Blockchain-based Runtime Attestation against Physical Fault Injection Attacks on Edge Devices

Charles Cao
University of Tennessee
Knoxville, TN, United States
Email:cao@utk.edu

Jayne Wu
University of Tennessee
Knoxville, TN, United States
Email:jwu10@utk.edu

Hairong Qi
University of Tennessee
Knoxville, TN, United States
Email:hqi@utk.edu

Shigetoshi Eda
University of Tennessee
Knoxville, TN, United States
Email:seda@utk.edu

ABSTRACT

With the ever-increasing proliferation of edge devices for applications, such as home automation and vehicle systems, their security vulnerabilities have received additional attention. A recent type of attack, physical fault injections, are particularly powerful as they can compromise these devices by skipping necessary instructions through physical methods, such as induced voltage glitches. Hence, they can trigger a wide range of software behavior anomalies and vulnerabilities not caused by the programs themselves. These attacks allow adversaries to carry out severe security breaches such as control flow hijacking and information leakage, even if the original device firmware has been well tested.

To defend against physical fault injections, this paper develops an innovative approach based on a runtime attestation of code execution, i.e., given a sequence of instructions, they should be executed thoroughly and correctly in a verified manner. Hence, instruction anomalies can be detected and reported if faults are injected. This protection mechanism does not require additional hardware or specialized instructions. Instead, it leverages a lightweight virtual machine to protect critical code segments such as password-related operations. It integrates two techniques: blockchain-based instruction integrity assurance and memory randomization-based data protection. We fully implemented this framework on an AVR-based microcontroller, and our evaluation results demonstrate that our methodology is practical enough to effectively prevent a wide range of hardware-based fault injection attacks, paving the way for more secure edge applications.

CCS CONCEPTS

• **Security and privacy** → *Hardware attacks and countermeasures*; **Embedded systems security**; **Trusted computing**; **Virtualization and security**.

SEC '23, December 6–9, 2023, Wilmington, DE, USA

ACM ISBN 979-8-4007-0123-8/23/12…$15.00
https://doi.org/10.1145/3583740.3628441

KEYWORDS

Physical Fault Injection, Edge Device Security, Embedded Systems

ACM Reference Format:
Charles Cao, Jayne Wu, Hairong Qi, and Shigetoshi Eda. 2023. Blockchain-based Runtime Attestation against Physical Fault Injection Attacks on Edge Devices. In *The Eighth ACM/IEEE Symposium on Edge Computing (SEC '23), December 6–9, 2023, Wilmington, DE, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3583740.3628441

1 INTRODUCTION

Edge computing has attracted considerable attention in recent years, as it has been deployed in multiple applications in everyday lives, from smart homes and vehicles to wearable and handheld devices [26, 29]. These devices, primarily embedded systems built on microcontrollers, wireless transceivers, and sensors, provide numerous benefits, including automation, remote control, and real-time monitoring. However, the widespread adoption and inherent characteristics of these devices make them an appealing target for potential attackers, who can exploit their vulnerabilities to launch a wide range of attacks, including control flow hijacking, information leakage, and privilege escalation, among others [16, 29].

We notice that in many deployments, the accessibility of these devices allows potential attackers to interact with them with additional hardware tools physically. This interaction has given rise to an active attack method known as physical fault injections, which have been demonstrated to be powerful enough to undermine encryption algorithms, facilitate unauthorized privilege escalation, and bypass security restrictions [2, 3, 8]. Specifically, researchers have developed and refined numerous fault injection techniques based on a range of methodologies, such as clock/voltage glitches [7, 19], electromagnetic pulses [15, 31], lasers [12, 25], X-rays [1], and Rowhammer attacks [22]. Recent measurements show that these methods are practical enough to be deployed within reasonable budgets [8]. Furthermore, these attacks are invasive enough to bypass security routines within a limited time using trial-and-error only [2, 6].

On the other side, we observe that existing edge computing platforms provide limited defense mechanisms against such types of attacks. Even the most recent and seemingly robust hardware security measures, such as ARM's TrustZone, have not been entirely immune to these attacks. This limitation is mainly because TrustZone safeguards the CPU's memory in safe mode from non-secure

mode programs [28]. However, with hardware-based fault injection, the very execution of code can be manipulated, e.g., instructions being skipped or modified, during execution, even in safe mode, leading to successful attacks.

Consequently, the developments of these attacks favor the attacker considerably. Some attacks, particularly those involving instruction skipping and byte and bit reversal, can be launched with resources no more potent than a standard desktop PC, and a budget between $100-$1,000 is often adequate, making these attacks far cheaper and hence more accessible than high-cost methods like lasers and EMP-based attacks [25].

```
bool authenticated = false;
counter--;
if (counter > 0)
{
  if (compare(input,
     password) == true)
     {
     counter = 3;
     authenticated = true;
     }
}
```

```
; Check if counter (r17) > 0
 cpi r17, 0x00
 breq end

; Compare input and password
; Load input into r18, r19
 lds r18, 0x0102
 lds r19, 0x0103
 cp r18, r19
 brne end
```

Figure 1: An example of instruction skipping attack where the attacker can skip the verification of the counter by introducing voltage jitters to skip the breq (branch-on-equal) and brne (branch-on-not-equal) instructions, thereby triggering the authenticated branch to be executed. In this figure, the left if the C code, and the right is the compiled code segment on the AVR instruction set. If we remove the breq and brne instructions, the code will not branch to end.

For example, consider a practical attack scenario targeting a 4-digit PIN verification algorithm (e.g., the ones used widely in ATMs) shown in Figure 1. A test counter ('counter') is employed to deter brute force attacks (attempts to test all possible PINs). If the counter reaches 0, a conditional "if" statement will block access to the **compare** function, resulting in a verification failure. But an attacker with fault injection can exploit this logic by skipping the verification of the counter or bypassing the **breq** and **brne** instructions through voltage jitters, thereby triggering the authenticated branch of code to be executed. Another possible way to bypass the counter is called byte-altering, which alters the value of the maximum allowed threshold to a large number, e.g., 100,000, which will eventually trigger the authenticated section of code to be executed by trying all password combinations.

Furthermore, the program can't assure security even if placed in the ARM's TrustZone [23, 28] because TrustZone is designed to protect against software attacks, not hardware-based attacks. Specifically, TrustZone primarily protects the memory accessed when the system is secure. It does not prevent the alteration of the execution flow of the code running in secure mode, which is what a fault injection attack does, leading to successful attacks.

In the face of these ever-evolving threats, we propose a software-centric, multi-pronged approach to protect against hardware attacks such as instruction skipping and byte altering. Our approach is developed around a lightweight virtual machine that executes the critical sections of the code, thereby minimizing the attack surface. Specifically, we propose a lightweight virtual machine layer capable of executing the AVR instruction set widely used in embedded systems. We then verify that the instructions are executed in the correct order and with the right values by comparing the execution logs of multiple runs of the critical sections and by recording their execution traces into a blockchain. Hence, the end state of the blockchain provides an immutable record for instruction integrity assurance. Even if only one instruction is tampered with, its blockchain records will differ from others. Furthermore, by introducing random timing delays when running instructions, it is practically impossible for the attacker to alter the same instruction in two rounds of execution. Finally, we propose a memory randomization technique to safeguard data integrity by making it difficult for an attacker to predict the location of critical data, thereby defending against byte-altering attacks. In summary, our key contributions are as follows:

(1) To our knowledge, we are the first to develop and introduce a new type of virtual machine-based verification specifically designed for secure code execution. This virtual machine plays a vital role in reducing potential attack surfaces, thus enhancing security by confining the execution of critical code sections within a highly controlled environment.
(2) We develop an innovative use of blockchain technology as a verification scheme called blockchain-based attestation to guarantee instruction integrity. The immutable nature of blockchains allows us to create a robust execution log, which can effectively detect any discrepancies arising from instruction tampering by comparing the end state of multiple execution logs.
(3) We develop a memory randomization technique specifically designed to safeguard data integrity. Randomizing the storage locations of critical data in memory imposes substantial difficulties for attackers attempting to predict the exact memory locations, thereby serving as a potent defense against data corruption and information leaks.

We've implemented these mechanisms on AVR-based microcontroller hardware. We then measured its efficacy using a benchmark of 6 programs regarding runtime slowdown and execution overhead. We test the resulting design against various attack scenarios, including instruction-skipping attacks and byte-altering attacks. The experimental results affirm the effectiveness of our approach in defending against adversary attacks.

The rest of this paper is organized as follows. Section 2 discusses the background and related work. Section 3 presents the design of our framework. Section 4 presents the implementation details. Section 5 presents the evaluation results. Section 6 concludes the paper.

2 RELATED WORK

In this section, we review the literature on physical fault injection attacks and defenses. We also discuss the use of blockchain technology in the context of security and trust.

2.1 Physical Fault Injection Methods

Numerous physical fault injection methods have been presented for hardware-centric attacks targeting embedded devices. They exploit the physical properties of devices to induce errors, disrupt normal behavior, or uncover vulnerabilities. Each method offers different benefits regarding efficiency, precision, and equipment requirements, and attackers might select a method based on their specific goals and resources [16, 26, 29].

Clock Glitch Fault Injection (CGFI): The first type of attacks, clock glitches [7, 13], are disturbances intentionally introduced into the system clock of an embedded device. This injection method causes discrepancies in the synchrony of logic block operations due to propagation delays. Such irregularities in hardware can introduce software-level vulnerabilities, such as causing one specific instruction to be skipped. This is because modern microcontrollers usually have multiple pipeline stages. Clock irregularity will cause some stages to be flushed, leading to instructions to be **no-ops**. Given its simplicity and low equipment requirements, CGFI remains a widely employed fault injection technique. However, precisely targeting specific instructions can be challenging, and the effects of CGFI are often transient, making it difficult to exploit the resulting vulnerabilities.

Electromagnetic Field Injection (EMFI): EMFI [21] leverages electromagnetic fields to interfere with the regular operation of electronic devices. It involves generating an intense local electromagnetic field that can disrupt microcontroller operations or interfere with the system voltage. These induced disturbances can result in abnormal system behavior, providing an avenue for attackers to compromise the system, similar to CGFI. Though more complex and demanding than CGFI, EMFI offers higher precision and control, making it an attractive option for sophisticated attacks.

Optical Laser Injection Attack: Optical attacks, particularly those using lasers, induce faults in the hardware components of embedded devices in a more controlled manner [11]. These attacks can cause transient or long-lasting faults in semiconductor materials by delivering concentrated light energy to specific device components. The faults can propagate throughout the system, potentially influencing software behavior and creating exploitable vulnerabilities. Despite the higher cost and complexity of laser-based attacks, they offer a significant advantage in precision, making them particularly effective for targeted attacks.

2.2 Impact of Fault Injection

Fault injection techniques alone can cause significant disturbances in the targeted devices, but they are not helpful if isolated at the hardware level. Therefore, understanding their impacts on software is crucial for attackers seeking to exploit vulnerabilities and defenders aiming to develop protection mechanisms. We discuss the most common effects of fault injection attacks below.

Transient Bit Flipping: This method refers to temporarily changing a bit's value to its opposite. This simple alteration forms the backbone of many fault attacks, as it offers a direct method to manipulate data or control flow within a system [9]. While individual bit flips might seem minor, they can lead to significant software vulnerabilities and pose severe security risks when strategically employed. For example, in Figure 1, if we could temporarily change the **authenticated** from false to true, we can skip the authentication process and gain access to the system. Therefore, understanding and defending against transient bit flipping is essential for maintaining software security in the face of physical attacks.

Stuck-at Bit Faults: Such faults occur when a bit's value is permanently changed [3]. Such faults can significantly affect system performance and reliability, distorting stored data and disrupting system behavior. Moreover, they can be used to skew True Random Number Generators (TRNGs), which play a critical role in security-related tasks, such as encryption and digital signatures [3, 27]. Therefore, the ability to induce and exploit stuck-at faults can considerably enhance an attacker's capabilities.

Control Flow Hijacking: In the control level, fault injections can lead to erroneous program execution, allowing attackers to manipulate the program's control flow. This technique could enable attackers to bypass authentication mechanisms, escalate privileges, or execute malicious code, among other security breaches [24]. Understanding and defending against control flow hijacking is thus essential for maintaining software security in the face of physical attacks.

Information Leakage: Induced faults can also lead to unintended information exposure, potentially revealing sensitive data [20]. By carefully analyzing the behavior of a faulty system, an attacker can infer and extract confidential information. Therefore, protection measures should also consider potential information leakage from fault injection attacks.

2.3 Blockchain-Based Security Measures

Blockchain has been developed for cryptocurrencies such as bitcoins, and also attracted attention in security and trust management [14] in recent years. Blockchain's immutable and distributed ledger capabilities make it an ideal candidate for implementing secure and trusted systems.

Specifically, several works have explored the use of blockchain technology for securing embedded devices and IoT systems [30]. For instance, by recording all changes to a piece of data in the blockchain, one can ensure that the data hasn't been tampered with [18]. This is particularly useful in scenarios where data integrity is of utmost importance, such as in healthcare or financial applications.

Besides data integrity, blockchain has also been used for managing access control in a distributed manner [10]. By using smart contracts, one can encode complex access control policies that can be enforced in a decentralized manner. This can be particularly useful in scenarios where traditional centralized access control mechanisms are not feasible or are prone to attacks.

However, while blockchain offers many benefits, it also has some limitations. For instance, the conventional distributed blockchains introduce excessive overhead, where the transaction throughput of blockchain is relatively low compared to traditional database systems, which may not be available on resource-constrained edge devices [17].

Our work lies in the intersection of blockchain technology and embedded systems, leveraging the strengths to protect against physical fault injection attacks. Hence, our blockchain implementation is sufficiently lightweight for embedded devices.

3 SYSTEM DESIGN

We now present the system's design, which we call Verified Instruction Execution (VIE). We first outline the attack surface analysis and usage scenarios, followed by the design overview and components.

3.1 Threat Model

Our threat model primarily focuses on hardware-based fault injection attacks on edge devices executed with physical access to the device. These attacks may target both hardware and software layers. The attacker may utilize various techniques such as clock or power glitching, laser fault injection, or electromagnetic pulses to induce errors in the system. The objective is to alter the normal execution of instructions on the device, such as tampering with specific instructions (e.g., skipping them) or modifying critical data stored in memory (RAM or registers). Such attacks can lead to unauthorized access, data leakage, or bypassing security protocols.

The threat model assumes that the attacker is sophisticated, possessing the technical knowledge needed and the equipment to perform fault injection attacks. However, the attacker is constrained by the integrity mechanisms of the VIE system. The adversary won't be able to completely disable the hardware and software stack, or the VIE mechanism by reprogramming the device or exploiting vulnerabilities in VIE itself. Attacks that fall outside these constraints are considered to be beyond the scope of this paper and the VIE system's designed defenses.

3.2 Attack Surface Analysis

A comprehensive security design requires understanding the potential attack vectors and vulnerabilities that an attacker might exploit. In the context of our system, the attack surface comprises the interfaces and functions exposed by the system which an attacker could target. The surface analysis consists of physical attack vectors on hardware and software vulnerabilities.

Physical Attack Vectors: As many edge systems are situated in physically accessible environments [1, 2, 6, 8, 12, 22, 25, 25], they are susceptible to various physical fault injection attacks. The impact of these attacks can lead to a broad spectrum of outcomes, from disrupting the normal functioning of a system to revealing sensitive information.

Software Attack Vectors: Software vulnerabilities offer a substantial attack vector, primarily due to the sheer number and variety of potential exploits. These can be combined with the physical attack vectors to initiate a successful exploit. These types of vulnerabilities can arise for multiple reasons that can be combined with hardware attack vectors, such as errors in coding or software design, buffer overflows, use of insecure functions, and race conditions. Misconfigurations, such as improper file permissions, weak passwords, or misconfigured services, can provide an attack surface that may be exploited. Finally, weak or improperly implemented cryptographic algorithms can lead to the exposure of sensitive data. Our design focuses on software vulnerabilities that can be exploited more easily by physical fault injection attacks.

3.3 Usage Scenarios

We designed the VIE engine to facilitate blockchain-protected and verifiable computing on edge devices, defending against multiple fault injection attacks. Therefore, we envision the user using VIE as a library that can be linked during compilation time. Due to the introduced overhead and slowdown effects, we do not envision VIE being used to run entire user applications. Instead, it is used to strengthen critical sections such as sensitive data processing, identity verification, and similar vulnerable tasks that are most likely exploited.

More specifically, VIE works for user apps at the function level and puts a function into *protected* mode. This is achieved by compiling the function into a specific section at a certain memory address. For example, suppose we want to protect a function **foo()**, we need to specify it as follows:

Listing 1: C code example

```
void  __attribute__((section(".protected")))
foo() { ... }
```

Here, the **section** attribute specifies that the function **foo()** is compiled into a specific section called **.protected**. The linker script then specifies that this section is placed at a specific address in memory. If enabled and linked, the VIE engine then uses this address to fetch instructions one by one and execute them in the protected mode. At the end of the execution, the VIE engine will perform the necessary checks. If the **.protected** is not used, all functions are executed in the normal mode.

3.4 Design Overview

At a high level, Figure 2 shows the overview of VIE, where VIE is used to protect critical segments of instructions. As shown in this figure, once the device is reset, the microcontroller runs initialization routines. It starts the scheduler, which loads one or more user processes into memory for execution. We assume the most basic scenario where a simple round-robin scheduler is used. Each process has protected and unprotected code segments, depending on whether they are critical. The red boxes are unprotected, while the green boxes are protected. Unprotected code is executed natively, while the protected code triggers the VIE controller and has to be placed into the protected segments during compilation time.

Figure 3 shows the sequence of protected instructions, where the VIE controller loads instructions and calculates the blockchain state in each step of execution. For this purpose, VIE uses multiple meta parameters, such as the number of instructions and the type of instruction, to calculate their hash values during this procedure. The hashed value is propagated through the execution steps until, at the end, all instructions are executed. Blockchain can also be enabled for every T instructions to reduce overhead. In our prototype, we did not turn on this feature so that we could evaluate the worst-case performance overhead.

During the execution cycle of the VIE controller, three main tasks are performed: logging metadata, hash value calculation, and saving the final hash value at the end. Since VIE is designed to attest that a sequence of instructions is correctly executed, it needs to run the execution multiple rounds to compare the final outputs. Further, to prevent the adversary from using timing information to infer the instructions being executed, the VIE-protected program actively inserts random delays ranging from a specific interval of values. At each point, VIE marks the current instruction into "waiting"

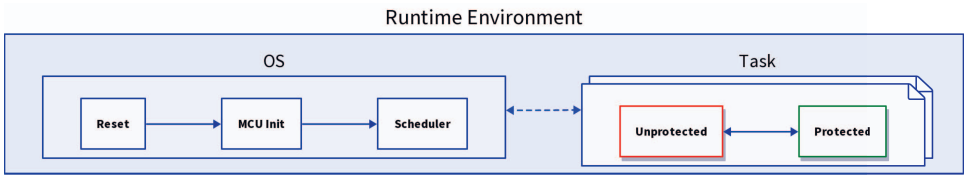

Figure 2: Workflow of the VIE protection system, illustrating the sequence of initialization, scheduling, and execution of protected and unprotected code segments. The green and red boxes represent protected and unprotected code, respectively.

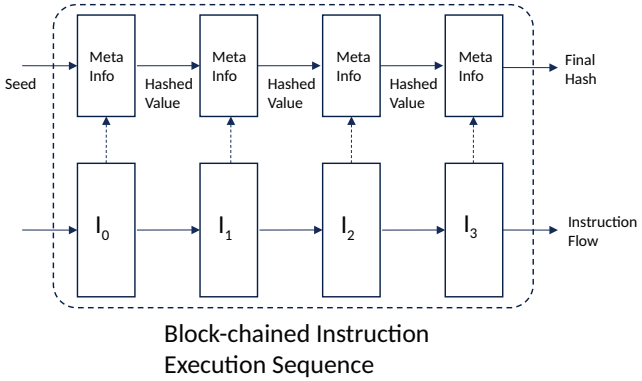

Figure 3: The operation of the VIE controller in sequence execution, employing blockchain to calculate hash values of protected instructions. The figure also illustrates how the propagation of hashed value occurs through each execution step.

mode and proceeds only after a temporal counter counts down to zero. Since everything is run on an embedded system, we minimize using computationally expensive routines such as random number generators. Hence, the random interval is emulated by picking from a sequence of values using the modulo operation. Such design choices allow the VIE controller to run as fast as possible.

3.5 Design Choices

As illustrated earlier, we focus on protecting critical instructions such as key authentication operations. To ensure the low overhead of the virtual execution employed by VIE, this section presents the critical design choices on how we achieved our goal. We will explain the details of virtualized execution, blockchain verification, and address randomization.

3.5.1 Virtualized Instruction Execution. As the name suggests, VIE engine employs Virtualized Instruction Execution to significantly enhance the security of edge devices against potential attackers attempting to perform attacks such as skipping necessary instructions or tampering with register values. Specifically, VIE develops a sophisticated approach by simulating the instructions and managing multiple executions of the instruction flow to attest the integrity of the execution process. In the subsequent sections, we discuss how these two techniques collectively make the VIE engine a trustworthy safeguard against fault injection attacks.

Simulating Instructions: The cornerstone of the VIE lies in its ability to simulate instruction execution on embedded platforms. By replicating the details of each operation, memory access, and I/O operation, VIE safeguards critical instruction segments and thwarts any attempts of unwarranted intrusion or manipulation.

The simulation process starts by loading instructions one at a time from a specific memory location, denoted as the **.protected** section. The VIE engine sequentially fetches these instructions. It then simulates each operation involving operands in a manner akin to an actual execution scenario. The VIE engine also simulates memory access during this phase by providing translated access to RAM memory segments. While this by itself cannot guarantee that the integrity of the memory contents and any attempts to modify the memory contents are detected, through multiple rounds of execution, any change to RAM contents by potential adversaries will cause differences in the blockchain state at the end of execution rounds and get detected. Finally, the engine simulates I/O operations for the given instruction segment. This includes tracking interactions with the system's peripherals and redirecting all I/O instructions through the VIE module before forwarding them to real I/O locations.

Adding Randomized Delays: Following the simulation of instructions, the VIE introduces an innovative second layer of defense: implementing randomized delays. This strategy is rooted in preventing any external attacker from attempting to gain insights into the system's internal workings through timing side-channel attacks. More specifically, upon the execution of a sequence of instructions in the protected section, the VIE engine introduces a random delay, which is chosen from a pre-defined range of values, ensuring that the delay introduced is neither too small to be ineffective nor too large to impact the system's performance adversely.

The addition of random delays has a profound impact on an attacker's ability to decipher the instruction execution pattern. For an observer, the timing information of the instruction execution becomes erratic and unpredictable. This randomness effectively prevents attackers from predicting the next instruction or the stage of execution. For example, if an attacker attempts to infer the execution order by analyzing the I/O behavior or memory access pattern, the randomized delays will render their analysis ineffective.

3.5.2 Blockchain-based Verification. The VIE engine incorporates the innovative use of blockchains to verify the instruction flows. This methodology provides enhanced security for the instruction execution process, which we describe in detail.

Blockchain Calculation Using Metadata: When VIE simulates instructions, it generates multiple types of metadata, such as the number of cycles, the current register contents, and the type of

the current instruction, among others. Blockchain verification calculates hash values based on these various types of metadata. One notable exception is that we cannot employ timing data due to the integration of random delays, which distort the timing analysis. To navigate around this, the VIE engine strategically selects metadata from each instruction execution as follows:

(1) Before the first instruction, a seed value is chosen as the hashing value. For one instruction, e.g., the instruction I with operands A and B, the next hash is computed by **Output = HASH(Prev, MetaInfo(I,A,B))**, where the **Prev** is the previous hash value. The **MetaInfo** is the metadata of the current instruction. In practice, we use a combination of multiple types of metadata, including the instruction type and the program counter, as well as operand values, in the blockchain calculations. This will make it more challenging for an attacker to tamper with the metadata without being detected.
(2) This procedure is repeated for every instruction within the critical region. This repetition ensures that the blockchain captures the details of each instruction and the order in which they were executed.
(3) After all instructions have been executed and their respective hashes computed and chained, the final hash value is produced as the output. Such output is considered the signature of the execution round and is used to verify the integrity of the execution process.
(4) Each simulated execution creates new random delays, forming a unique timeline for each execution round. This randomness is propagated through the resulting blockchain, ensuring its unpredictability and security.

The use of blockchain in our context brings benefits as follows. The first benefit is immutability. Since the blockchain calculates blocks based on previous ones, once data is recorded on a blockchain, it is extremely difficult to alter without affecting all the following blocks, ensuring a high level of data integrity. This makes a blockchain more robust than an append-only log. The second benefit is auditability: The blockchain can serve as a permanent record that can be audited at any time to verify the correctness of instructions in our context. Each sequence of instructions will generate a unique sequence of hashed blocks that can be audited for correctness.

Multiple Execution Rounds: The VIE engine enforces another layer of security by implementing multiple rounds of execution for the same critical section. The key idea is that if any instructions were tampered with, skipped, or altered in the execution process, this would lead to a disparity in the final hash values, thereby indicating a breach in the execution process. To detect such anomalies, the VIE engine compares the final hash value through at least two rounds of executions. If the two values match, the execution process is deemed secure. Otherwise, the VIE engine flags the execution as compromised. By implementing the blockchain mechanism alongside the VIE, a robust defense structure that maintains the integrity of crucial instruction execution and the reliability of edge devices is established.

We clarify that we do not assume a fully trusted environment for re-execution. Instead, the VIE mechanism will verify the correctness of the execution by comparing the hash values at the end of re-executions. Hence, it does not assume each re-execution is not compromised. Instead, we assume that if there is a compromise, it is very challenging for the attacker to compromise the re-executions in exactly the same way. That is, during re-executions, the attack will almost always cause the final hash value for the blockchain for multiple executions to be different, which causes the compromise to be detected. To this end, we recognize that maintaining a consistent seed across multiple executions is essential for accurate hash value calculation.

Memory Randomization: One of the key challenges in safeguarding against fault injection attacks is protecting specific critical memory contents, such as registers and RAM for storing encryption keys, from unauthorized access or tampering. In such attacks, the adversary can modify the memory or register value through various physical means, temporally or permanently. VIE addresses this concern by incorporating memory randomization, a sophisticated strategy that drastically increases the difficulty for potential attackers to access and modify memory locations of critical data.

Note that to ensure consistency over multiple rounds of execution, instead of using randomized memory addresses in the hash value calculations, we only use the logical address of variables before it is randomized, so that the randomization process does not interfere with the program's control flow. Hence, the VIE engine can maintain consistent hash values in the presence of memory randomizations. An advantage of this design is that it does not require any changes to the program's source code involving memory accesses, which is essential for the VIE engine to be used as a library.

Recall that we use simulation-based execution for instructions in VIE. Hence, VIE allocates variables for protected memory contents through randomized memory locations rather than fixed, predictable addresses. The uncertainty is controlled during compilation time by remapping the memory contents dynamically. This hampers any attempt by attackers to modify these variables, as they cannot determine the precise memory locations where the variables reside. More specifically, the memory randomization process takes into account three critical inputs: (1) the location of the register simulation, (2) the location of the memory simulation, and (3) the location of the I/O simulation. The rationale behind these inputs is as follows:

Register Simulation Location: As instructions typically have one or more operands stored in registers, the VIE engine naturally generates a set of variables for representing registers during simulation, whose locations are unknown to the attackers.

Memory and I/O Simulation Locations: In the simulations, direct accesses to critical and protected memory-stored variables are avoided as it could lead to a map revealing the location of variables. Instead, memory values are mirrored by VIE-owned variables. This approach requires an additional verification step to determine whether an operation is valid according to the address of accesses. Further, the VIE engine also simulates I/O operations by redirecting all I/O instructions through the VIE module before forwarding them to real I/O locations. This strategy ensures the VIE engine can monitor and verify all I/O operations.

4 IMPLEMENTATION

This section presents the implementation of the VIE engine on an edge system. For this study, we chose the AVR microcontrollers [5], a family of microprocessors developed by Atmel, now a part of Microchip Technology. There were several key reasons for selecting the AVR microcontroller architecture for this study. Firstly, these microcontrollers are extensively utilized in various embedded systems due to their versatility and efficiency, making them suitable for a broad range of real-world applications. Secondly, the AVR architecture, with its unique features and lack of inherent security mechanisms such as memory protection, provides a suitable platform to demonstrate the effectiveness of the VIE engine in enhancing system security.

We emphasize that although we picked the AVR platform in the implementation, our work can be extended easily to other microcontrollers, such as ARM, with minor modifications. The reason is that the VIE engine is designed to be portable and easily adapted as it is implemented entirely in C and can be adapted to support different types of instruction sets.

4.1 AVR Architecture Overview

AVR microcontrollers are renowned for their 8-bit RISC architecture and Harvard architecture design. These devices stand out due to their in-system programmable flash memory and extensive on-chip peripherals [4]. In the following, we describe its instruction set and memory organization.

AVR Instruction Set: The AVR instruction set supports more than 100 instructions, most of which are single-word instructions capable of executing in a single clock cycle. A typical AVR instruction consists of opcode and operands, where opcode determines the operation, and operands specify the data on which the operation is performed. In our implementation of VIE, we do not need to support specific instructions that do not appear in critical protected sections, such as those for reprogramming flash memory, watchdog operations, and hardware-related operations, e.g., timers and EEPROMs. Instead, we focus on the instructions critical sections will encounter, such as ADD, SUB, AND, OR, JMP, BRNE, etc., and optimize their decoding to reduce the overall overhead.

Memory Organization in AVR: The memory organization of the AVR microcontrollers is different from other microcontrollers due to its division into program memory (Flash), data memory (SRAM), and EEPROM. In AVR, the memory map encompasses data and I/O memory space, where general purpose registers, I/O registers, and SRAM share a common address space. The Flash memory is used to store the program code, while the SRAM is used to store data and stack. The EEPROM stores data that needs to be retained even after the device is powered off. In our implementation, we simulate the access to RAM locations and I/O registers by mapping them to a randomized memory space. This ensures that the memory contents are protected from unauthorized access and modification.

4.2 Overview of VIE Implementation

The Virtualized Instruction Execution (VIE) engine is designed to provide a robust defense mechanism against fault injection attacks on AVR microcontrollers. This is achieved by moving the execution of instructions into a protected space and introducing random delays to obfuscate the execution flow. In doing so, VIE deters timing and fault injection attacks, enhancing the edge devices' security.

4.2.1 Instruction Simulation. To ensure secure instruction execution, the VIE engine simulates each operation, leveraging the properties of the AVR architecture. In the following, we describe each step separately.

Running the Instructions: The VIE engine fetches instructions from the protected memory section, ensuring serialized order of execution. As each instruction is of a fixed length, the decoder in VIE parses the next instruction and maintains the order of their processing. The VIE engine mitigates risks associated with instruction hijacking or manipulation by creating a secure and predictable sequence of actions for instruction fetching.

After fetching, the VIE engine simulates the execution of each instruction. This includes data processing operations, memory access operations, and I/O operations. Specifically, the VIE engine performs actions according to the specific instruction and also maintains a CPU status flag to indicate the current instruction execution result. It ensures that the integrity of the operation is maintained throughout the process. This secure simulation prevents malicious alterations during instruction execution, safeguarding the embedded system.

Managing Memory Contents: Memory management is another critical feature of the VIE engine. During instruction simulation, the engine carefully tracks the memory contents and ensures that any attempt at unauthorized modification is detected and mitigated. This ensures the preservation of data integrity during the execution process.

More specifically, memory content accesses to sensitive data are performed indirectly by mapping the original variables used in a program into a randomized memory space. The VIE engine protects this randomized memory space, maintaining the mapping throughout the execution. This ensures that the memory contents are protected from unauthorized access and modification. To save overhead, during the compilation process, the user can specify protected and non-protected memory data, depending on whether they are sensitive and hold protected data. In addition to internal operations, the VIE engine also simulates I/O operations. It tracks all I/O interactions and directs them through the secure controller.

4.3 Adding Randomized Delays

Integrating randomized delays into the VIE engine is strategic for enhancing security against timing-based side-channel attacks. In our implementation, the VIE engine incorporates random delays into the execution flow. The engine selects delay durations from a predefined range at random and integrates them into the execution flow after one or more instructions, depending on the configuration. This ensures that the delays are unpredictable, thereby enhancing system security.

More specifically, we use the linear congruential generator to generate random delays. This generator is a type of low-overhead pseudorandom number generator that operates in a deterministic manner, as defined by the recurrence relation:

$$X_{n+1} = (a * X_n + c) \pmod{m} \tag{1}$$

In this formula, the values a, c, and m are integer constants, and X is the sequence of pseudorandom values. The choice of a, c, and m values is crucial for the period and quality of the generated pseudo-random numbers. In practice, we choose m as a power of 2, which is the size of the register. We also choose a and c such that $a - 1$ is divisible by all prime factors of m and c is relatively prime to m. This ensures that the generator has a total period of m.

4.4 Blockchained Protection

To protect the integrity of instructions, VIE is designed to operate in a blockchain manner. This ensures the engine can detect and mitigate any unauthorized modifications to the execution flow.

Upon experimentation, we use the implementation shown in Figure 4 for blockchain protection. This implementation is initialized with a seed value, and the next value is generated by multiplying the previous value by a constant and adding another constant. The VIE engine is implemented in C and is designed to be portable across different AVR microcontrollers. The engine is designed to be modular, with each module performing a specific task. The modules are designed to be independent of each other, thereby allowing for easy integration into existing systems.

5 EVALUATION

5.1 Overview

In this section, we provide an overview of the experiments. Specifically, we perform experiments using a suite of six benchmarks, each representing a critical code segment. These benchmarks are not designed as entire user applications. Instead, they serve as code to be protected by VIE. Their functions are shown in the following table. We select these benchmark programs as they represent a range of common data-intensive operations in security-sensitive applications, such as CRC calculations for checksums, matrix operations (e.g., commonly used in neural network models and edge AI applications), and password verification, among others. Our experiments are performed on the AVR ATxmega384c3 microcontroller using the Microchip Studio environment, which was chosen due to its larger RAM size available for the experiments.

Benchmarks	Description
CRC	Calculating CRC for data
Matrix	Basic matrix operations
Motion	A simple signal filtering algorithm for detecting motion
Sort	Performing a sorting operation
RND	Random number generator
Password	Authentication of user password

Table 1: Benchmarks and their descriptions

5.2 Slowdown for Protected Execution

We first analyze the slow-down effect of the VIE execution for protected code. In this experiment, we compare the execution time of the protected code with the native execution time and measure the number of CPU cycles needed in each execution. For the VIE-based execution, we also measure the effect of introducing blockchain, where we measure the number of CPU cycles needed with and without the blockchain operations. For each program, we test with three compiler settings, including **-O0**, **-O1**, and **-O2**. The evaluation figures demonstrate the inherent trade-offs in securing code execution using VIE and blockchain. While there are apparent increases in computational demand, these need to be viewed in light of their enhanced security. We will discuss their overall effects on applications later.

In Figure 5, we compare the computational overhead induced by VIE. The figure compares the performance under native execution, VIE, and VIE with integrated blockchain operations for six benchmarks. The performance metric is the number of CPU cycles, and the data is represented on a logarithmic scale due to the significant differences among the measurements.

A clear trend across all benchmarks is the increased computational overhead when transitioning from native to VIE execution. This is an anticipated result, given that the VIE execution encompasses more stages in processing the code, which inherently requires more CPU cycles.

The benchmarks display varying degrees of runtime increase, indicating that the overhead of VIE execution depends on the benchmark's nature. Some benchmarks are more affected by the transition from native to VIE execution than others, possibly due to the differences in the execution paths and the complexity of operations each benchmark undertakes.

One consistent observation is that introducing blockchain operations into the VIE execution increases the computational demand across all benchmarks. This signifies the additional overhead the blockchain operations impose in ensuring the security and verifiability of the execution. While this increase may appear significant, it is essential to consider the trade-off between the added security assurances and the computational overhead.

In the context of the **O1** and **O2** compiler settings, we still observe increased CPU cycles for both VIE and VIE with Blockchain compared to the native execution. However, this increase is less pronounced than with the **O0** setting, suggesting the potential for a more balanced trade-off between computational efficiency and the enhanced security provided by VIE and Blockchain. For example, in benchmarks ranging from simple tasks such as CRC to more computationally intensive tasks like RND, the **O1** setting consistently requires fewer additional CPU cycles than the **O0** setting. These observations underscore that the choice of compiler settings can significantly influence the performance-security trade-offs and highlight the potential advantages of the **O1** and **O2** setting for specific applications.

Next, we demonstrate the slowdown factor, defined as the ratio between the number of CPU cycles with VIE and the native execution. The results for **O0** are shown in Figure 8. The chart provides an interesting perspective into the additional computational overhead the VIE and Blockchain mechanisms introduced. Here, it is noticeable that the slowdown factor is more significant for VIE with Blockchain than VIE alone across all benchmarks. This trend highlights the inherent overhead introduced by blockchain operations on top of the already intensive VIE mechanism. However, it is essential to reiterate that these measurements pertain only to the protected code regions, not to entire user applications. The overall impact on application performance will depend on the

```
uint32_t block_chain_calc(uint32_t prev, uint32_t instruction_count, uint32_t instruction_type
    uint32_t hash = prev;
    hash += instruction_count;
    hash += instruction_type;
    hash = hash * 0x5DEECE66DL + 0xBL;
    hash = (hash & 0xFFFFFFFF0000FFFF) | (hash >> 32);
    return hash;
```

Figure 4: Example code for calculating the blockchain state.

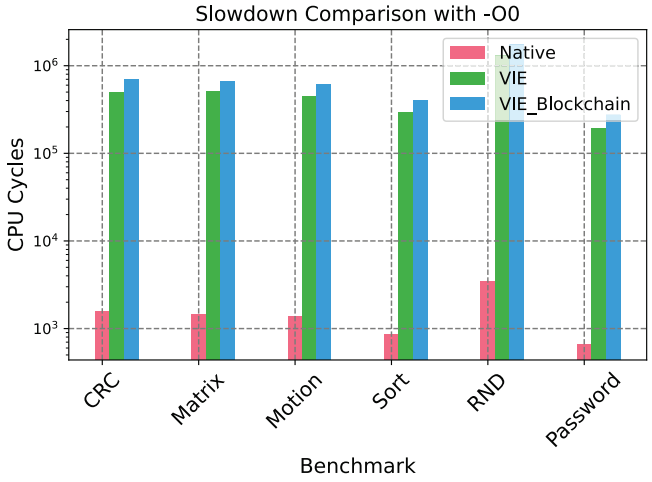

Figure 5: Slowdown Comparison with -O0

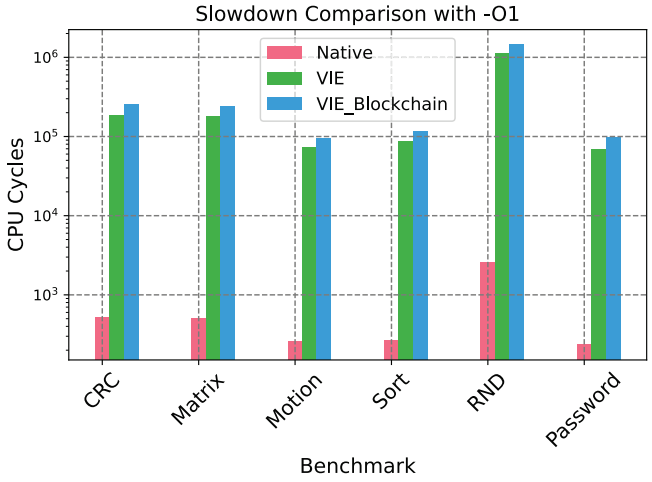

Figure 6: Slowdown Comparison with -O1

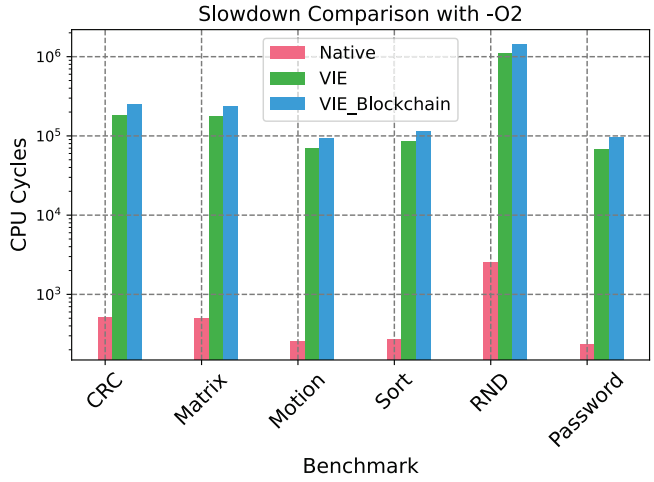

Figure 7: Slowdown Comparison with -O2

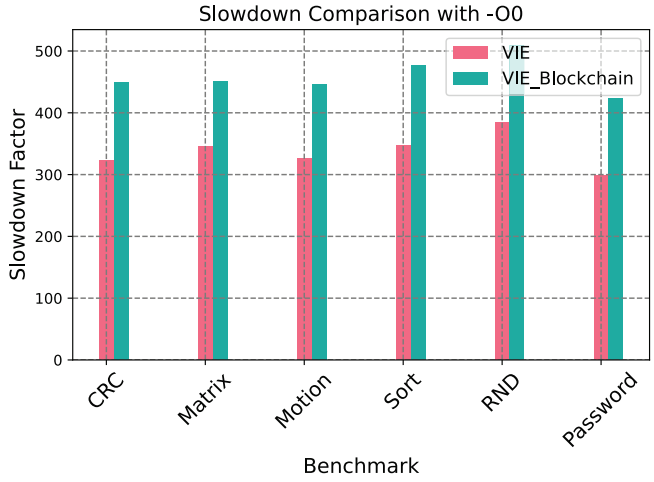

Figure 8: Slowdown Comparison with -O0

protected code proportion and the specific computational demands of that code. Consequently, this view underscores the importance of a careful and strategic application of these protection mechanisms, balancing the benefits of robust security against the potential performance impacts.

We next show the results for comparison using **O1** and **O2** settings. Across both the **O1** and **O2** optimization levels, the slowdown factor remains consistently higher for VIE with Blockchain compared to VIE alone, mirroring the observations at the **O0** level. This is an expected outcome, given that the Blockchain operations will add further computational overhead on top of the VIE mechanism.

However, interesting patterns emerge when comparing the slowdown factors across the different optimization levels. For both VIE and VIE with Blockchain, the slowdown factors seem relatively

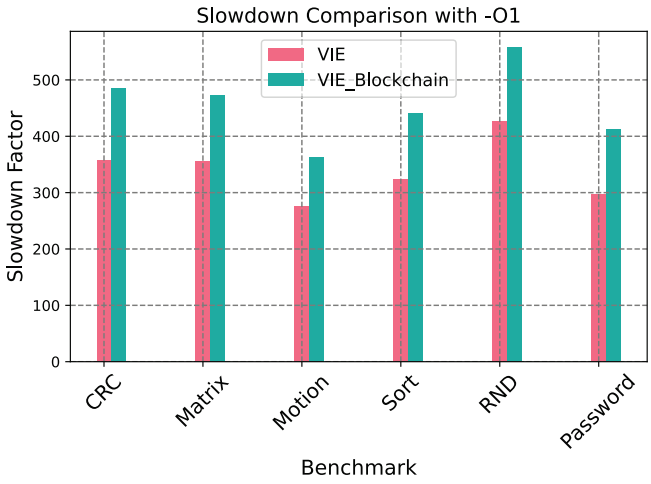

Figure 9: Slowdown Comparison with -O1

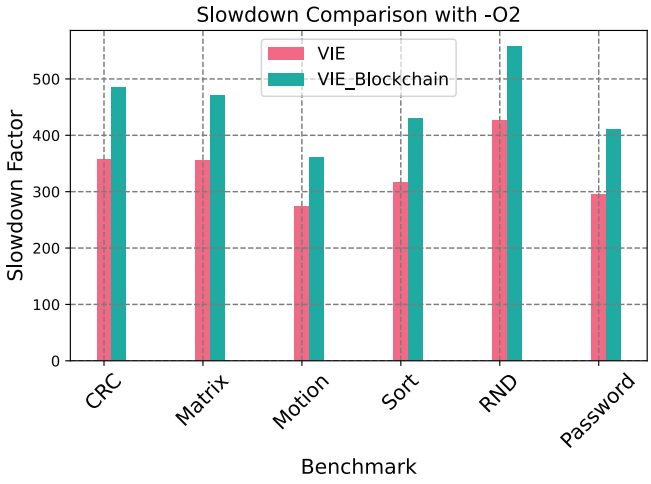

Figure 10: Slowdown Comparison with -O2

stable from **O1** to **O2**, with slight reductions visible in most benchmarks. This can likely be attributed to the more efficient code generated at higher optimization levels, reducing the overall overhead. These observations highlight how compiler optimizations can play a significant role in managing the performance impacts of security mechanisms like VIE and VIE with Blockchain and how these performance impacts need to be considered in the broader context of system optimization strategies.

We next demonstrate the aggregate slowdown factor when we consider the protected region is only taking a small percentage of the total number of CPU cycles. As shown in Figure 11, a nearly linear increase in aggregate slowdown is evident as the percentage of protected code regions increases. This pattern manifests consistently across all tested benchmarks, ranging from CRC to Password, indicating that the trend holds regardless of the unique attributes of the benchmarks themselves. This linearity in the aggregate slowdown suggests that the overhead introduced by the

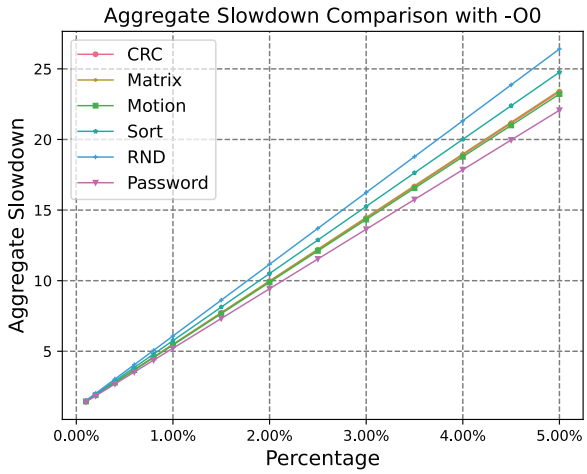

Figure 11: Aggregate Slowdown Comparison with -O0

protective mechanisms is directly proportional to the extent of the code region under protection.

Despite the slowdown, the overall impact on execution time is generally acceptable, particularly given the security benefits of the protective measures. For instance, if we consider a hypothetical program that typically executes in 0.1 seconds, a slowdown factor of 10x would result in a new execution time of 1 second. While this is indeed a significant increase, the actual impact in terms of real-world time is relatively minimal, merely adding 0.9 seconds. In many application scenarios, this could be a very acceptable trade-off when enhanced security protections are considered. Therefore, this evaluation suggests that developers can afford to protect a substantial portion of their code without incurring prohibitive performance penalties, thereby enhancing the security robustness of their applications.

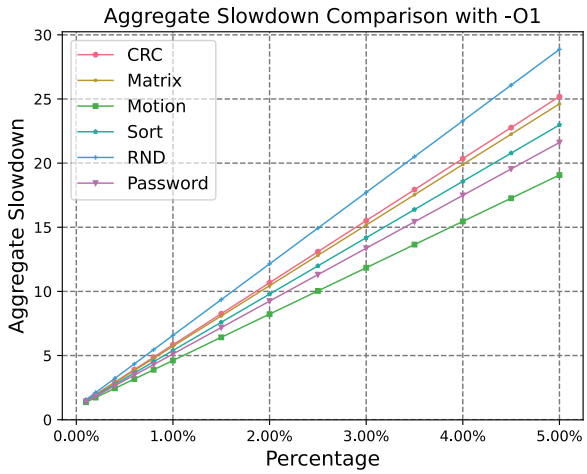

Figure 12: Aggregate Slowdown Comparison with -O1

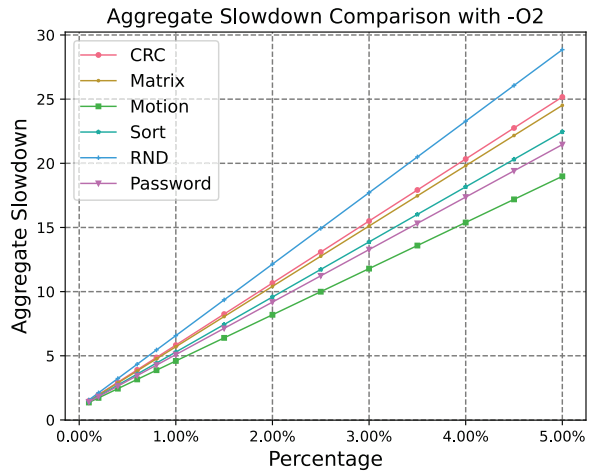

Figure 13: Aggregate Slowdown Comparison with -O2

We also show the evaluation results at **O1** and **O2** settings, where we notice a similar pattern of a near-linear increase in aggregate slowdown with the percentage increase of protected code regions, just as we observed with the **O0** setting. This again validates the understanding that the overhead introduced by the protective mechanisms directly correlates with the proportion of code under protection.

When comparing the **O1** and **O2** settings with the baseline **O0** setting, it's evident that the degree of slowdown is somewhat increased. However, the slope of the increase remains relatively consistent across all optimization settings. This suggests that while the aggregate slowdown may be higher at more aggressive optimization levels, the rate of this slowdown does not drastically alter. This consistency in the rate of slowdown across optimization levels reinforces the overall acceptability of the security overhead.

5.3 Emulating Physical Fault Injection Attack

To rigorously assess the efficacy of our VIE engine in safeguarding against fault injection attacks, we simulate a variety of such attacks on the AVR microcontroller. These attacks aim to introduce faults during the microcontroller's operation to induce unintended behavior and potentially compromise sensitive data.

Our case study involves multiple types of hardware-based fault injection attacks:

- Instruction skipping: This attack involves skipping an instruction. One practical way to introduce this attack involves intentionally causing a sudden drop in the microcontroller's supply voltage for a brief period. This voltage drop can cause the microcontroller to misinterpret instructions or data, leading to faulty operation.
- Instruction repeating: This attack involves the repetition of an instruction.
- Register bit/byte-flips: This attack involves changing a controlled critical bit/byte value to its opposite value in a register.
- Random byte: This attack modifies a value of a particular in-memory byte associated with a critical variable to a random value.

5.3.1 Fault Injection Setup. We execute each benchmark on the microcontroller, typically resulting in the blockchain end state being a specific value. Next, we introduce one of the attacks above for the protected instruction sequence while VIE is enabled. To precisely control the fault injection, we emulate the faults using the Microchip Studio Simulator instead of injecting faults on real hardware so that we can repeatedly inject the same fault for controlled comparisons. All attacks are introduced after the first round of normal execution for a protected segment of code. After the attack is introduced, the blockchain state is compared with the original value.

5.3.2 Effects of Fault Injection. Our experiments show that the physical fault injection attack indeed alters the blockchain end state across all benchmarks with different attack modes, as observed based on the built-in debugger to view the blockchain final state. That is, the final state of the blockchain is different from what it should be if the instructions had been executed correctly. This shows that the fault injection attack is effective at altering the execution flow of the microcontroller.

5.3.3 Effectiveness of VIE Engine. In our experiments with the attacks turned on, once the VIE engine detects a compromised blockchain state in repeated executions, the error handler routine is triggered, and the normal program execution stops. This demonstrates that the VIE engine can protect the AVR microcontroller from fault injection attacks.

There are several reasons why the VIE engine is effective at thwarting these attacks. Firstly, whenever a fault is injected, the actual instruction being affected will cause unpredictable changes to the blockchain state calculations. Second, the VIE engine introduces randomized delays into the execution flow, disrupting the timing of the fault injection attack and preventing the attacker from introducing the fault at the same point. Therefore, even if the same attack is injected in two rounds of executions, the ending state of the blockchain is still different. Finally, the VIE engine constantly checks the blockchain state and detects if it differs across execution rounds, thereby identifying any potential attacks.

In summary, through emulation of fault injections on the AVR microcontroller, we demonstrated that the VIE engine can effectively thwart such attacks, ensuring the integrity of the blockchain and enhancing the security of the edge system.

6 CONCLUSIONS

This paper presented an innovative and comprehensive approach to address the growing threat of hardware-based fault injection attacks on edge devices. Recognizing that existing defenses such as ARM's TrustZone are not sufficient to fend off these types of sophisticated attacks, we proposed a novel combination of technologies that are designed to offer robust and effective security protections for edge devices without relying on any specific hardware features. Our approach consists of a three-pronged defense strategy, including a new type of virtual machine-based verification for secure code execution, a blockchain-based attestation mechanism for instruction integrity assurance, and a memory randomization technique for

safeguarding data integrity. Through this approach, we were able to effectively protect critical sections of code from being tampered with or skipped, ensuring that instructions are executed in the correct order and with the correct values. Furthermore, by recording execution traces into a blockchain, we provided a powerful method to detect any discrepancies arising from instruction tampering.

To evaluate the practicality and effectiveness of our approach, we implemented these mechanisms on an AVR-based microcontroller and conducted a thorough evaluation using six benchmark programs. Our results showed that our solution can defend against various adversary attacks, including instruction-skipping attacks and byte-flipping attacks, among others, demonstrating the robustness of our methodology. These results demonstrate that our solution is particularly effective in the real world, such as protecting edge devices that perform password checks, user authentication, and intrusion detection. Looking forward, we believe our work contributes significantly to safeguarding edge devices from physical fault injection attacks. The solutions presented in this paper could have broad implications for the field of hardware security, and we anticipate that they will inspire further research and development in this area.

The source code for this paper is available at https://github.com/utk-security/blockchain-attestation.

ACKNOWLEDGMENTS

This work was supported in part by the National Institute of Food and Agriculture/USDA under Grant 2023-67021-40613. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. 2017. Nanofocused X-ray beam to reprogram secure circuits. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Springer, 175–188.

[2] Anna L Duque Antón, Johannes Müller, Mohammad R Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2023. Fault Attacks on Access Control in Processors: Threat, Formal Analysis and Microarchitectural Mitigation. *IEEE Access* (2023).

[3] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H Leber. 2003. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on computers* 52, 9 (2003), 1115–1133.

[4] Steven F Barrett and Steven Frank Barrett. 2010. *Embedded Systems Design with the Atmel AVR Microcontroller*. Morgan & Claypool Publishers.

[5] Steven F Barrett and Daniel J Pack. 2012. *Atmel avr microcontroller primer: Programming and interfacing*. Morgan & Claypool Publishers.

[6] Yohannes B Bekele, Daniel B Limbrick, and John C Kelly. 2023. A Survey of QEMU-based Fault Injection Tools & Techniques for Emulating Physical Faults. *IEEE Access* (2023).

[7] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. 2019. Shaping the glitch: optimizing voltage fault injection attacks. *IACR transactions on cryptographic hardware and embedded systems* (2019), 199–224.

[8] Jakub Breier and Xiaolu Hou. 2022. How practical are fault injection attacks, really? *IEEE Access* 10 (2022), 113122–113130.

[9] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. 2021. SNIFF: reverse engineering of neural networks with fault attacks. *IEEE Transactions on Reliability* 71, 4 (2021), 1527–1539.

[10] Volkan Dedeoglu, Raja Jurdak, Guntur D Putra, Ali Dorri, and Salil S Kanhere. 2019. A trust architecture for blockchain in IoT. In *Proceedings of the 16th EAI international conference on mobile and ubiquitous systems: computing, networking and services*. 190–199.

[11] Jean-Max Dutertre, Timothé Riom, Olivier Potin, and Jean-Baptiste Rigaud. 2019. Experimental analysis of the laser-induced instruction skip fault model. In *Secure IT Systems: 24th Nordic Conference, NordSec 2019, Aalborg, Denmark, November 18–20, 2019, Proceedings 24*. Springer, 221–237.

[12] Aakash Gangolli, Qusay H Mahmoud, and Akramul Azim. 2022. A systematic review of fault injection attacks on iot systems. *Electronics* 11, 13 (2022), 2023.

[13] Lachlan J Gunn, N Asokan, Jan-Erik Ekberg, Hans Liljestrand, Vijayanand Nayani, Thomas Nyman, et al. 2022. Hardware platform security for mobile devices. *Foundations and Trends® in Privacy and Security* 3, 3-4 (2022), 214–394.

[14] Florian Hawlitschek, Benedikt Notheisen, and Timm Teubner. 2018. The limits of trust-free systems: A literature review on blockchain technology and trust in the sharing economy. *Electronic commerce research and applications* 29 (2018), 50–63.

[15] Yu-ichi Hayashi, Naofumi Homma, Takeshi Sugawara, Takaaki Mizuki, Takafumi Aoki, and Hideaki Sone. 2011. Non-invasive EMI-based fault injection attack against cryptographic modules. In *2011 IEEE International Symposium on Electromagnetic Compatibility*. IEEE, 763–767.

[16] Baydaa Hassan Husain, Shavan Askar, et al. 2021. Survey on edge computing security. *International Journal of Science and Business* 5, 3 (2021), 52–60.

[17] Sin Kuang Lo, Xiwei Xu, Yin Kia Chiam, and Qinghua Lu. 2017. Evaluating suitability of applying blockchain. In *2017 22nd international conference on engineering of complex computer systems (ICECCS)*. IEEE, 158–161.

[18] Jannik Lockl, Vincent Schlatt, André Schweizer, Nils Urbach, and Natascha Harth. 2020. Toward trust in Internet of Things ecosystems: Design principles for blockchain-based IoT applications. *IEEE Transactions on Engineering Management* 67, 4 (2020), 1256–1270.

[19] Dina G Mahmoud, Vincent Lenders, and Mirjana Stojilović. 2022. Electrical-Level Attacks on CPUs, FPGAs, and GPUs: Survey and Implications in the Heterogeneous Era. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–40.

[20] Kohei Matsuda, Sho Tada, Makoto Nagata, Yuichi Komano, Yang Li, Takeshi Sugawara, Mitsugu Iwamoto, Kazuo Ohta, Kazuo Sakiyama, and Noriyuki Miura. 2020. An IC-level countermeasure against laser fault injection attack by information leakage sensing based on laser-induced opto-electric bulk current density. *Japanese Journal of Applied Physics* 59, SG (2020), SGGL02.

[21] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. Ieee, 77–88.

[22] Onur Mutlu and Jeremie S Kim. 2019. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2019), 1555–1571.

[23] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[24] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. 2015. High precision fault injections on the instruction cache of ARMv7-M architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 62–67.

[25] Carlton Shepherd, Konstantinos Markantonakis, Nico van Heijningen, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann, and David Naccache. 2021. Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis. *Computers & Security* 111 (2021), 102471.

[26] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[27] Rickard Svenningsson, Henrik Eriksson, Jonny Vinter, and Martin Törngren. 2010. Model-implemented fault injection for hardware fault simulation. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation*. IEEE, 31–36.

[28] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. 2020. RusTEE: developing memory-safe ARM TrustZone applications. In *Annual Computer Security Applications Conference*. 442–453.

[29] Yinhao Xiao, Yizhen Jia, Chunchi Liu, Xiuzhen Cheng, Jiguo Yu, and Weifeng Lv. 2019. Edge computing security: State of the art and challenges. *Proc. IEEE* 107, 8 (2019), 1608–1631.

[30] Bin Yu, Jarod Wright, Surya Nepal, Liming Zhu, Joseph Liu, and Rajiv Ranjan. 2018. Iotchain: Establishing trust in the internet of things ecosystem using blockchain. *IEEE Cloud Computing* 5, 4 (2018), 12–23.

[31] Igor Zavalyshyn, Thomas Given-Wilson, Axel Legay, Ramin Sadre, and Etienne Rivière. 2021. Chaos duck: A tool for automatic iot software fault-tolerance analysis. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 46–55.