchain Applications. *J. Sens. Actuator Netw.* **2025**, *14*, 35. https://doi.org/10.3390/jsan14020035

1. Introduction

Internet of Things (IoT) applications rely on a network of edge devices, including sensing devices, to collect a significant amount of data. The integrity and authenticity of these data are critical to the success of the application goals. Edge devices are typically owned and operated by a small group of business stakeholders in IoT applications. Devices may also be shared across multiple applications. Edge devices run *firmware*, an embedded program that collects and sends data from the edge to the fog or cloud. Edge firmware is periodically updated to add new features. This typically happens over the air or sometimes via physical wiring. Once updated, the device starts to execute the latest firmware code indefinitely. This makes firmware authenticity important for the continued safe operation of the IoT system [1–3]. This importance is growing in the era of edge AI [4], in which edge devices are less observed by humans and given more autonomy. Thus, their decisions and data become more valuable.

Several security threats to IoT have been identified [5,6], and corresponding solutions have been developed with regard to identity management and encryption. Several measures have been taken to safeguard the data generated by IoT, such as blockchain [7]. IoT edge devices are kept safe by employing security features such as secure bootloading. These measures are effective in protecting against external security threats. In this paper, we address security threats from the developers of programs and devices. Firmware that is developed and signed by authorized stakeholders must pass every security measure. If this firmware falsifies data, it is hard to detect, and these data could impact subsequent applications that use the data. Thus, it is important for all users of IoT data to be sure that the firmware running on the edge devices adheres to the IoT application goals. The data reported by this firmware are continuous. Thus, if a few pieces of data are occasionally falsified by faulty or malicious firmware, it is nearly impossible to detect until it is too late.

This paper's key contribution is an IoT device design architecture that uses multiple processors, where one verifies the program being executed on the other. This architecture depends on the processor's ability to access another processor's flash memory. It is based on a zero-trust model of design solutions for multi-user applications. Blockchain acts as a supporting database to store the expected firmware and corresponding hashed values. Blockchain provides immutability of these data in a multi-user environment.

The rest of the paper is organized as follows. Section 2 discusses related work regarding blockchain, IoT, and firmware. Section 3 presents the new zero-trust multi-processor architecture, and Section 4 presents the performance results of corresponding devices. Section 5 discusses the advantages and limitations of the proposed architecture.

2. Related Work

This section discusses the current state of blockchain, IoT applications involving blockchain, and edge device design, with a particular focus on the authenticity of firmware.

2.1. Blockchain in Internet of Things

The combination of blockchain and the Internet of Things (IoT) represents a significant technological advancement with regard to automation and heterogeneity. It can revolutionize various industries by enhancing security and transparency.

2.1.1. Blockchain-IoT Applications

Blockchain is a decentralized ledger technology that ensures data integrity and security through cryptographic hashing and consensus mechanisms. Each block in the chain contains an immutable and transparent record of transactions, making blockchain an ideal solution for secure data management. The benefits of integrating blockchain with IoT are as follows:

- *Enhanced Security*: Blockchain's decentralized nature and cryptographic techniques provide robust security for IoT devices and data. Each transaction or data exchange is recorded in an immutable ledger, reducing the risk of unauthorized access and data breaches.
- *Data Integrity* and *Immutability*: Blockchain ensures that the data recorded from IoT devices are tamper-proof and verifiable. This immutability is crucial for applications in which data accuracy and trust are paramount.
- *Transparency* and *Traceability*: Blockchain's transparent ledger allows all participants in the IoT ecosystem to verify transactions and data exchanges. This transparency enhances trust and accountability, particularly in supply chain and logistics applications.
- *Decentralization*: By eliminating the need for a central authority, blockchain reduces single points of failure and enhances the resilience of IoT networks.

Figure 1 depicts the general architecture of blockchain IoT applications. There are multiple stakeholders in the system. A set of dedicated stakeholders provides the IoT devices and the corresponding software that runs on the hardware. This stakeholder may or may not build the actual devices, but is responsible for the software that is currently running on them. The device is meant to collect data from the environment and send it to the blockchain, which then forwards them to the relevant clients of the data. As the data are in the blockchain, there is a guarantee that the data are not manipulated. Also, *smart contracts*, which run within the blockchain, automate the data processing and take action automatically. The smart contracts and notification procedures are known to all stakeholders, and as everything is in blockchain, any alterations can happen only with the knowledge of all stakeholders.

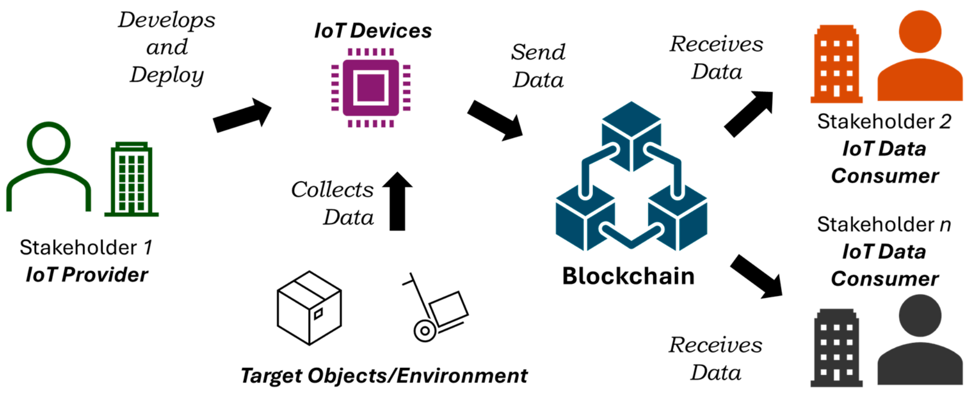

Figure 1. A typical blockchain-based IoT application.

2.1.2. Shortcomings with Regard to Firmware Authenticity

The key shortcoming of blockchain with IoT design is the device itself. A normal IoT edge device can be compromised, i.e., the code running on the device may be altered to send fake data that pretend to be in compliance with IoT applications. For example, if the sensor is supposed to represent an environmental parameter such as temperature, which is supposed to be within a limit, there is a possibility and incentive for some stakeholders to compromise the device's code to report the temperature falsely. As they own and operate the devices, it is possible for them to put in malicious code without being detected as an intrusion event or bypassing any security protocols. The weakness of the blockchain–IoT combination is that the data are immutable and indisputable once the data enter the blockchain. Still, there is no definite way to ensure that the devices' reported data were correct to begin with.

There have been recent attempts to run light with blockchain from within an edge device [8–22]. However, running a blockchain is complicated and power-intensive. Moreover, edge devices are often required to sleep for an extended period of time, making it difficult to maintain a blockchain.

IoT edge devices have firmware that can be esoteric, i.e., the source code is meant to be kept a trade secret. However, the binary code compiled from it cannot be converted back to the source code. Trusted Execution Environment (TEE) aims to provide a secure onboard area when the device is under the attacker's control [1]. ChkUp [2] is a program analyzer that can resolve the program execution paths during firmware updates using cross-language inter-process control flow analysis and program slicing. RASSIFAB [3] is another attempt to ensure that firmware updates are authentic and auditable. All these measures are taken against external attacks on devices or the IoT system, but they do not protect against a threat from a genuine stakeholder.

2.1.3. Zero Trust Security Model

This model is based on the principle of "trust no one, always verify", which involves rigorous verification of every device and user, regardless of whether they are inside or outside the corporate network [23]. In a zero-trust model, every access request is authenticated and continuously verified. This approach helps to eliminate the risk of compromising systems by ensuring that access is constantly monitored and checked.

2.2. Internet of Things Edge Devices

The IoT devices that feed data to the blockchain are designed for a single dedicated purpose. There are various types of security threats to the devices and methods to detect and prevent them from happening. We primarily focus on the security and authenticity of firmware on the edge device.

2.2.1. Detecting Security Threats in IoT Edge

From an IoT application perspective, security threats are detected by mainly observing individual devices that are registered with the system [24]. The key signs and methods to check for potential compromises in edge devices are:

- *Unexpected Device Behavior*: If the device starts acting strangely, such as turning on or off without user input, changing settings spontaneously, or performing tasks outside its normal function, it might be compromised [25].
- *Increased Bandwidth Usage*: A sudden spike in bandwidth usage can indicate that the device is being used as part of a botnet or for other malicious activities [26].
- *Unexplained Data Traffic*: Monitoring network traffic for unusual data being sent to unfamiliar network addresses can help identify compromised devices [26].
- *Slow Device Performance*: Compromised devices often exhibit poor performance, such as responding slowly or freezing frequently [27].
- *Unusual Account Activity*: Check for unauthorized login attempts, password changes, or unknown devices accessing accounts linked to the IoT device [28].
- *Disabled Security Features*: If security features like firewalls or antivirus software are disabled without explanation, it could be a sign of a hack [29].
- *New and Unknown Software Installations*: Unrecognized software on the device can indicate malware or unauthorized applications [29].

Security applications can monitor these issues and detect or prevent the edge devices from being compromised. However, the code that runs on low-end devices may be compromised during updates. Low-cost devices may not exhibit some of the above characteristics, such as slowness or disruptive network behavior, as they may already have small-scale operation and capability.

The biggest problem with detecting internal security threats is *false positive data*. Most efforts are designed to detect deviation from normal data, which are benchmarked for the application. If the devices falsely report the expected data, there is usually no reason to suspect them, and hence, no additional checks can be initiated.

2.2.2. Authenticity of Edge Firmware

Several studies have looked into the issue of proof-of-authenticity within blockchain. A combination of hardware-restricted identification and blockchain-based smart contracts has been proposed in [30–32]. This work prevents unauthorized devices from entering and operating within an IoT blockchain system. Another work proposes an architecture where an embedded device requests firmware updates to the blockchain [32]. It then determines whether its firmware needs an update based on the blockchain response. This guarantees that the embedded device's firmware is up-to-date and not tampered with. Once again, none of these acts against a willful malicious code from the stakeholders that does not break the IoT system itself but falsifies the data reported.

Security threats to IoT firmware have been recognized for various issues [33]. In [34], anti-tampering is proposed to prevent firmware from being corrupted by external entities in the device supply chain. Sophisticated Hashing algorithms have been created to ensure only authentic updates are put into the target system with the help of blockchain, providing data immutability [35]. However, these focus on preserving the authenticity of the firmware and protecting against external threats.

The authenticity of firmware ensures that the code running on the edge device is precisely the same as that developed by the IoT developer. Currently, the best practices to verify firmware authenticity are:

- *Digital Signatures*: The manufacturer digitally signs the firmware. This cryptographic signature verifies that the firmware has not been tampered with and is from a trusted source.
- *Secure Boot*: Implementing a secure boot process can verify the integrity of the firmware before the device boots up. This ensures that only authorized firmware runs on the device.

Firmware can also be corrupted while being updated over the network [36]. It is common to upgrade firmware on an ongoing basis for hardware that performs advanced tasks managing multiple sensors and actuators. There are some measures to ensure that the correct firmware is uploaded every time [37]:

- *End-to-end Encryption*: Ensuring that firmware updates are delivered through encrypted channels to prevent tampering during transmission.
- *Multi-Factor Authentication* (MFA): MFA is used for firmware updates to add an additional layer of security, ensuring that only authorized personnel can update the firmware.

There are also safety measures with human practices. These measures mitigate the faults or malicious intents of human software developers and managers:

- *Trusted Supply Chain*: Establish a trusted supply chain to ensure that the firmware is authentic from the point of manufacture to deployment.
- *Regular Audits*: Regular and thorough code reviews by independent human teams can help identify malicious code or vulnerabilities introduced by developers.

All these measures are key to preventing or reducing the possibility of corrupt code being executed on a device. However, if edge developers or anyone in the supply chain themselves introduce malicious code or vulnerabilities into IoT firmware, it can be challenging to detect and prevent. A few passive strategies that can help mitigate this risk are:

- *Segregation of Duties*: Implementing segregation of duties ensures that no single developer has control over the entire development and deployment process.
- *Transparency and Accountability*: Maintain transparency in the development process and hold developers accountable for their work through logging and monitoring.
- *Security Training*: Provide regular security training to developers to ensure they understand the importance of secure coding practices and the potential consequences of introducing vulnerabilities.

2.2.3. Secure Bootloader

Secure bootloading [38] is a critical security feature designed to ensure that a device boots using only software trusted by the manufacturer. It is essential for maintaining the security and integrity of modern computing devices, protecting them from various types of malware and unauthorized access. Secure bootloaders are part of a chain of trust, which means each stage of the boot process verifies the integrity and authenticity of the next stage before executing it. They authenticate new software images and verify their integrity using cryptographic algorithms. This helps prevent unauthorized or malicious code from running during the boot process. However, secure bootloaders cannot prevent malicious code originating from the manufacturers themselves.

2.2.4. Software Authenticity and Signatures

Well-established methods exist to verify the authenticity of software on an ongoing basis, such as the app stores provided by big tech platforms, e.g., the Google Play Store. They use several methods to verify the safety and authenticity of software before allowing it to be downloaded.

App stores have a built-in security feature that continuously scans apps for harmful behavior [39]. Before an app is published on the app store, it undergoes a manual review process to ensure compliance. This includes checking for malware, inappropriate content, and adherence to privacy policies. Developers must sign their apps with a digital certificate. This ensures that the app has not been tampered with since the developer signed it. App stores use a combination of automated tools and manual reviews to test apps for security vulnerabilities and performance issues. App stores also rely on user feedback and reports to identify and remove malicious or problematic apps. Users can report apps that they believe violate policies or pose security risks.

By implementing these practices, organizations can reduce the risk of developers' malicious actions and ensure the security and integrity of mobile apps, but such app stores, user feedback, and similar infrastructure are not possible or available on low-cost *edge devices*. Moreover, all these methods would have limited success rates due to one or more of the following:

- Codes must be *manually* checked, which is time-consuming and challenging. One stakeholder generates IoT edge data, but it is relevant to multiple other stakeholders. Each one wants the data to be actual. However, the owner can put any malicious code in the device, misreporting the data even with a perfect digital signature and secure boot loading.
- IoT Edge often reports only numeric or *smaller* data. Observing any deviation from normal behavior is already challenging, as the only thing coming from the device is a set of numbers or characters. It is even more complicated if the device reports normal data when the actual data are bad. There is usually no other interface that can be observed for performances.
- They work in retrospect, i.e., if a problem occurs, it can be traced back to the source of the fault. However, again, false normal data are more challenging to detect digitally. They can only be detected when the actual environments or products face actual damage when it is too late.
- They are focused on the network transfer but not on the actual code's execution.

A new architecture based on hardware design for embedded devices supported by embedded software and cloud-based blockchain could solve these problems.

2.3. Embedded Systems Memory and Data Protocols

The necessary embedded system elements are the structure and nature of flash memory and protocols for reading the memory.

2.3.1. Flash Memory

Flash memory stores the microcontroller's firmware, which includes the program code and any static data [40]. Flash memory allows for firmware updates, enabling the microcontroller to be reprogrammed without replacing the hardware. Flash memory retains data for an extended period, typically 10 years or more, making it reliable for storing critical information. Flash memory has a limited number of write/erase cycles, typically ranging from 10,000 to 100,000 cycles. This means that over time, the memory can wear out if frequently written to. Despite the limit in the write-cycle, it is more than sufficient for occasional reading by the verifier.

2.3.2. In-Circuit Serial Programming

In-Circuit Serial Programming (ICSP), also known as In-System Programming (ISP), is a method for programming microcontrollers, programmable logic devices, and other embedded systems while they are installed in a bigger system [41,42]. This technique allows for firmware updates and programming without the need to remove the chip from the circuit. Devices can be programmed directly on the circuit board, eliminating the need for pre-programmed chips. It allows for easy updates to the device's firmware, enabling bug fixes and new features without hardware changes. It also reduces the need for additional programming circuitry on board, simplifying the overall design.

The pins used in ICSP are Vdd: Power supply voltage, Vss: Ground, MCLR/Vpp: Master Clear/Programming voltage, PGD: Program Data, and PGC: Program Clock. Many microcontroller manufacturers, including Microchip, Atmel, and others, have widely adopted ICSP. This makes it a standard method for programming a variety of devices. For many microcontrollers, ICSP programming can take anywhere from a few seconds to a minute. For example, programming a PIC16F628A microcontroller can take around 10 to 15 s. ICSP is essential for this reporter-verifier model, not for changing the code but for reading the binary flash from the microcontroller.

3. Proposed Firmware Validation Architecture with Multi-Processor Method

This section presents the overall system architecture and its components in more detail—the reporter-verifier model, embedded software design, and blockchain's role.

3.1. Firmware Validation Architecture

Like in a typical IoT application, there is a *primary* stakeholder who is responsible for developing the IoT solution—the devices, software, and operational procedures. There could be more than one IoT stakeholder in an IoT application, but for any particular device, there is only one business entity responsible for the data from the device. The device, once deployed, periodically sends the data collected from the environment to the blockchain. The following are a few extra steps in this architecture:

(1) When the device is created, its *source code* is also formed to serve the application requirements. All stakeholders agree upon these requirements in the IoT-blockchain application.
(2) The *binary code* is generated from the *source code* using a compiler.
(3) The corresponding binary code *hash(es)* is generated in parallel with the device's deployment or upgrade.

All stakeholders can be aware of the code and, hence, its binary and hash, and these can be shared offline or outside of the blockchain during the IoT system setup. There is always a one-to-one relationship between the code's hash and the code running on the device. Therefore, if there is any attempt to falsify the data with a different firmware code before reporting it to the blockchain, it will be detected. When data are in the blockchain, they are given to clients or consumers.

Figure 2 shows this system architecture in detail, and it is described in the following few sections. The new innovative device would extract its own firmware as a binary code when operational and verify it. This is called a *reporter-verifier* design.

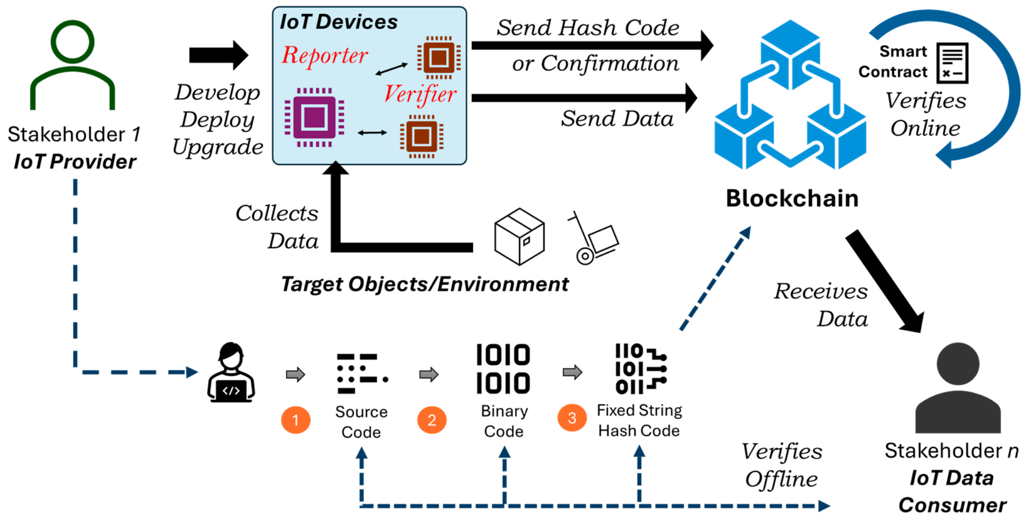

Figure 2. A new blockchain-based IoT architecture with code hash checking. There is one *reporter* processor, and it may have multiple *verifier* processors.

A device with this new design acts like a typical IoT device by sending the data along with the verification or hash of the sensor device's firmware. The device is supported by a blockchain network that provides integrity of the information—both the sensor information and the validity of the firmware executing on the device to collect that information from its environment. The device contains one processor—the reporter—that acts as the typical processor of an IoT device for controlling the sensor components, local decision-making, communication modules, and local storage. The device also contains one or more additional microprocessors that verify the reporter's firmware or binary code. The verifier has to extract the reporter's firms, convert them to hash, and access any storage to verify the hashes before communicating through the internet.

3.2. Reporter-Verifier Model

The reporter-verifier model is the core of the proposed IoT device design architecture. Typically, an IoT device contains only one microprocessor or microcontroller. This contains the code to execute to collect data from sensors monitoring the environment. We term this processor the *reporter* due to its role in continuously reporting data to external entities such as the blockchain and other clients. The IoT developers decide on the *reporter* microprocessor, and the corresponding firmware is also created by the stakeholders owning the IoT devices and network.

In this new device design model, there is another microprocessor called the *verifier* that can support the required protocols, e.g., ICSP. These microprocessors can be generic, and the other competing stakeholders can decide their type. Verifiers can be put as add-on stackable connectors [43] for easy plug-in or swapping of the verifier. Alternatively, the verifier may be permanently embedded into the device's PCB, along with the reporter. The designers need to ensure that there is an ICSP connector, power, and, optionally, access to the communication module. In this model, we use an extra pin line along with the ICSP pin lines called the *Program Lock*. This line has a digital output pin for the slave (LOCK_OUT) and a digital input pin for the master (LOCK_IN). If pulled high, this would indicate that the slave is not ready for a system reset, and it should only be reset when the pin is pulled too low. Depending on the nature of the *verifier* deployment, there can be two unique designs.

3.2.1. Active Verifier Design

In this design, the *verifier* processor's role is to periodically extract the binary code from the *reporter* and check its authenticity. In most cases, it can locally confirm this and send a confirmation message or send the hashes of the binary code to the blockchain. Although insecure, in rare cases, it can send the binary code outside to be confirmed on the blockchain. The *verifier* acts as a sentry to monitor the reporter processor continuously. Other stakeholders can develop the code in the verifier, or it can come from an open-source code explicitly designed for verification purposes.

The reporter program is meant to perform a typical function—collect the sensor information, encrypt it if necessary, and transmit it to the blockchain. For this, it is also connected to the radio. The *verifier* needs to connect with the radio/communication module to send the hash(es) or the confirmation message outside to the blockchain. The radio or communication module is primarily connected to the reporter but can be operated by the verifier if needed. In order to verify the hash, the verifier and blockchain can follow a few different options:

- *Local Hash and Match Mode* (LHM): In this, the verifier aims to perform all the tasks on the device—*binary code extraction*, *hashing*, and *matching*—before sending a confirmation. The fundamental drawback with this approach is that the verifier cannot be too strong, i.e., have large RAM or processing power. Most low-power microprocessors would have a minimum speed of 1 MHz, which is sufficient for performing hashes. Assuming the processing power is constant, the major limitation is memory. Different hashing techniques generate a hash of different fixed-size strings ranging from 128 to 1024 bits. However, the compiled code can be in several KBs. For example, the Arduino ATMEGA2560 processor has a flash memory size of 256 KB and a RAM of 8 KB. This means the code written can be much larger than the RAM available in a typical microprocessor. The size of the reporter flash memory is denoted as α and the RAM of the verifier as β. In order to verify the program, the verifier needs to divide the reporter code into blocks with a block size of at most 8 KB, and the number of blocks n can be:
$$n = \alpha/\beta \tag{1}$$
  The verifier reads the code from the reporter byte by byte so it can partition the code into blocks and then run the hashing on each block. This is shown in Figure 3.
  The *verifier* stores the expected n hash(es) in an EEPROM. When a new code is loaded into the reporter in a secure medium with the knowledge of all stakeholders, the verifier creates the hashes and stores them in a dedicated EEPROM. EEPROM can be low-cost and available in larger sizes than the reporter flash size α. The verifier can detect any further changes in the reporter code by matching the hashes with the hashes in EEPROM.
  The *verifier* only sends *confirmation* of matching to the blockchain, which requires much less memory. This is the most secure approach for protecting the device code, as nothing leaves the device. It can also keep verifying even when there is no network connection for a period of time and send the confirmations when the network becomes available.
- *Local Hash Remote Match Mode* (LHRM): In a more untrusted but likely environment, the *verifier* does not match the hashes locally. It creates the n hashes and sends them to the blockchain, where smart contracts match the hash(es) on the blockchain. It may not need an EEPROM for this, but the size of the blocks is determined by the size of the buffer available on the verifier for transmissions.
  This is a zero-trust model and the most likely implementation. However, the verifier may need to send many batches of hashed data, which requires a lot more transmissions.

- *Remote Hash and Match Mode* (RHM): If no hashing is implemented on the verifier, then the verifier can send segments of the code back to the blockchain, where it can be hashed and matched. This is very insecure, as the code, even though it is in a binary form, has to be regularly transmitted over a network. Even if this is encrypted, there is a possibility of a leak.

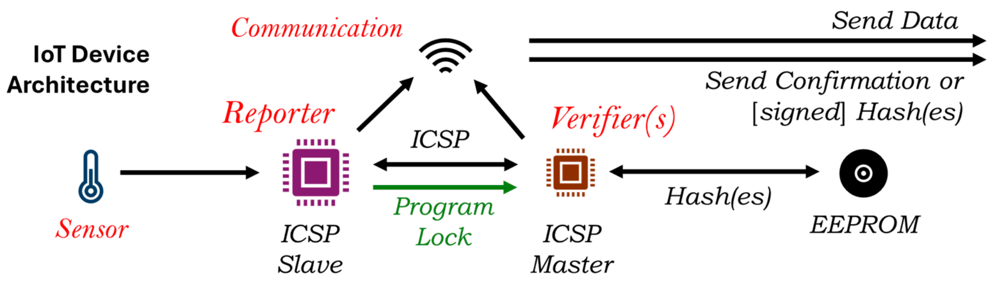

Figure 3. Detailed reporter-verifier model: Active Verifier Design.

In LHM and LHRM, the block size can be altered according to the RAM available to the verifier. If the hash verifications take place on the blockchain, it can use a random block size every time to enhance security. The other drawbacks of this design include:

- The *verifier* resets the *reporter*. To address this, the lock pin may be used, where the reporter can set the lock pin to logically *high* to prevent the verifier from initiating a read. As most edge devices sleep after operating for a while, the reset of the reporter should not matter in its operations.
- The verifier needs to sleep as well and synchronize with the reporter to stay awake and perform the check at the right time. This can be achieved easily with the proper hardware design. Also, if the verifier does not send the hash or confirmation in time, the blockchain can start an alert notification.

Optionally, to strengthen identity and authentication and prevent verifier forgery, the verifier can have a private key as part of its code. The private key can be used to encrypt/sign the hashes. Once the encrypted/signed hash is received outside the device, it can be decrypted/verified by its paired public key. The private and public keys are generated corresponding to individual verifiers.

3.2.2. Verifier as a Module Design

An alternative design is to have the verifier act solely as an ICSP master and as an external electronic module. The reporter initiates the verifier. The verifier reads the binary code and uses a preloaded algorithm to create a digital signature based on the binary code and its own verifier identification details. This design eliminates the need to share any onboard resource with the verifiers. However, this needs a stronger, unique identification mechanism for each verifier to ensure that the digital signature is valid and cannot be forged by the reporter. This is shown in Figure 4. In this design, the lock pins are not needed, as the reporter can ask the verifier to read when ready. However, as the reporter will be reset after this request, the verifier needs to wait until the reporter has restarted and is prepared to receive the signed hashes. Due to the memory limitation, as discussed earlier, it would be more challenging to pass the message back to the reporter. This would invariably require extra memory, such as a larger EEPROM.

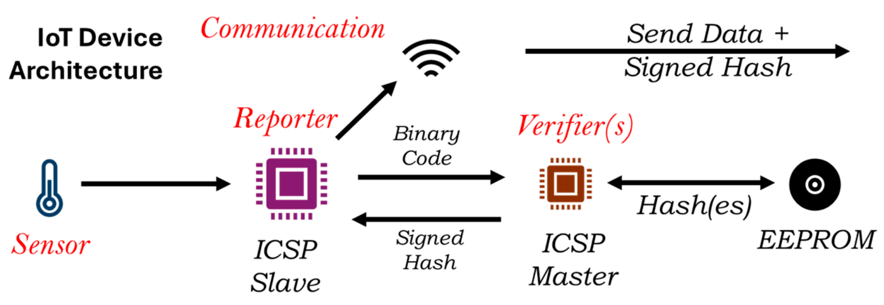

Figure 4. Detailed reporter-verifier model: Verifier as a Module Design.

3.2.3. Authenticity of the Verifier Code

The verifiers themselves are also expected to behave honestly. They should not misreport the authenticity of the reporter firmware. However, the verifier code can be sealed, i.e., Write Once Read Many (WORM), as it has only one task—to verify—which does not require any update. Thus, instead of a variable code, the verifier can be created such that the code cannot be altered once written, or it would require a specialized setup to change the code. This way, the reporter can be assured that the verifier will not falsify the hash code either.

Also, it is difficult to change the verifier code often. This means that once the verifier starts to send anything incorrectly, intentionally, or due to a hardware fault, it will keep reporting that continuously. However, it is possible to verify the reporter firmware manually if there is any suspicious verifier activity.

To sum up, without a verifier, the reporter can falsify their data intermittently without ever being detected, even with manual checking. This is due to the variable output expected from the reporter. However, the verifier is expected to produce the same results every time, so the verifier's misreporting can be manually verified.

3.3. Unique Identification of an Embedded Program in Flash Memory

In order to enable the verifier processor to extract and verify the code in the reporter, it is necessary to fulfill four essential criteria: generating *binary code*, *code reconstruction*, uniqueness of *binary code*, and *hashing*.

3.3.1. Binary Code—Hexadecimal Representation

When a code for the IoT device is compiled with a toolchain such as the STM or Arduino IDE, it goes through several stages to produce the final binary file, which is uploaded to the microcontroller. The source code is created in raw High-Level Language (HLL) such as C/C++. After that, the IDE typically performs the following steps:

- *Preprocessing*: The compiler adds necessary function prototypes and includes core libraries.
- *Compilation*: The preprocessed code is compiled into object files, such as the avr-gcc compiler for AVR processors. This compiler is different for each type of board.
- *Linking* and *Conversion*: The object files are linked together to create a single binary file, such as an Arduino board .elf file. The binary file is then converted into the final machine code, which is in the hexadecimal format. In the case of Arduino boards, it is a .hex file.

The hexadecimal file or binary code contains hexadecimal representations of binary data. It consists of multiple lines, each representing a record with specific fields like start code, byte count, address, record type, data, and checksum. This file is used to upload the compiled code to the microprocessor's flash memory. For example, tools like XLoader or the Arduino CLI can be used to upload the .hex file to an Arduino board.

3.3.2. Reconstructing Source Code

It is essential to ensure that the code is not recoverable from the hexadecimal machine code to preserve privacy and prevent future tampering. The code is uploaded over the air or manually. Either way, it is best for security purposes to ensure that the code is not visible to any party who may have access to the network or the device and is able to extract the binary file. The stakeholders involved in the entire IoT application may have the source code, but no one else should have the source code to ensure the performance and features of the device are kept secret.

Fortunately, recovering the original source code from a hex file is quite challenging. The hex file contains compiled binary data, which are not directly convertible back to the original source code. It is possible to manually interpret the corresponding assembly code to understand the program logic. However, this step requires a good understanding of both the assembly language and the original program structure. Even then, it is extremely unlikely for anyone to recover the entire code.

3.3.3. Duplicate Binary Files

As the code's originality is aimed at being verified, two source codes should not generate identical binary files. Two different programs can generate identical .hex files under certain extremely rare conditions, typically due to compiler optimization:

- *Dead Code Elimination*: If both programs contain code that does not affect the program's output or behavior, the compiler might remove this "dead code", resulting in identical .hex files. Optimized code for an IoT device will not contain dead code. An eventual upgrade to the code would always add new features and corresponding function codes.
- *Identical Functionality*: If two programs perform the same operations in the same way, even if their source code looks different, the compiler might produce the same machine code for both. This is also nearly impossible for commercial IoT devices, as the upgrade would use the same libraries and only add to the original code.
- *Minimal Programs*: Very simple programs, such as those that only initialize variables or contain empty loops, might compile to the same .hex file because there is not much code to differentiate them. Commercial IoT devices should definitely have complex code.
- *Compiler Settings*: Using the same compiler settings and libraries can also lead to identical hex files for different programs. The developer can set this to ensure that no two codes generate the same binary file.

3.3.4. Hashing

The binary hex code of the source code is of variable size. This is not ideal for use in a blockchain application where various stakeholders own the distributed application interfaces. A non-uniform size could prevent matching in all applications. To resolve this, one or more hashes can be generated for the binary code. Hashing is a speedy process on a regular computer. However, it may be challenging to perform the hash properly on a low-power microprocessor. Hence, for a given application, it is decided on the basis of the processors' capabilities.

If implemented, hashing generates a fixed-size string (hash value) from input data, which can be used to verify the integrity of the file. Standard hashing algorithms include MD5, SHA-1, and SHA-256. Creating a hash is the best way to confirm that the .hex file has not been altered.

3.4. Role of Blockchain and Version Control

Blockchain plays a passive role in this reporter-verifier model. It provides reliability in matching the hashes or validating the confirmations. Without the blockchain, the proposed reporter-verifier model will become essentially meaningless. If an alternative database is used, then that leaves the same vulnerability to manipulation, which is the very reason blockchain has been proposed [7].

The process of matching the hash value can take place at the edge or on the blockchain. In any of the implementations of LHM, LHRM, or RHM, the blockchain has a smart contract.

- *LHM smart contract*: For this, the smart contract only needs to record a successful confirmation message from the verifier along with its identification details in definite time periods. These confirmations are then written in the blockchain.
- *LHRM smart contracts*: For this, the smart contract receives the reporter's hash and identification details. It then matches the hash with the expected hash of the device based on the identification details. This requires some additional calculations but nothing significant for a cloud-based blockchain. If there is a match, then confirmation is stored in the blockchain.
- *RHM smart contract*: The smart contract receives the device's hexadecimal/binary code along with the identification. It then has to look up the corresponding hashing and, optionally, any decryption mechanism corresponding to the device and perform the operations to obtain the binary code's hash. Then, it matches the hash with the expected hash and stores the results in the blockchain.

In all of the above cases, the blockchain would expect the inputs at specific time periods, and an alert notification would be generated if the hash does not match. However, it should be noted that blockchain does not add much to the networking requirements of devices. The connection between a device and the blockchain is just an HTTP request (or a similar TCP/IP protocol) to send the target information. The verifier confirmation messages can be piggybacked or co-scheduled on the normal data interaction. Also, there is no need for the verifier to verify continuously, i.e., the rate of verification needs to be equal to or less than the rate of the sensor data message being sent to the blockchain, e.g., m message/hr. There may be an increase in transmission in LHRM where the rate of message can become a maximum of $m \cdot n$ messages/hr if the verifier verifies every single block. However, depending on the application, the actual verification can be performed at much slower rates as long as it meets the security requirements.

4. Results

This section analyzes the time and excess power consumption of the verifier. The experimental setup is shown in Figure 5. Both the *reporter* and *verifier* are microcontroller units, but only the reporter is changed for different analyses.

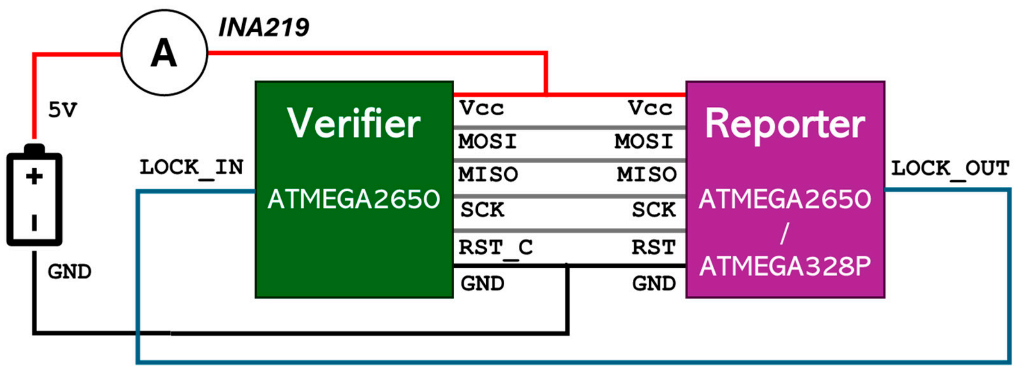

Figure 5. The experimental setup.

As the blockchain is a passive database, we do not test the blockchain in this paper. The blockchain is implemented over the network, and regardless of the scalability of the cloud, the device's performance will not be impacted by this. We used an INA219 current and power sensor for this setup. The LOCK line had a resistance of 100 KΩ to reduce the current leakage to a minimum. Other lines had no additional resistance except for what was built into the Arduino boards.

4.1. Time Consumption Analysis

For this setup, only the *reporter* is connected to a PC to see the serial output regarding the progress of the hashing. Neither is connected to any other components, and the current sensing module is not connected in this setup.

It is necessary to ensure that the hashing algorithm does not take too much time. If the verifier takes too much time, then the reporter needs to remain awake and idle during that time period. For most reporters, the code size will not be too big to require much time. Still, it is possible that the verifier may inadvertently interrupt the *reporter* during the critical data collection periods.

We tested the block partitioning and hashing scheme of the verifier with 10 programs that were larger than 10 KB in the flash memory. These were chosen randomly from the examples provided by Arduino IDE [44]. We changed the block size from 16 to 2048. When the block size is increased, the time taken to process the entire code is reduced. Obviously, the time taken for each block increases with block size. We used the SHA1 and SHA256 hashing algorithms on the ATMEGA2560 and ATMEGA328P. The results are shown in Figure 6.

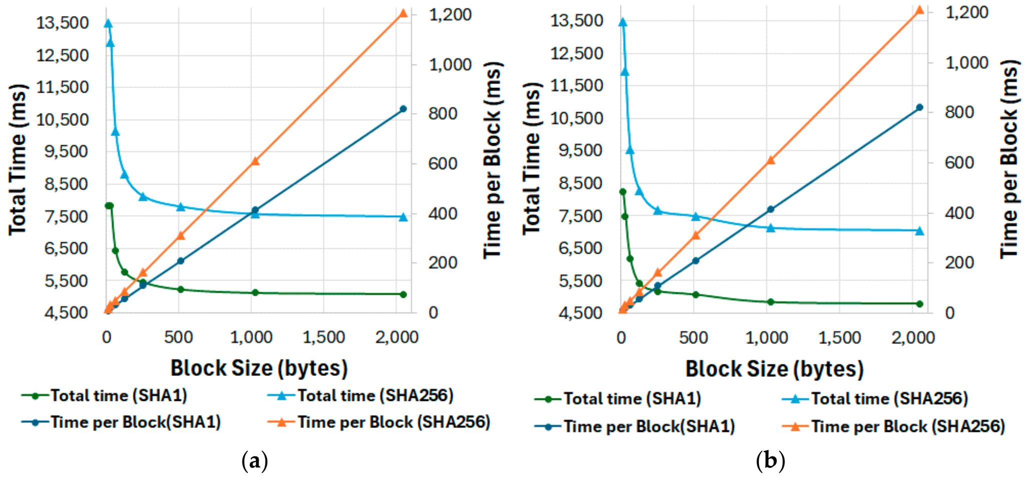

Figure 6. Time consumption characteristics of the reporter-verifier model. (**a**) Reporter: ATMEGA2560 (Arduino MEGA). (**b**) Reporter: ATMEGA328P (Arduino Duemilanove).

4.2. Power Consumption Analysis

For this setup, neither the *reporter* nor the *verifier* is connected to the PC or any other component. The current sensing module is connected, as shown in Figure 5, and directly connected to a PC to read the current consumption.

The current consumption increased by only about 5 mA for ATMEGA2650 and about 2 mA for the ATMEGA328P when the verification process was running compared to when the reporter code was executing normally (see Figure 7). Given that the verifications require about 7.5 to 13.5 s as per the time consumption test, the power consumption for each check is approximately 5.2×10^{-5} to 9.0×10^{-5} Wh, with an additional 5 mA power consumption. No additional peripheral modules were connected to the microprocessor for this, although the Arduino boards had typical electronic components expected of a microcontroller unit for IoT edge devices, such as LEDs and FTDI chips. The impact of the verifier code should be even less if the microprocessor and the peripheral modules consume more power themselves, and the additional power consumed by the verifier remains constant.

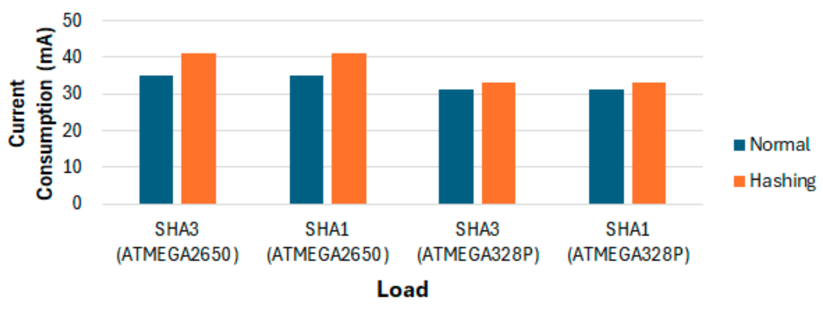

Figure 7. Power consumption characteristics of the reporter-verifier model.

The excess power consumption can be further optimized for a different set of reporters/verifiers without any additional electronics. However, the device would obviously have additional modules, such as storage and sensors, with a much higher current consumption than the verifier.

4.3. Encryption Time and Power Analysis

Encryption may be required if digital signatures are implemented to manage the identity of verifiers. If this technique is applied, then additional time will be needed to sign. On the ATMEGA2560 (verifier) tested here, using ED25519 cryptography, signing with a private key of size 32 bytes of a message of 32 bytes takes approximately 6.2 s, and the time remains almost the same with increasing message size. Figure 8 shows the time consumed by ED25519 and AES encryptions for various payload or block sizes. Figure 9 shows the increase in current consumption, which is again very minimal—about 2 mA for the ATMEGA328P (reporter). This indicates that encryption can also be implemented on the verifiers without causing excessive delays or increases in power consumption.

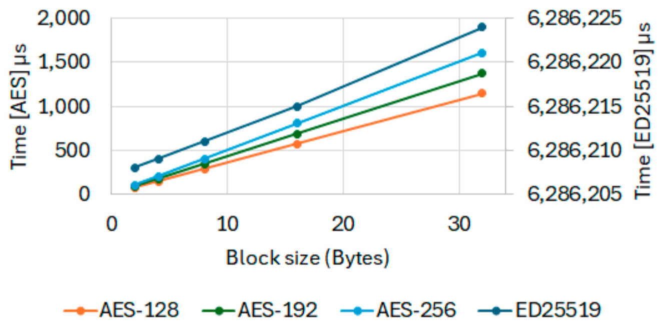

Figure 8. Time consumption characteristics of the reporter-verifier model for encryption for digital signature.

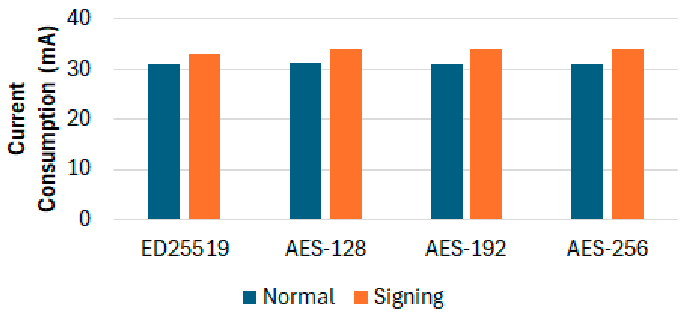

Figure 9. Power consumption characteristics of the reporter-verifier model for encryption for digital signature (ATMEGA328P).

5. Discussion

The proposed model has significance in improving the reliability of devices with complex functionalities in the age of edge AI, but it also faces some challenges.

5.1. IoT Reliability in the Era of Edge AI

Edge AI enables the deployment of artificial intelligence algorithms directly on IoT devices. This approach allows for faster data processing and reduced latency [4]. Edge AI is beneficial where real-time decision-making is crucial. This also means that the firmware becomes bigger and more complex. This increases the potential and incentive to produce biased results from edge devices and, hence, necessitates more scrutiny at the edge with approaches like the reporter-verifier model.

The results show that the reporter-verifier model is feasible in terms of memory space and speed of memory reading. The main advantage of the proposed architecture is that it is perfect in preventing stakeholder-generated IoT code that may be designed to send falsified data.

The verifier hardware can be chosen on the basis of Equation (1) and the non-linear nature of hash execution time. The execution time advantage of increasing block sizes diminishes rapidly. For example, in the above setting, with an LHM, the best block size would be 500 B. The devices can be designed at a low cost. The cost of adding a processor to the design should not exceed more than USD 1–2. The verifier does not have to be powerful, either. Adding an EEPROM is also inexpensive.

The additional electronic components would definitely increase space requirements. However, using a stackable connector can allow the replacement of the verifier in future designs. The space requirements should not be prohibitive in design. The concept of reading the exact code in the reporter can be expanded to a networked reporter-verifier where the reporter and the verifier are not on the same circuit board. On the other hand, not all IoT devices would require this level of security.

We did not test the blockchain's performance with this new architecture, as it is designed to fit into existing IoT blockchain networks and cannot overload the network with data. Also, several techniques have been developed in blockchain to increase the transaction processing rates with sharding [45]. These can handle incoming requests at a rate of several hundreds of transactions per second [46]. The proposed device design architecture can seamlessly replace any existing IoT device that can communicate and requires an extra layer of security. However, future work could investigate any variations of the efficiency for the different schemes of LHM, LHRM, and RHM.

5.2. Limitations and Challenges

There are also some inherent disadvantages to this architecture. The first issue is the overhead in the supply chain and manufacturing. The reporter-verifier model can be perfect once the device is manufactured, the verifier is sealed in the device, and it can no longer be physically compromised. However, it is necessary to ensure that the correct verifier components and the code running on the verifier are what the IoT application stakeholders expect. This can be addressed by writing the verifier code with embedded private keys and unique identifications belonging to non-owners of the devices, which can be used to sign the hash and can be decoded with corresponding public keys. Once written, the verifier code can be compiled, and only the binary verifier code is shared in the manufacturing process. This way, the device owners can be sure that the verifier is acting in good faith but cannot replace the code on the verifier or the verifier components themselves. However, this process still requires additional steps during manufacturing and deployment, as well as an infrastructure to maintain the keys.

The second issue is the scalability of the verifiers. Despite its low power consumption, adding multiple verifiers consumes more space and complicates the electronics. It is also not possible to have unlimited verifiers accessing the memory of the reporters, as they would have to coordinate their operation as well. There is also a technical limitation with ISCP or similar mechanisms in terms of how many processors can be connected together.

The third issue is the exposure of the binary code. Even in binary form, it is still intellectual property that may be compromised. While reconstructing the code completely is impossible, experts can identify specific characteristics if they can see the binary code. The reporter-verifier model, particularly the LHM and LHRM, does not innately expose the code. However, it is still possible for hackers to extract this information if they can physically obtain a device.

6. Conclusions

IoT and blockchain are two innovative technologies that can enable a wide range of applications. These applications have many users with various interests. These applications require trust at all levels, including on the IoT devices themselves. This paper proposed and evaluated a multi-processor architecture based on a zero-trust principle. IoT edge devices created with this approach can be developed cost-effectively and deployed without high overhead requirements. The devices can work with a range of back-end blockchain services. The supporting blockchain can be permissioned or semi-permissioned based on the IoT applications.

Author Contributions: Conceptualization, A.M.; methodology, A.M.; software, A.M.; validation, A.M. and A.A.K.; formal analysis, A.M. and A.A.K.; investigation, A.M.; resources, A.M.; data curation, A.M.; writing—original draft preparation, A.M.; writing—review and editing, A.M. and A.A.K.; visualization, A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Chen, Z. Security of Esoteric Firmware and Trusted Execution Environments. Ph.D. Thesis, University of Birmingham, Birmingham, UK, 2023.
2. Wu, Y.; Wang, J.; Wang, Y.; Zhai, S.; Li, Z.; He, Y.; Sun, K.; Li, Q.; Zhang, N. Your firmware has arrived: A study of firmware update vulnerabilities. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 5627–5644.
3. Akkaoui, R.; Stefanov, A.; Palensky, P.; Epema, D.H.J. Resilient, Auditable, and Secure IoT-Enabled Smart Inverter Firmware Amendments with Blockchain. *IEEE Internet Things J.* **2024**, *11*, 8945–8960. [CrossRef]
4. Singh, R.; Gill, S.S. Edge AI: A survey. *Internet Things Cyber-Phys. Syst.* **2023**, *3*, 71–92.
5. Abbassi, Y.; Benlahmer, H. IoT and Blockchain combined: For decentralized security. *Procedia Comput. Sci.* **2021**, *191*, 337–342. [CrossRef]
6. Obaidat, M.A.; Rawashdeh, M.; Alja'afreh, M.; Abouali, M.; Thakur, K.; Karime, A. Exploring IoT and Blockchain: A Comprehensive Survey on Security, Integration Strategies, Applications and Future Research Directions. *Big Data Cogn. Comput.* **2024**, *8*, 174. [CrossRef]
7. Rejeb, A.; Rejeb, K.; Appolloni, A.; Jagtap, S.; Iranmanesh, M.; Alghamdi, S.; Alhasawi, Y.; Kayikci, Y. Unleashing the power of internet of things and blockchain: A comprehensive analysis and future directions. *Internet Things Cyber-Phys. Syst.* **2024**, *4*, 1–18. [CrossRef]
8. Allouche, M.; Frikha, T.; Mitrea, M.; Memmi, G.; Chaabane, F. Lightweight Blockchain Processing. Case Study: Scanned Document Tracking on Tezos Blockchain. *Appl. Sci.* **2021**, *11*, 7169. [CrossRef]

9. Anagnostakis, A.G.; Giannakeas, N.; Tsipouras, M.G.; Glavas, E.; Tzallas, A.T. IoT Micro-Blockchain Fundamentals. *Sensors* **2021**, *21*, 2784. [CrossRef]
10. Anderson, J.S.; Ravindran, B.; Jensen, E.D. Consensus-Driven Distributable Thread Scheduling in Networked Embedded Systems. In Proceedings of the International Conference on Embedded and Ubiquitous Computing, Taipei, Taiwan, 17–20 December 2007; Kuo, T.-W., Sha, E., Guo, M., Yang, L.T., Shao, Z., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 247–260.
11. Dorri, A.; Jurdak, R. Tree-Chain: A Lightweight Consensus Algorithm for IoT-based Blockchains. In Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Sydney, Australia, 3–6 May 2021; pp. 1–9. [CrossRef]
12. Frikha, T.; Chaabane, F.; Aouinti, N.; Cheikhrouhou, O.; Ben Amor, N.; Kerrouche, A. Implementation of Blockchain Consensus Algorithm on Embedded Architecture. *Secur. Commun. Netw.* **2021**, *2021*, 9918697. [CrossRef]
13. Kolaric, P.; Chen, C.; Dalal, A.; Lewis, F.L. Consensus controller for multi-UAV navigation. *Control Theory Technol.* **2018**, *16*, 110–121. [CrossRef]
14. Ktari, J.; Frikha, T.; Chaabane, F.; Hamdi, M.; Hamam, H. Agricultural Lightweight Embedded Blockchain System: A Case Study in Olive Oil. *Electronics* **2022**, *11*, 3394. [CrossRef]
15. Ktari, J.; Frikha, T.; Hamdi, M.; Hamam, H. Enhancing Blockchain Consensus with FPGA: Accelerating Implementation for Efficiency. *IEEE Access* **2024**, *12*, 44773–44785. [CrossRef]
16. Ktari, J.; Frikha, T.; Yousfi, M.A.; Belghith, M.K.; Sanei, N. Embedded Keccak implementation on FPGA. In Proceedings of the 2022 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS), Cairo, Egypt, 6–9 June 2022; pp. 1–5.
17. Mahmoud, M.A.; Gurunathan, M.; Ramli, R.; Babatunde, K.A.; Faisal, F.H. Review and Development of a Scalable Lightweight Blockchain Integrated Model (LightBlock) for IoT Applications. *Electronics* **2023**, *12*, 1025. [CrossRef]
18. Andola, N.; Venkatesan, S.; Verma, S. PoEWAL: A lightweight consensus mechanism for blockchain in IoT. *Pervasive Mob. Comput.* **2020**, *69*, 101291. [CrossRef]
19. Rao, N.M.; Veer, M.S.; Jayachandra, K.; Santosh, C.V.; Goud, B.M.; Naidu, P.R. Blockchain Over Low Power Microcontrollers. In Proceedings of the 2022 Third International Conference on Intelligent Computing Instrumentation and Control Technologies (ICICICT), Kannur, India, 11–12 August 2022; pp. 735–739. [CrossRef]
20. Tanjung, W.; Fat, J.; Hugeng, H. Blockchain Technology Implementation on Simple Microcontroller Based Ballots System. *Int. J. Appl. Sci. Technol. Eng.* **2023**, *1*, 770–777. [CrossRef]
21. Xu, W. Hybrid Fault Tolerant Consensus in Wireless Embedded Systems. Ph.D. Thesis, Technische Universität Braunschweig, Braunschweig, Germany, 2021.
22. Zhang, W.; Wu, Z.; Han, G.; Feng, Y.; Shu, L. LDC: A lightweight dada consensus algorithm based on the blockchain for the industrial Internet of Things for smart city applications. *Future Gener. Comput. Syst.* **2020**, *108*, 574–582. [CrossRef]
23. He, Y.; Huang, D.; Chen, L.; Ni, Y.; Ma, X. A survey on zero trust architecture: Challenges and future trends. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, 6476274.
24. Xiao, Y.; Jia, Y.; Liu, C.; Cheng, X.; Yu, J.; Lv, W. Edge Computing Security: State of the Art and Challenges. *Proc. IEEE* **2019**, *107*, 1608–1631. [CrossRef]
25. Mahadevappa, P.; Al-amri, R.; Alkawsi, G.; Alkahtani, A.A.; Alghenaim, M.F.; Alsamman, M. Analyzing Threats and Attacks in Edge Data Analytics within IoT Environments. *IoT* **2024**, *5*, 123–154. [CrossRef]
26. Zhao, Y.; Cheng, G.; Duan, Y.; Gu, Z.; Zhou, Y.; Tang, L. Secure IoT edge: Threat situation awareness based on network traffic. *Comput. Netw.* **2021**, *201*, 108525. [CrossRef]
27. Abawajy, J.; Huda, S.; Sharmeen, S.; Hassan, M.M.; Almogren, A. Identifying cyber threats to mobile-IoT applications in edge computing paradigm. *Future Gener. Comput. Syst.* **2018**, *89*, 525–538. [CrossRef]
28. Ficco, M.; Granata, D.; Rak, M.; Salzillo, G. Threat Modeling of Edge-Based IoT Applications. In Proceedings of the International Conference on the Quality of Information and Communications Technology, Algarve, Portugal, 8–11 September 2021; Springer: Cham, Switzerland, 2021; pp. 282–296.
29. Aldaej, A.; Ullah, I.; Ahanger, T.A.; Atiquzzaman, M. Ensemble technique of intrusion detection for IoT-edge platform. *Sci. Rep.* **2024**, *14*, 11703. [CrossRef]
30. Guin, U.; Cui, P.; Skjellum, A. Ensuring Proof-of-Authenticity of IoT Edge Devices Using Blockchain Technology. In Proceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 30 July–3 August 2018; pp. 1042–1049. [CrossRef]
31. Guin, U.; Singh, A.; Alam, M.; Canedo, J.; Skjellum, A. A secure low-cost edge device authentication scheme for the internet of things. In Proceedings of the 2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), Pune, India, 6–10 January 2018; IEEE: New York, NY, USA, 2018; pp. 85–90.
32. Lee, B.; Lee, J.-H. Blockchain-based secure firmware update for embedded devices in an Internet of Things environment. *J. Supercomput.* **2017**, *73*, 1152–1167. [CrossRef]

33. Nadir, I.; Mahmood, H.; Asadullah, G. A taxonomy of IoT firmware security and principal firmware analysis techniques. *Int. J. Crit. Infrastruct. Prot.* **2022**, *38*, 21. [CrossRef]
34. Verderame, L.; Ruggia, A.; Merlo, A. PARIOT: Anti-repackaging for IoT firmware integrity. *J. Netw. Comput. Appl.* **2023**, *217*, 18. [CrossRef]
35. Yeasmin, S.; Haque, A.; Sayegh, A. A novel and failsafe blockchain framework for secure OTA updates in connected autonomous vehicles. *Veh. Commun.* **2023**, *43*, 12. [CrossRef]
36. Bruschi, F.; Zanghieri, M.; Terziani, M.; Sciuto, D. Decentralized Updates of IoT and Edge Devices. In Proceedings of the International Conference on Advanced Information Networking and Applications, Kitakyushu, Japan, 17–19 April 2024; Springer: Cham, Switzerland, 2024; pp. 161–170.
37. Bakhshi, T.; Ghita, B.; Kuzminykh, I. A Review of IoT Firmware Vulnerabilities and Auditing Techniques. *Sensors* **2024**, *24*, 708. [CrossRef]
38. Rashmi, R.; Karthikeyan, A. Secure boot of embedded applications-a review. In Proceedings of the 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 29–31 March 2018; pp. 291–298.
39. Alanzi, T. A review of mobile applications available in the app and google play stores used during the COVID-19 outbreak. *J. Multidiscip. Healthc.* **2021**, *14*, 45–57. [CrossRef]
40. Meza, J.; Wu, Q.; Kumar, S.; Mutlu, O. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Perform. Eval. Rev.* **2015**, *43*, 177–190. [CrossRef]
41. Ahmed, A.I.; Sharobim, B.K.; Fetteha, M.A.; Said, L.A.; Eltawil, A.M.; Madian, A.H. A Design of Rechargeable In-Circuit Serial Programmer for Industrial Embedded Systems. In Proceedings of the 2024 International Conference on Microelectronics (ICM), Doha, Qatar, 14–17 December 2024; pp. 1–5.
42. Ahmed, A.; Sharf, S.; Badawy, W.; Madian, A.; Abdelhamied, R.; Mekky, M.; Abdelhamied, M. Wireless ATMEL AVR In-Circuit Serial Programmer based on Wi-Fi and ZigBee. In Proceedings of the 2020 16th International Computer Engineering Conference (ICENCO), Cairo, Egypt, 29–30 December 2020; pp. 187–190.
43. Ajanovic, J. PCI express 3.0 overview. In Proceedings of the 2009 IEEE Hot Chips 21 Symposium (HCS), Stanford, CA, USA, 23–25 August 2009; pp. 1–61.
44. Amestica, O.E.; Melin, P.E.; Duran-Faundez, C.R.; Lagos, G.R. An Experimental Comparison of Arduino IDE Compatible Platforms for Digital Control and Data Acquisition Applications. In Proceedings of the 2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), Valparaiso, Chile, 13–27 November 2019; pp. 1–6. [CrossRef]
45. Bulgakov, A.L.; Aleshina, A.V.; Smirnov, S.D.; Demidov, A.D.; Milyutin, M.A.; Xin, Y. Scalability and Security in Blockchain Networks: Evaluation of Sharding Algorithms and Prospects for Decentralized Data Storage. *Mathematics* **2024**, *12*, 3860. [CrossRef]
46. Haque, E.U.; Abbasi, W.; Almogren, A.; Choi, J.; Altameem, A.; Rehman, A.U.; Hamam, H. Performance enhancement in blockchain based IoT data sharing using lightweight consensus algorithm. *Sci. Rep.* **2024**, *14*, 26561. [CrossRef]