From Hardware Fingerprint to Access Token: Enhancing the Authentication on IoT Devices

Yue Xiao[†*], Yi He[‡*], Xiaoli Zhang[§], Qian Wang[†✉], Renjie Xie[‡], Kun Sun[¶], Ke Xu[‡], and Qi Li[‡✉]
[†] Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education,
School of Cyber Science and Engineering, Wuhan University,
[‡]Tsinghua University, [§]Zhejiang University of Technology, [¶]George Mason University,
[†]{yuexiao, qianwang}@whu.edu.cn, [‡]{heyi21, xrj21}@mails.tsinghua.edu.cn,
[‡]{xuke, qli01}@tsinghua.edu.cn, [§]xiaoli.z@outlook.com, [¶]ksun3@gmu.edu

***Abstract*—The proliferation of consumer IoT products in our daily lives has raised the need for secure device authentication and access control. Unfortunately, these resource-constrained devices typically use token-based authentication, which is vulnerable to token compromise attacks that allow attackers to impersonate the devices and perform malicious operations by stealing the access token. Using hardware fingerprints to secure their authentication is a promising way to mitigate these threats. However, once attackers have stolen some hardware fingerprints (e.g., via MitM attacks), they can bypass the hardware authentication by training a machine learning model to mimic fingerprints or by reusing these fingerprints to craft forged requests.**

In this paper, we present MCU-Token, a secure hardware fingerprinting framework for MCU-based IoT devices even if the cryptographic mechanisms (e.g., private keys) are compromised. MCU-Token can be easily integrated into various IoT devices by simply adding a short hardware fingerprint-based token to the existing payload. To prevent the reuse of this token, we propose a message mapping approach that binds the token to a specific request by generating the hardware fingerprints based on the request payload. To defeat the machine learning attacks, we mix the valid fingerprints with poisoning data so that attackers cannot train a usable model with the leaked tokens. MCU-Token can defend against adversaries who may replay, craft, and offload the requests via MitM or use both hardware (e.g., use identical devices) and software (e.g., machine learning attacks) strategies to mimic the fingerprints. The system evaluation shows that MCU-Token can achieve high accuracy (over 97%) with low overhead across various IoT devices and application scenarios.

I. Introduction

The emerging Internet of Things (IoT) technologies have been widely applied in various areas of our daily life. For instance, passive keyless entry (PKE) systems [36] can remotely unlock and activate the vehicles with a small key fob, and IoT hardware security tokens (HSTs) [11] are used to protect crypto wallets or login websites as universal two-factor (U2F) authentication. The cost-effective and power-efficient Microcontrollers (MCUs) are widely adopted by these IoT devices since they integrate CPU, RAM, ROM, and peripherals on a single chip. Meanwhile, the low cost and high integration also limit the hardware resources available on these devices (e.g., 256KB memory, 64-300MHz clock frequency). Also, IoT devices lack of hardware protection such as memory management unit (MMU) or trusted execution environment (TEE), rendering them less secure than mobile phones and laptops.

It is essential to ensure that MCU-based IoT devices are securely authenticated when interacting with other devices or the cloud [1], [33]. However, the existing token-based authentication solutions (e.g., JSON Web Token [5] and rolling code [9]) suffer from various attacks due to the constrained system resources and insecure implementations [63], [67]. For instance, Tesla key fobs are vulnerable to key clone attacks [3] and RFID/BLE relay attacks [10], [62], which allow attackers to activate vehicles by masquerading as valid keys or relaying communication to the real owner's keys. Moreover, single-function devices (e.g., U2F hardware keys [48] and hardware wallets [58]) can be easily cloned [1], [7] once attackers obtain the internal private keys during manufacturing, retail, or usage stages. Similarly, smart homes are at risk of token compromise attacks, which enable adversaries to impersonate legitimate devices, access user data, manipulate device status, and trigger malicious rules [30], [29].

The root cause of attacks against these IoT devices is that they can be impersonated, e.g., by compromising the communication protocols and secrets, so that the fake devices can generate the same requests to deceive their peers. Although unclonable hardware authentication factors have been proposed to prevent these attacks [21], [38], [24], [2], [49], [56], they are ineffective when they are applied to MCU-based IoT devices. First, most of the required hardware features (e.g., magnetic sensor [21], NAND-Flash [24] and TEE [2]) are not supported by most commercial-off-the-shell (COTS) MCUs. Although physically unclonable functions (PUF) [41] can produce device-specific crypto keys or fingerprints, they need extra integrity circuit (IC) manufacturing procedures to provide special circuits.

Second, it is still difficult to prevent the man in the middle (MitM) adversaries [61], [46], [33] that can mimic the hardware fingerprints via machine learning (ML) attacks or reuse previous fingerprints in forged requests. In particular, machine learning based attacks are the main threat to these hardware feature based solutions [51], [40], [57], where the attackers can

[*] The first two authors contributed equally to this paper.
[✉] Qian Wang and Qi Li are the corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.241231
www.ndss-symposium.org

collect the leaked fingerprints to train a ML model to mimic the hardware features and predict valid unused fingerprints. MitM attacks are real threats for various IoT devices including USB devices [13] (e.g., U2F and hardware wallets [25]), short distance devices (e.g., BLE and RFID Passive Keyless Entry (PKE) [10]), and WiFi and Ethernet devices (e.g., Smart Home [30]). Although secure communication protocols can prevent attackers from stealing fingerprints [56], [21], numerous real-world exploits indicate that it can still break these secure communications by constructing different attacks, e.g., remote exploiting [17], [18], stealing hard-coded crypto keys [6], and investigating unencrypted traffic in millions of IoT devices [23], [19].

In this paper, we develop a new authentication system called MCU-Token for MCU-based IoT devices, which generates access tokens based on the commonly supported hardware features. MCU-Token can ensure authentication security even if the existing cryptographic keys and algorithms are compromised. In particular, it can prevent MitM adversaries from crafting requests to reuse valid fingerprints when message integrity (i.e., signature) cannot be guaranteed. To achieve this, we design a one-round protocol that uses fingerprints to ensure the message's integrity and thus cannot be intercepted. We map the message with a nonce to a hash digest and utilize the digest bits to decide hardware fingerprinting methods and settings. MCU-Token can support six different hardware features to generate different fingerprints with thousands of different configurations. Consequently, it can produce tens of thousands of different fingerprints enabling multiple unique fingerprints for each request. Thus, once an attacker attempts to reuse a fingerprint, our MCU-Token backend authentication service can easily detect the attack by identifying a mismatch between the messages and fingerprints.

Moreover, in order to prevent attackers from obtaining fingerprints for model training even when the message confidentiality (e.g., algorithms and encryption keys) is compromised, MCU-Token injects noises to fingerprints that the attacker models trained on the fingerprinted are poisoned. If an attacker uses leaked fingerprints to train his model, the poisoned data cannot be used to train the model to accurately predict new fingerprints, and the MCU-Token backend authentication service can easily verify if the request is legitimate by checking if a portion of the fingerprints is valid. Thus, it can effectively defend against machine learning based attacks that cannot be throttled by existing hardware feature based defenses such as PUF [40], [51]. Meanwhile, data poisoning does not affect backend authentication because it uses multiple fingerprints for authentication each time, and only a few fingerprints are poisoned, allowing the unpoisoned ones to successfully pass authentication.

MCU-Token does not rely on the security of existing cryptography mechanisms on IoT devices considering many devices may lack hardware resources to enable strong encryption protection. Furthermore, MCU-Token is lightweight so that it can be applied on various resource-constraint embedded devices as it has small memory and storage footprints. The MCU-Token backend authentication service can also be easily deployed on normal IoT devices or on the cloud as it only uses simple machine learning algorithms such as RandomForest and ExtraTrees.

We prototype MCU-Token by ∼5100 LoC including ∼3900 lines of C code for the client runtime and ∼1200 lines of Python code for the MCU-Token's backend service, which is open source on Github[1]. We conduct experiments on different real-world devices, which demonstrate that MCU-Token is robust to existing attacks. Due to the difficulty of breaking our fingerprint binding mechanism, e.g., by manipulating payloads, attackers can hardly forge messages and reuse the fingerprints to bypass MCU-Token. The success rate of mimicking fingerprints via hardware or software approaches is less than 1%. In the meantime, MCU-Token achieves an average 97.78% true positive rate (TPR) with 4.87% false positive rate (FPR) in identifying 60 devices with 3 different MCUs. Moreover, the incurred overhead is reasonable low. The additional power consumption is less than 4% and the average extra authentication time is around 31ms.

In summary, our contributions are three-fold:

- We perform a systematic study on hardware features for fingerprinting the COTS MCUs. We explore six key hardware features and theoretically analyze and experimentally verify the sources of hardware uniqueness.
- We propose a new hardware fingerprint based authentication mechanism called MCU-Token, which utilizes a novel ML based design to protect device authentication without relying on cryptographic mechanisms. It binds fingerprints to specific requests and injects poisoned data to defeat different adaptive attacks, e.g., ML based attacks.
- We prototype MCU-Token and demonstrate its usability and performance by extensively evaluating it on 60 IoT devices of three types across three real-world scenarios, i.e., PKE/BLE key fobs, smart home sensors, and FIDO-U2F [12] hardware tokens.

II. Background

A. Hardware-based Authentication

Various hardware-based authentication mechanisms [24], [14] are proposed to secure IoT authentication. These approaches consider various hardware features such as physical signal characteristics [36], [26], magnetic characteristics [21], sensors with human interaction [49], [38], [59], or physically unclonable functions (PUFs) [42]. Based on the authentication methods, these approaches can be categorized into two types.

Hardware Fingerprint as New Device Identifier. These approaches utilize hardware-derived data to generate unique device fingerprints as the device identifier to distinguish different devices. The fingerprint data can be uploaded along with the payload or concealed as side-channel information, like physical signal strength [36] or time delay [24].

Hardware-Involved Challenge Response Authentication Protocol. Instead of directly identifying devices via static device identifiers generated by hardware features, challenge-response based approaches utilize diverse challenges as input to the hardware features, obtaining variable responses for authentication. For instance, the arbiter PUF [40], [15] can use different bits as input and statistic the relative delays in

[1] https://github.com/IoTAccessControl/MCU-Token

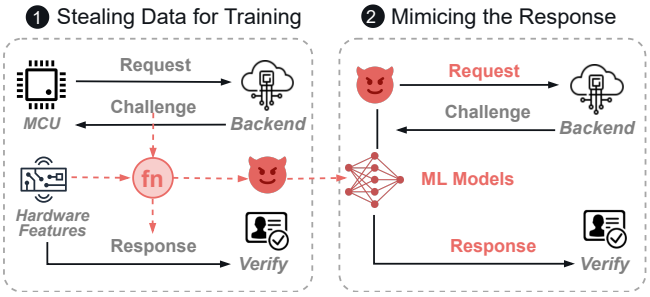
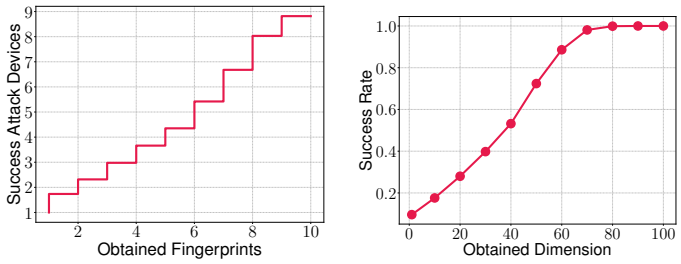


Fig. 1: Machine learning attacks to PUF-based challenge-response authentication. ❶ Attackers eavesdrop on the communication to steal challenge-response pairs. ❷ Attackers learn the mapping between challenges and responses via machine learning and mimic the responses to pass the authentication.



(a) Compromised device number (b) Attack with different data size

Fig. 2: Machine learning attack on IoT-ID [56]'s ADC feature.

the paths of the circuit as responses. These approaches collect enough challenge-response pairs (CRPs) and store them as key-value data [66] in the server or learned by ML models [20] during the device enrollment phase. In the authentication phase, the corresponding CRP is retrieved to verify if the response is coincident with a specific challenge. Human interactions can also be used as challenges, e.g. T2Pair [38] employs users' button or twist operations as challenges and validates the received actions on the server.

B. Attacks to Hardware-based Authentication

The MitM adversaries should be considered in IoT scenarios, as many IoT devices are resource constraint to adopt a secure implementation of TLS with SSL pinning [46], [27], or even do not encrypt the transmitted messages [19]. Under an insecure communication channel, attackers can launch two typical attacks:

Fingerprint Mimic Attacks. Attackers may attempt to replicate the hardware characteristics of target devices using identical hardware or alternative devices such as FPGAs. This threat is addressed by existing hardware fingerprinting studies because no two devices are truly identical at IC level, and devices of the same type can still be discriminated by their micro hardware features. However, attackers can also use software approaches (e.g., machine learning) to mimic the hardware features. Figure 1 shows the steps of machine learning attacks, where attackers can train a model based on a few existing fingerprints and mimic the new fingerprints. ML attack is a common threat to both hardware fingerprints based device identifiers and the challenge-response based authentication protocols. For instance, attackers can easily predict the responses of a given challenge for all existing PUF [51]. The hardware features of MCU can also be easily mimicked by machine learning. As shown in Figure 2, the features used by IoT-ID [56] can be mimicked with high accuracy after attackers gain less than 10 unique fingerprints.

Fingerprint Reuse Attacks. When the communication channel is insecure, attackers can eavesdrop and relay existing requests [63], [10], which is prevalent in existing RFID and BLE car key fobs. MitM attackers can also replay the existing requests to reuse the fingerprint or offload the challenges of servers to real devices to get valid responses. In this case, they do not need to mimic and generate fake fingerprints but can reuse the valid devices' real fingerprints. They can further modify communication data while reusing the existing authentication data (e.g., hardware fingerprints, PUF response), such as changing less sensitive commands (e.g., "turn on the light") into dangerous commands (e.g., "open door") after compromising the encryption and signature secrets.

Table I: The drawbacks in existing hardware-based authentication approaches. ○, ◑, and ● refer to unsupported, partial support, and full support respectively.

Approaches	Support COTS MCU	Resist Mimic Attacks	Resist Reuse Attacks
Signals [26], [35], [36], [32]	◑	●	◑
Human Interactions [38], [49], [59], [39]	◑	◑	○
PUF [15], [40], [41], [42], [45]	◑	◑	○
Hardware Fingerprints [37], [21], [24], [44]	◑	◑	○
Multiple Hardware Features (MCU-Token)	●	●	●

C. Limitations of Existing Hardware-based Authentication

As shown in Table I, existing works including both device identifier based or challenge response based approaches are all vulnerable to fingerprint mimic attacks and fingerprint reuse attacks. Currently, only signal-based approaches [35] can resist these attacks as the physical radio features are difficult to mimic and replicate. However, the physical signal-based approaches can only work on wireless (e.g., RFID [36] and BLE [32]) devices. This is also a common flaw of most of the existing works (except IoT-ID) that can only work on dedicated devices and are not practicable for COTS MCU. PUF-based approaches usually require special IC fabrication processes to produce hardware discrepancies, e.g., the arbiter PUF [42] requires additional arbiter circuitry and is only supported by dedicated devices (e.g., some types of NXP MCUs [8]).

IoT-ID [56] is the only work that supports general MCU-based IoT devices, which use commonly supported hardware features such as clock oscillators and ADC. However, it does not take adversaries into account and the hardware-based device identifier is just another access token that can be stolen by the attackers during transmission or at the server. Thus, it is still vulnerable to token compromise attacks.

III. Threat Model and Assumption

We assume attackers may have compromised the communication channel and stolen the authentication tokens of valid

devices. Their attack goal is to impersonate legitimate devices to perform malicious operations, such as unlocking a car by mimicing a key fob or triggering the execution [29] of trigger-action rules in smart homes by event spoofing [30].

As the attackers have compromised both the access control token and the encryption and signing mechanisms (if existing), they can eavesdrop and manipulate the requests of real devices or even impersonate the devices to send fake requests. To bypass the potential additional hardware-based authentication mechanisms (e.g., MCU-Token), they can perform fingerprint mimic attacks to generate valid fingerprint data via software or hardware approaches, such as collecting the existing hardware fingerprint data and training their own models to mimic the hardware behaviors (software mimic attack) or using the same types of hardware to mimic the real devices (hardware mimic attack). They can also reuse previous authentication information to send fake requests or forward the packages between the devices and the server (i.e., replay attack), or alter requests (i.e., tampering attack).

We assume that the collection of training data can take place in a secure environment, such as during device manufacturing in a factory, and that the training mode cannot be triggered after the training phase. Moreover, we assume that the device is not compromised by attackers, either locally or remotely. Adversaries who have compromised the device are beyond our scope, and we discuss them in § VIII.

IV. System Design

A. MCU-Token Overview

Figure 3 shows the overall architecture of MCU-Token. For a sensitive request, the client runtime on local devices can generate a hardware fingerprint based access token and send this token along with the requests. A client fingerprint generation module is integrated into the devices' firmware for generating an extra hardware fingerprint based access token that is sent along with the requests. A backend fingerprint verification module can be deployed on other devices or on the cloud for validating the token.

MCU-Token generates fingerprints based on the message digest of corresponding request. Since fingerprints derived from static features may still be stolen, we use non-repetitive fingerprints for different requests. Similar to the challenge response based approaches, our fingerprint is created by changing the fingerprint generation configurations (e.g., using different hardware features) to produce unique fingerprint values. To prevent attackers from impersonating the backend to send fake challenges and directly read out the devices' responses, we adopt a one-round protocol and autonomously generate the challenges from the local devices by adopting the client's message digest as the challenge code. In this way, our fingerprint is bound to specific requests and attackers can no longer reuse fingerprints to craft requests.

MCU-Token protects the fingerprints by randomly mixing poisoned results on the responses. Our one-round protocols can still be exploited by ML attacks as our message digesting rules are known to attackers. They can retrieve the original challenges and responses by eavesdropping on the communication and training a model to predict the responses. It is

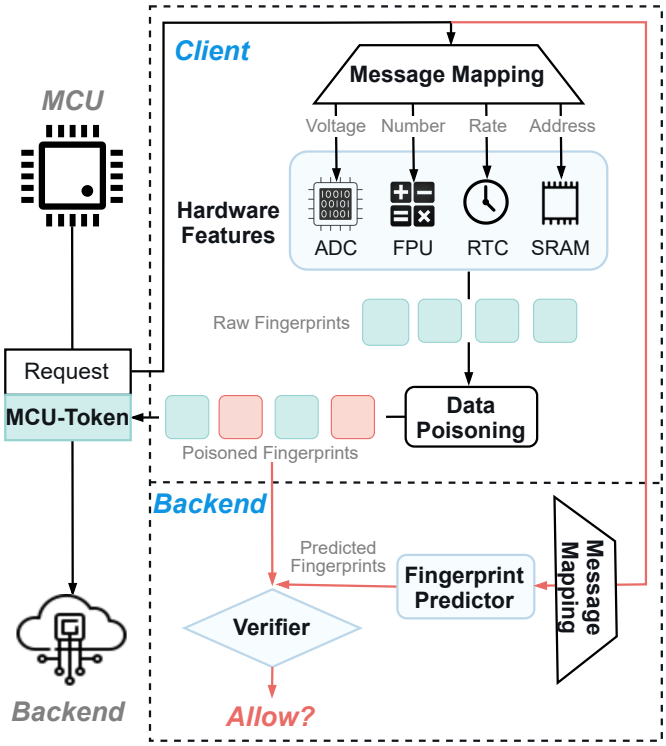


Fig. 3: The architecture of MCU-Token. A hardware fingerprint-based token (i.e., MCU-Token) is sent along with the request from devices. The token mixes multiple valid fingerprint values (green block) with poisoned results (red block), and the backend verifies the token by comparing the fingerprints with the predicted values.

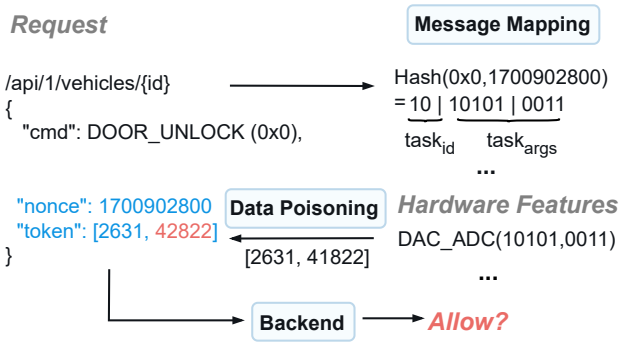


Fig. 4: A running example of the Telsa car key. The blue fields are the extra payload added by MCU-Token and the red text is the poisoning data.

difficult to make the hardware fingerprints unpredictable to the attackers as the responses of dedicated hardware features are simple to be fitted by machine learning. A more practical way is to prevent attackers from obtaining enough valid fingerprints to train their model. We choose to add poisoned data to the responses by changing some of the fingerprints to fake data, making attackers cannot distinguish the valid data and fake data. If they train their model with the poisoned data, their model may fail to precisely predict the responses, which can be easily identified by our backend verification module.

In contrast, MCU-Token is only trained with valid data in the device binding phase and does not update the model with poisoned data. As such, MCU-Token can resist machine learning attacks as our backend module is not affected by the poisoned data and can still authenticate devices based on the remaining unchanged fingerprint data.

Figure 4 shows a running example of using MCU-Token in the Telsa BLE car key. For sensitive commands, such as unlocking the car door, MCU-Token is triggered to add a token to the request. The token consists of two fingerprints generated by different hardware features. Both the client and the backend implement the same message mapping algorithms that use the digest bits of the raw command and a nonce to select hardware features as fingerprinting tasks and determine the corresponding task arguments. One of the fingerprints is intentionally altered with poisoned data. The backend authentication service independently maps the task arguments from the request payload and generates two fingerprint values using the predictor. Access is granted when the verifier determines a close match between one of the client's fingerprints and the predicted values.

B. Selecting Hardware Features

To ensure MCU-Token can generate unique fingerprints for different requests, it is crucial to identify an adequate number of commonly supported hardware features in major MCUs. However, previous studies [56] only explored limited hardware features. Therefore, we explore new hardware features by examining the datasheets of MCUs. We identify potential hardware features by looking for theoretical evidence [42] that the IC-level variation of a particular hardware feature can lead to performance or accuracy deviations. We then conduct experiments to validate the output variable ranges of these features across different settings or inputs, and evaluate their ability to reliably discriminate between identical devices. Table II, lists some of the hardware modules on STM32F4 serials with their functional descriptions. The features behind these modules may not have been explored or implemented on MCU devices, but their sources of uniqueness have already been revealed by existing work. Thus, we investigate the following 6 common hardware modules of COTS MCUs:

DAC/ADC. A digital-to-analog converter (DAC) can convert digital values to analog signals, such as voltage. Conversely, an analog-to-digital converter (ADC) performs the reverse function of converting analog signals to digital outputs. Previous studies [64], [56] have demonstrated that each ADC exhibits distinct biases when outputting digital values. By generating multiple analog signals through the DAC, we can induce variations in the ADC's output values and use these biases to uniquely identify devices.

Float Point Unit (FPU). Similar to GPU [37], the FPU is also dedicated to accelerating float number arithmetic. Their computing power for float point calculations can vary among devices of various models. By assessing their performance in executing diverse computing tasks, we can discern and differentiate between distinct devices.

Pulse Width Modulation (PWM). PWM regulates power levels by turning signals on and off at a constant frequency.

Table II: Hardware modules on STM32F4 serials with their functional descriptions and sources of uniqueness.

Hardware on MCUs*	Functionality	Source of Uniqueness
TIMx. RTC, WDG	Clock and timer	Skew[56], Phase[44]
SRAM, Flash, ROM	Storage medium	Special property[60]
ADC, DAC, PWM	Voltage processor	Numerical error[56]
FPU, CRC, CRYP	Computing units	Performance[37]
PWR, DMA, RCC	System controller	Manufacturing defects[32]
I2C, SPI, USART	Data transmitter	Transmission delay[22]

* All abbreviations refer to *RM0090 Reference Manual*.

By analyzing the accumulated power over specific time intervals at different frequencies, it is possible to differentiate between different MCUs based on the observed accumulation discrepancy.

Real Time Clock (RTC). RTC provides timers by maintaining an accurate time base via the crystal oscillator. As clocks usually have fixed drifts from ideal frequencies, we use this feature (i.e., **RTCFre**) to set timers with diverse frequencies to statistically record the accumulated time drift. The time phase [44] among multiple clocks can also be used as a feature (i.e., **RTCPha**). On MCU-based devices, the main clock is always a fast clock and peripheral clocks are always slow. For instance, the main clock's frequency is 180MHz and a crystal oscillator clock's is 32kHz on STM32F429. Thus, we can use the dual clocks of the system and the peripherals to get instantaneous phases and measure the phase features.

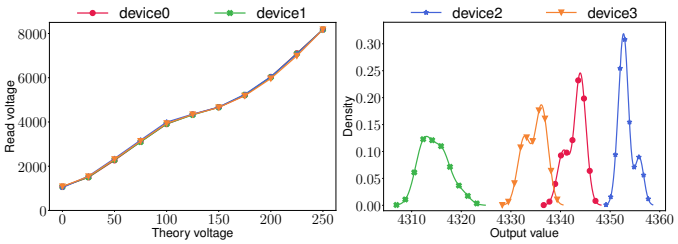
SRAM. Previous approaches [34], [42] indicate that the initial states of SRAM cells are usually stable and can be used as a kind of PUF. We collect the initial bit states within a specified SRAM address range during device boot-up and employ statistical features derived from these bits to differentiate various devices.

Flash. Previous study [24] uses NAND-flash's different sector read times as distinctive fingerprinting features because the NAND-Flash is located on a separate chip with dedicated drivers, which requires access via the I2C/SPI bus and can affect the access time of sectors. In contrast, most MCUs are only equipped with NOR-Flash, where different sectors exhibit similar read times due to their integration on the same chip as the MCU, enabling direct access. Thus, we exclude this feature as only a few devices have NAND-flash.

Rather than generating a static fingerprint, we manipulate the settings or inputs to generate varying fingerprints from these hardware features. Each individual hardware feature can serve as a fingerprinting task, producing multiple fingerprint results by utilizing different input arguments. For instance, we use DAC to generate a voltage and ADC to read it. Theoretically, the read voltage of ADC and the input voltage of DAC can be formulated as a proportional mapping,

$$V_{ADC} = V_{DAC} * \frac{2^{res_{ADC}} - 1}{2^{res_{DAC}} - 1} \quad (1)$$

res means the resolution. However, as shown in Figure 5a, this mapping is not exactly linear and its density distribution may vary by devices (see Figure 5b). By inputting different



(a) Mapping in DAC-ADC (b) Density of DAC-ADC

Fig. 5: Example of DAC-ADC fingerprints of four ESP32S2 devices. (a) shows the mapping between theory voltages of DAC and read voltages of ADC. (b) shows the density distributions of the read voltages whose theory voltage is 125.

Algorithm 1: Message Mapping Algorithm

Input: $request$
Output: $task$
1 $task \leftarrow [\ ],\ digest \leftarrow 0$
2 $operation \leftarrow$ `GetOperation`$(request)$
3 $nonce \leftarrow$ `GetNonce`$(request)$
4 $payload \leftarrow$ `GetPayload`$(request)$
5 $totalNum \leftarrow$ `GetTaskNumber()`
6 **for** $i \leftarrow range(0, totalNum)$ **do**
7 `// provide protection for the operation`
8 $h_1 \leftarrow$ `Hash`$(operation, nonce, digest)$
9 `// provide protection for the payloads`
10 $h_2 \leftarrow$ `Hash`$(nonce, payload_i)$
11 `// connect with another payload`
12 $h_3 \leftarrow$ `Hash`$(nonce, payload\_i)$
13 $digest \leftarrow$ `Hash`(h_1, h_2, h_3)
14 $arg_0, ..., arg_m \leftarrow$ `DivideArguments`$(digest)$
15 $task_i \leftarrow (arg_0, ..., arg_m)$
16 **end**
17 **return** $task$

voltages via DAC, we can distinguish different devices from the actual voltage read by ADC. Similar approaches can be applied to other hardware features, such as setting different RTC clock sources and reading different address ranges in SRAM. The detailed designs of the fingerprinting tasks for the 6 hardware features are discussed in Appendix A. In this way, we can gain enough $(arguments, fingerprint)$ pairs for different requests.

C. Binding Requests with Unique Hardware Fingerprints

We aim to protect the integrity of client requests, so as to prevent adversaries from gaining access to the backend via tampering with users' requests. Traditional techniques based on message authentication codes [4], [27] are not sufficient, since they require the client side to store secret keys which would be easily compromised on IoT devices as reported by [67], [6]. We propose to bind each request with unique hardware fingerprints without relying on the assumptions of key security on IoT devices. In this way, any deliberate manipulation targeting requests would be detected by checking the correctness of the fingerprints.

As described in § IV-B, we leverage multiple hardware modules to execute some hardware tasks which take as input the request-derived arguments and output unique hardware fingerprints. Here, the request should be mapped into unique task arguments to avoid collisions between fingerprints from different requests. Unfortunately, the requirement may be hard to satisfy, because the input space of one hardware task may be very small. As an example, the number of the arguments designed for ESP32S2 is about 20,000. For a hash function whose outputs are mapped into a space size (denoted as d) of 20,000, assuming there are 2^{10} distinct inputs, the probability of a collision in the output has already reached 99%. If each task's arguments are generated by the above hash function based on the specific content from the request, an attacker can compromise the integrity of the request with high probability by manipulating specific content and constructing collisions.

To address the issue, we devise a novel random message mapping algorithm that maps some content in the request to arguments of multiple hardware tasks via a hash function, rather than each content corresponding to only one task. This exponentially increases the output space of the single hash function, thus exponentially reduces the probability of collisions. The detailed process is shown in Algorithm 1. $request$ contains an operation, a nonce (e.g., a random number), and several payloads (line 2-4). For each $request$, we divide payloads into $totalNum$ groups and generate the i-th payload via $payload[i\ mod\ totalNum]$ sequentially. Note that $totalNum$ means the number of tasks used for an authentication which is a pre-defined fixed number (line 5). Then, we generate task arguments for the $totalNum$ hardware tasks according to the $request$ information (line 6-15). For each round, h_1 is the digest of the operation, the nonce, and the digest in the last round. h_2 is the digest of the i-th payload from the beginning of the payload group while h_3 is calculated by the i-th payload from the end of the payload group. Such a design correlates a payload content to arguments of two hardware tasks, decreasing the probability of output collisions. Finally, we concatenate $h1, h2, h3$ and use an extra hash calculation to get the $digest$ in this round. The specific segmentation of $digest$ constitutes the arguments which are further fed into the corresponding hardware tasks.

D. Countering Machine Learning Attacks

Once requests and fingerprints are transferred in the network, they may be abused by attackers. Essentially, they may learn the relationships between requests and fingerprints and then forge reasonable fingerprints to cheat the backend (i.e., software mimic attacks).

To resist those misbehaviors, one intuitive method is to make the hardware tasks as complex as possible to prevent the attackers from easily learning the relationships. Existing works on PUF [41] concentrate on constructing unpredictable relationships between the inputs and outputs of hardware modules. However, these approaches may depend on special circuits which are only available on some dedicated devices. In this work, instead of relying on specific complex hardware tasks, we tend to generate unlearnable fingerprints only based on some common hardware modules (as specified in § IV-B).

Inspired by the data poisoning attacks widely explored in the area of machine learning [55], we randomly add

some well-crafted noises to the raw hardware fingerprints to generate poisoned fingerprints. There are three requirements for the poisoned fingerprints: (1) Verifiability: the poisoned fingerprints can be successfully authenticated by the backend according to the raw ones. (2) Dissimilarity: the poisoned fingerprints should detach from the raw ones as much as possible to prevent attackers from learning the features of the raw fingerprints. (3) Unidentifiability: the noises in those fingerprints cannot be identified and removed by attackers through advanced techniques such as machine learning.

To satisfy the first requirement, we randomly retain a portion of the raw fingerprints as normal ones which will be used to pass the authentication on the backend side. For the remaining fingerprints, we make a trade-off between the dissimilarity and the unidentifiability when adding random noise. To increase the dissimilarity between the raw and the poisoned fingerprints, the noises added should be as large as possible yet would enable attackers to easily identify poisoned fingerprints. Here, we generate the well-crafted noises that are slightly larger than the inherent hardware errors. Specifically, for a pair of $(arguments, fingerprint)$, the poisoned fingerprint is computed as:

$$fp_{poisoned} = fp_{raw} * (noise + 1) + C \tag{2}$$

where C is a constant and $noise$ is randomly sampled from distributions (e.g., Laplace distribution).

E. Verifying Fingerprints at Backend

We describe how to authenticate a request along with its fingerprints on the backend side. According to the above construction method of poisoned fingerprints, a straightforward solution is to compare them with the raw fingerprints and check whether the number of matched elements is larger than a pre-defined threshold. Based on this idea, we propose a novel fingerprint predictor to mimic the behavior of each hardware module and to predict an approximation of the raw fingerprints generated by the clients. The predicted fingerprints generated by the predictor are finally fed into a verifier to compare with the poisoned fingerprints sent by the clients. Throughout the authentication process, the backend does not know if a fingerprint is poisoned. It just checks the number of fingerprints that match the raw ones to decide whether the authentication passes. The retained raw fingerprints (i.e., the normal ones) will match the predicted ones, ensuring that the authentication can succeed. The details are as follows.

The predictor consists of a set of sub-predictors, which are regression models for one task of one client. Essentially, before one client device is deployed (e.g. during device manufacturing in a factory), the backend collects enough $(arguments, fingerprint)$ pairs for each task and uses them to train a new sub-predictor. Meanwhile, the verifier also contains a set of sub-verifiers, each as a binary classifier for one sub-predictor. In the deployment phase, a sub-verifier is trained using fingerprints from the corresponding client (same with the sub-predictor's) as positive samples and those from other clients as negative samples. In the real-world environment, it takes as input one predicted fingerprint generated by the corresponding sub-predictor and that from a client and outputs whether the latter is correct.

When receiving a request from one client, the authentication process has the following steps. (1) The backend uses the message mapping algorithm to generate $totalNum$ tasks, similar to the relevant operations on the client side. The task output item is represented by a $(arguments, fingerprint)$ pair for convenient utilization in later operations. (2) The predictor launches appropriate sub-predictors for the above tasks and predicts the corresponding fingerprints based on the tasks and arguments. (3) The verifier uses relevant sub-verifiers to check whether the predicted fingerprints match that of the client for each task. The backend determines authentication as valid by checking if the number of matched fingerprints exceeds a pre-defined threshold (i.e., $acceptNum$). Specifically, the backend maintains a timestamp or sequence number of the requests to prevent replay attacks. To prevent wireless signal relay, the backend measures the message round trip time and compares it with the predicted times based on specific tasks. (4) The backend returns the authentication result to the client for further communications.

V. Security Analysis

A. Countering Fingerprint Mimic Attacks

In this subsection, we present how MCU-Token resists fingerprint mimic attacks, including hardware mimic attacks and software mimic attacks as introduced in § III.

1) ***Hardware Mimic Attack****:* To launch hardware mimic attacks, one adversary may purchase devices with MCU-Token installed and with the same types of hardware as the victim devices to generate fingerprints for any request. Regarding those attacks, we utilize hardware features to construct hardware fingerprints. These hardware features exhibit variations across different devices, even if they have the same model, making fingerprints generated for different devices distinct. Consequently, attackers' devices can be identified as unauthorized due to the distinctive fingerprint patterns derived from the specific hardware characteristics.

2) ***Software Mimic Attack****:* Attackers can launch software mimic attacks by eavesdropping on communication channels, monitoring requests with fingerprints, and learning their relationships. To defeat those misbehaviors, MCU-Token utilizes the data poisoning based method that adds random well-crafted noises to the raw hardware fingerprints. Note that the noises not only make adversaries fail to learn the correct relationships between the requests and the raw hardware fingerprints, but are also random and stealthy to avoid adversaries identifying and removing them. It is difficult for attackers to distinguish the poisoned data as the data is slightly modified from valid fingerprints. This discrepancy is adequate to prevent attackers from precisely predicting fingerprints. If attackers learn with poisoned fingerprints, the deviated data can significantly degrade the performance of their model. We present a quantitative analysis through a linear hardware task as an example, which can be represented by the function: When all the fingerprints are poisoned, the function parameters fitted by an attacker are,

$$\begin{aligned} w' &= (1 + noise) * w \\ b' &= (1 + noise) * b + C \end{aligned} \tag{3}$$

In Appendix B-A, we prove the above equation and show how MCU-Token rejects the poisoned fingerprints.

B. Countering Fingerprint Reuse Attacks

Besides fingerprint mimic attacks, we illustrate the security of MCU-Token against fingerprint reuse attacks, including replay attacks, forwarding attacks, tampering attacks, and relay attacks.

1) ***Replay Attack****:* In our message mapping approaches, any changes in the request payload result in different fingerprints. Therefore, MCU-Token can utilize the existing timestamp or sequence numbers of the protocols, or keep our nonce growing. The backend can record the last value of this increasing number and reject repeated requests.

2) ***Relay Attack****:* For wireless devices, attackers may relay the physical signals (e.g., BLE [10], RFID [31]) to valid devices at a distance. Similar to existing approaches [50], [54], MCU-Token can measure the request's round-trip time to identify the signal relay. For the requests delivered by networks (e.g., HTTP), attackers may forward authentication requests to valid devices. Our one-round protocol can ensure that requests are initiated by the clients, so attackers cannot simply offload the server's requests to trigger authentication on real devices. They can only try to reuse the fingerprints of existing requests, which is discussed in the subsequent tampering attacks.

3) ***Tampering Attack****:* If an attacker tampers with requests to launch tampering attacks, such as replacing operations in the request while retaining the fingerprints, MCU-Token can easily detect such misbehaviors with high probability. Specifically, the request is tightly bound with its fingerprints via the message mapping algorithm. Any unwanted modifications targeting request contents would result in significantly different hardware tasks being generated on the server and client sides using the same message mapping algorithm with a high probability. Compared to the naive approach of just hashing once, our algorithm can exponentially increase the attackers' attempt times. In Appendix B-B, based on a hash collision problem, we analyze Algorithm 1 step by step to show how MCU-Token creates obstacles to the tampering attack.

VI. Evaluation

To evaluate MCU-Token's authentication performance and security, we aim to answer the following four questions:

- **Q1** Which hardware features can be used for fingerprinting?
- **Q2** How accurate is MCU-Token's authentication under different client and backend settings?
- **Q3** Can MCU-Token defend against various fingerprint mimic attacks and reuse attacks?
- **Q4** How much overhead MCU-Token bring to real scenarios?

To answer **Q1**, we evaluate the performance of every single fingerprint and their combinations for device authentication and their stability under different environments (§ VI-B). For **Q2**, we show the true positives and false positives of MCU-Token in authentication with different parameter settings, especially poisoning-related configurations (§ VI-C). For **Q3**, we launch hardware mimic attacks, software mimic attacks, and tampering attacks to evaluate the security of MCU-Token (§ VI-D). At last, we conduct case studies on various usage scenarios to demonstrate the usability of MCU-Token for **Q4** (§ VI-E).

Table III: Devices used in evaluation.

Model-brand	Microcontroller	Frequency	# of devices
ESP32S2	Xtensa LX7	240MHz	30
STM32F103	Cortex M4	72MHz	20
STM32F429	Cortex M4	180MHz	10

A. Experiment Setup

1) ***Client-side Implementation****:* We implement MCU-Token on 30 ESP32S2, 20 STM32F103, and 10 STM32F429 MCU-based devices, as shown in Table III. We deploy the 6 hardware features in IV-B on these devices. Details of the designed tasks are shown in Appendix A.

2) ***Backend-side Implementation****:* We implement the backend authentication service using Python and deploy it on a Windows 10 PC with 16 GB RAM and 2.8 GHz CPU. The communication between the backend and the client devices are developed through serial ports. Our predictors are regression models, specifically ExtraTrees, and our verifiers are classification models, specifically RandomForest, all implemented using Scikit-learn [47]. The hash function used in Algorithm 1 is APHash[2]https://github.com/ArashPartow/hash. For data poisoning, $noise$ is obtained from the uniform distribution of [0.08,0.2], and we empirically set C (in Equation 2) to 1 (discussed later).

The regression models are trained with $(arguments, fingerprint)$ pairs in the model training phase. When training classification models, we randomly sample 10 other devices as negative examples. We train a regression model and a classification model for each hardware task of each client. For each device and hardware feature, we gather 5,000 pairs of data. We use half of them to train our models and the other half to test.

B. Usability of Hardware Feature for Fingerprinting

1) ***Identifiability of Hardware Fingerprints for Device Authentication****:* We evaluate whether a hardware feature can be used to identify one device in the subsection. For each hardware task on each device, we generate a fingerprint (without poisoning) and check two properties: (1) if it can be successfully verified at the backend side; (2) if it will be misidentified as other devices. In the evaluation, we use all the devices described above. For each device type, we separately employ each device to impersonate all other devices to figure out whether it will be misidentified as other devices. Especially, we build two metrics: (1) true positive rate (TPR) equal to the proportion of devices that pass the authentication process successfully to the total devices, (2) false positive rate (FPR) equal to the proportion of misidentified devices to the total devices.

Table IV shows the TPRs and FPRs of different types of features of different devices. We can see that RTCFre and SRAM achieve a TPR of more than 90% and an FPR of less than 8% for various devices, meaning that these two hardware features can identify one device with a high probability. By

[2]https://github.com/ArashPartow/hash

Table IV: TPRs and FPRs of various hardware features.

	ESP32S2		STM32F429		STM32F103	
	TPR	FPR	TPR	FPR	TPR	FPR
DAC_ADC	83.74	8.58	82.73	16.83	96.25	37.90
FPU	76.59	38.90	83.50	29.94	76.65	36.63
PWM	84.83	17.54	84.90	37.67	80.00	35.57
RTCFre	91.76	1.96	89.88	7.49	99.19	1.96
RTCPha	77.04	58.38	73.88	58.10	74.56	36.88
SRAM	94.27	0.01	98.69	0.05	96.89	0.03
Ensemble	96.63	9.44	97.06	14.10	97.94	14.31
Ensemble*	98.47	1.06	97.67	6.89	98.68	1.64

* The results of excluding useless features, i.e., FPU and RTCPhra for ESP32S2, PWM and RTCPhra for STM32F249, DAC/ADC, FPU and PWM for STM32F103.

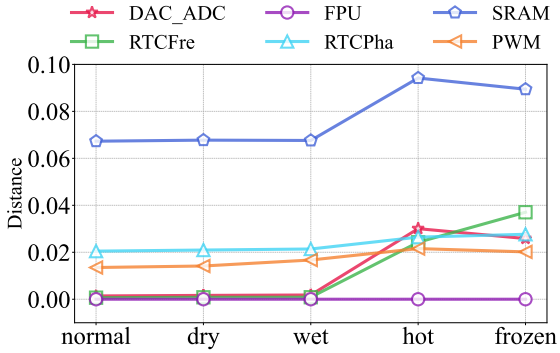


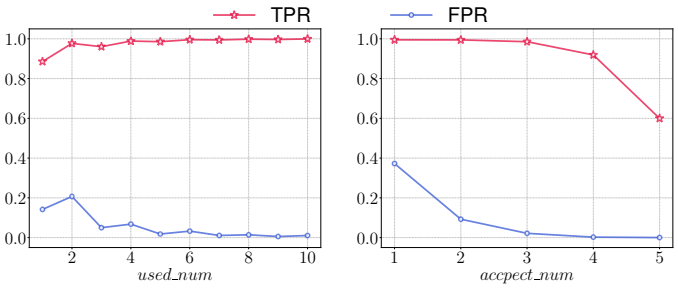
Fig. 6: Features fluctuations in different environments.

contrast, there are hardware features with high FPRs, such as the FPU on ESP32 and STM32F103. FPUs are unavailable on ESP32S2 and STM32F103. We use software-based floating point calculators, which results in a high false positive.

Besides evaluating each individual hardware features, we utilize multiple hardware features to achieve more accurate authentication. We eliminate useless fingerprints for each device category, i.e., FPU and RTCPhra for ESP32S2, PWM and RTCPhra for STM32F249, DAC/ADC, FPU and PWM for STM32F103. As shown in Table IV, our approach achieves a FPR of 1.06% while maintaining a TPR of 98% (on ESP32S2).

*2) **Stability of Hardware Fingerprints under Various Real-world Environments**:* We evaluate the stability of hardware features in different environments with varied temperatures and humidity. The environmental parameters in normal conditions are 28°C and 61% relative humidity (RH). Besides, we set up two humidity environments: 37% RH (called dry) and 98% RH (called wet), and two temperature environments: -23°C (called $frozen$) and 52°C (called hot). We collect $(arguments, fingerprint)$ pairs under different environmental conditions. Then, we calculate the distances between these pairs and the pairs collected under normal conditions. The distances are calculated as the relative error between $fingerprint$s under the same $arguments$. The average distance fluctuations during different environments are shown in Figure 6.

We find that fluctuations of different hardware features are different. For all features, the degree of change in distances is less than 0.1. Considering the influence of the two factors, humidity has almost no influence. It is evident that temperature has a significant impact on hardware fingerprints, in particular on RTCFre and DAC/ADC. However, it is important to note that these temperature settings are rarely encountered in real-life scenarios. Even if they do exist, we can collect fingerprint information from these environments to train the backend's predictors and validators to avoid false positives/negatives.



(a) Different $usedNum$ (b) Different $acceptNum$

Fig. 7: TPRs and FPRs of MCU-Token under different parameter settings.

C. Authentication Accuracy of MCU-Token

We assess the authentication accuracy of MCU-Token under different parameter settings. There are three parameters in the work: (1) $totalNum$ denotes the number of hardware tasks executed by clients; (2) $usedNum$ represents the number of fingerprints without being poisoned by clients; (3) $acceptNum$ depicts the threshold of verified fingerprints required for successful device authentication at the backend side. In the evaluation, we select DAC/ADC, PWM, RTCFre, and SRAM in MCU-Token, according to the evaluation results in § VI-B. By default, we set $totalNum$ to 10, $usedNum$ to 5, $acceptNum$ to 3. The following evaluations are all based on ESP32S2 devices.

We set $totalNum = 10$ and change other two parameters. Figure 7a shows the results under varied $usedNum$ and $acceptNum = \lceil \frac{usedNum}{2} \rceil$. When $usedNum$ is 1, MCU-Token only uses one type of hardware feature as the fingerprint, the TPR or FPR is equal to the average TPR or FPR value of all individual features in Table IV. As $usedNum$ increases, TPR increases and FPR decreases because the larger number of used fingerprints provides more information of device identities. In Figure 7b, we change $acceptNum$ with $usedNum$ as 5. As $acceptNum$ increases, more fingerprints need to be verified by the verifier, leading to the reduction in both TPR and FPR. Actually, we can set the ratio of $acceptNum$ to $usedNum$ to balance the TPRs and FPRs.

To confirm whether the normal fingerprints can pass authentication and the poisoned ones cannot, we conduct experiments by introducing various levels of $noise$ to modify the raw fingerprints. As shown in Figure 8, low TPRs indicate that the poisoned fingerprints are unlikely to pass authentication. When $noise$ exceeds 0.08 (the $noise$ used is from [0.08, 0.2]), the TPR drops to less than 2%. The results show that successful authentication only relies on normal fingerprints and is not affected by poisoned ones.

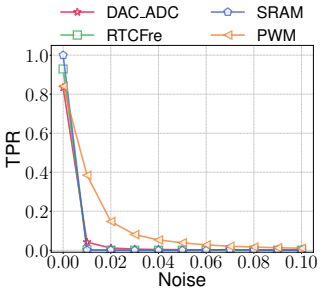
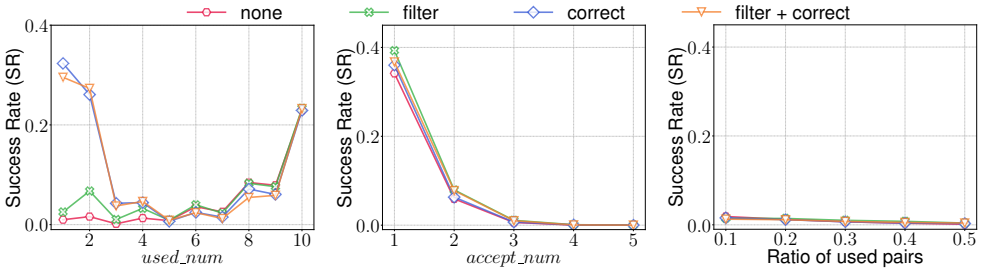


Fig. 8: Authenticate with poisoned fingerprints. TPR shows whether authentications pass.



(a) Different $usedNum$ (b) Different $acceptNum$ (c) Different used ratio

Fig. 9: Software mimic attack results on MCU-Token. "filter" indicates that the attacker filters out poisoned pairs for training and "correct" indicates the attacker corrects outputs.

Table V: Hardware mimic attack success rates.

	ESP32S2	STM32F103	STM32F429
ESP32S2	0.0188	0.0000	0.0000
STM32F103	0.0001	0.0606	0.0078
STM32F429	0.0000	0.0000	0.1058

D. Security of MCU-Token against Various Attacks

We launch various adaptive attacks including hardware mimic attacks, software mimic attacks, and tampering attacks to evaluate the security of MCU-Token.

1) ***Hardware Mimic Attack****:* During the experiment, we simulate hardware mimic attacks by having an attacker use a device that has the same or similar brand and model as the victim's device. The devices are randomly divided into two groups: legitimate devices and attacking devices. We then initiate the authentication process using the attacking devices and assess whether they are correctly identified as illegal devices. Finally, we measure the success rate of impersonation attacks using various types of devices. The success rate is shown in Table V.

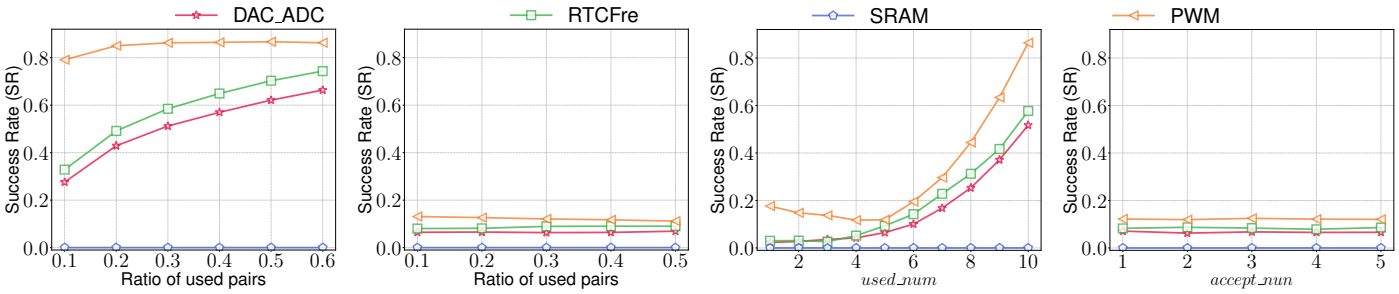
In Table V, the rows represent the type of devices that are known to the backend (called target devices) while the columns represent the types of devices used by the attacker (called source devices). When the source device has the same brand and model as the target device, the attacker can successfully launch an attack. However, the success rates are still low, less than 11%. For the attacks using different device models, the success rates are even lower, with less than 0.01% success rate. These results indicate that hardware mimic attacks are ineffective against MCU-Token.

2) ***Software Mimic Attack****:* We evaluate the machine learning based software mimic attacks and consider the that attackers can collect $(arguments, fingerprint)$ pairs (which are partially poisoned) and train a regression model to learn the relationship between $arguments$ and $fingerprint$. First, we evaluate the effectiveness of MCU-Token in defending against software mimic attacks. Then, we show the effectiveness of MCU-Token by analyzing attacks on single features. Furthermore, we consider an attacker who attempts to filter out poisoned pairs to demonstrate that poisoned pairs are unable to be identified.

Defending Effectiveness Against Software Mimic Attacks. We evaluate the effectiveness of MCU-Token in defending against machine learning attacks with different attack settings. The backend authentication service's settings are the same as § VI-C. We consider an attacker who trains a regression model for each hardware feature and generates fake fingerprints based on the requests to cheat the backend authentication. The models used by attackers are the same as those used by the backend. Furthermore, attackers can employ various training and predicting strategies to carry out their attacks. For training, the attacker chooses to filter $(arguments, fingerprint)$ pairs, (1) the attacker uses all the pairs directly; (2) the attacker randomly selects $usedNum$ pairs as normal pairs. At the same time, we consider that the attacker may try to correct the output fingerprints, (1) the attacker predicts fingerprints directly; (2) the attacker predicts fingerprints directly with probability $usedNum/totalNum$. Otherwise, the attacker selects a value from the $noise$ range and corrects fingerprints through the reverse process of poisoning. The attacks with corrected results can utilize the poisoned pairs to improve attack performances. Combining the choices of filtering and correcting, there are 4 different attack strategies. We evaluate success rates for different attack strategies under various parameter settings. The results are shown in Figure 9.

The parameter settings are the same as those used for device verification. The used ratio of every single feature is 30%, which means during the attacks the attacker obtains 30% of all normal pairs (non-poisoned pairs) for training. The value of $usedNum$ determines the ratio of the normal pairs. In Figure 9a we vary $usedNum$ from 1 to 10. When $usedNum$ is 1 or 2, the majority of pairs obtained by the attacker are poisoned and the authentication success threshold (i.e., $acceptNum$) is set to a very low value (i.e., 1). Therefore, the strategies involving corrected output achieve a high success rate of 32.3%. When $usedNum$ is close to $totalNum$, the attacker obtains more normal pairs and trains his models more effectively. When the number of normal pairs and poisoned pairs is equal, the entropy is at its highest, resulting in an attack success rate of around 1% regardless of the strategies. Figure 9b shows that as $acceptNum$ grows, the difficulty of passing the authentication also increases.

In Figure 9c, we change the used ratio of normal pairs. The used ratio refers to the number of normal pairs obtained by the attacker. We find that when the attacker gets more normal pairs,



(a) Authenticating without protection (b) Authenticating with protection (c) Different $usedNum$ (d) Different $acceptNum$

Fig. 10: Software mimic attack success rates on single fingerprints. (a) and (b) are the attack performances without and with data poisoning, respectively. (c) and (d) demonstrate the influence of different parameter settings.

the success rate (SR) decreases but does not increase. The reason is that with more normal pairs there are more poisoned pairs (with the default setting, the number of normal pairs and poisoned pairs are the same). The models trained by the attacker are affected more, resulting in the generation of invalid fingerprints for the corresponding arguments. This indicates that data poisoning is effective in preventing software mimic attacks.

Software Mimic Attacks on Single Features. We further analyze the mimic attacks on single features. In this experiment, we use the same attack settings as in the previous experiment. The training process for the attackers is the same as before. For testing, we use only one fingerprint for authentication and check if the backend is fooled. We use the highest success rate of the four attack strategies as the final result.

Figure 10a and Figure 10b show how MCU-Token provides protection to a single feature. In Figure 10a, we set $usedNum = totalNum$, and the pairs obtained by the attacker are all normal ones. In Figure 10b, we set $usedNum : totalNum = 1 : 2$, and half of the pairs are poisoned. It is important to note that, in these two different settings, the number of obtained normal pairs is the same, but in the latter one there are extra poisoned pairs. Without protection, the success rate mainly depends on complexity of the features. For SRAM, the power-on voltages of SRAM cells are unpredictable so the success rate is very low (almost 0%) no matter how many pairs are known. But for other single features, the attacker achieves more than 50% success rate with 0.3 of all the normal pairs, particularly in the feature PWM. When protected by MCU-Token, the success rate on PWM decreases to approximately 13% and for other fingerprints, the success rate is lower than 10%. The magnitude of the decline is remarkable. More importantly, as the obtained ratio increases, the success rate decreases. The results prove that the presence of poisoned pairs helps protect single features.

As for the parameters of MCU-Token, $usedNum$ affects the ratio of normal pairs. The attack success rate for a single feature will initially decrease and then increase as $usedNum$ increases. $acceptNum$ only works when authenticating with multiple features and has no influence on a single feature. The results are shown in Figure 10c and Figure 10d.

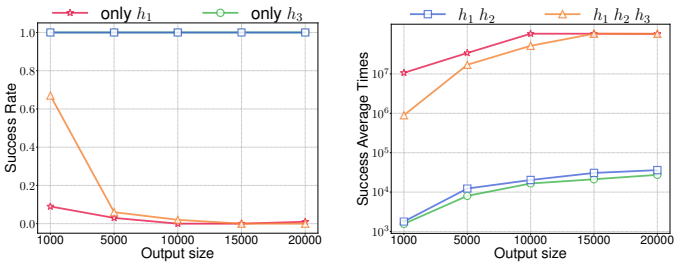
Poisoned Fingerprint Identification. We conduct an additional experiment to illustrate that an attacker is unable to identify the poisoned fingerprints. We consider three different attack methods: (1) Unsupervised learning: the attacker uses clustering algorithms to divide the $(arguments, fingerprint)$ pairs into 2 clusters. (2) Supervised learning: the attacker randomly selects a portion of the collected $(arguments, fingerprint)$ pairs as normal ones to train models. To identify a valid pair, the attacker predicts a fingerprint and calculates the related error with the true fingerprint. If the error is greater than a threshold (e.g., $noise$), the attacker regards the pair poisoned. (3) Incremental learning: this method is almost the same as the supervised learning based method. The difference between them lies in the way of training models. For initialization, the attacker randomly selects a small number of collected $(arguments, fingerprint)$ pairs to train the models, then uses the models to identify the subsequent unknown pairs. If a pair is classified as a normal (non-poisoned) one, the attacker can renew the models with this new pair. We assume the attacker retrains the models with a fixed number of pairs, i.e., the training step.

Table VI: Identification accuracy of poisoned fingerprints.

	DAC/ADC	RTCFre	SRAM	PWM
Unsupervised learning	0.5201	0.5042	0.4993	0.5354
Supervised learning	0.5142	0.5220	0.5409	0.5293
Incremental learning	0.5120	0.5005	0.5032	0.4889
Extra-device	0.9682	0.5745	0.4959	0.8991

We test various clustering algorithms, ratios of training data and training steps for each scheme. Also, we test different features individually. The maximum identification accuracy for each scheme is shown in Table VI. The identification of poisoned pairs is a 2-class classification task. The highest accuracy among different schemes is around 54%, only 4% higher than 50%, which indicates that software-based approaches fail to identify poisoned data.

Furthermore, we test a mixed scheme that combines software with hardware, called extra-device. The attacker replaces models with hardware to give fingerprints. In DAC/ADC and PWM, this scheme gets greater than 90% accuracy, but in the other two fingerprints, the accuracy is still low. Accuracy is related to the discrepancy between two devices and the value of added noises. For instance, for a $arguments$, fp_0, fp_1 are



(a) Tampering attack success rate (b) Attack success average number

Fig. 11: Tampering attacks on MCU-Token.

fingerprints from two devices. If $|\frac{(fp_0-fp_1)}{fp_1}| > noise$, the poisoned pair may not be identified. This guides us to a better way to launch poisoning, i.e., keeping $noise$ in the range of discrepancies among different devices.

Other Parameter Settings. MCU-Token prevents software mimic attacks via data poisoning and the poisoned pairs cannot be identified. We do not experiment with other parameter settings such as C in Equation 2, as they are not key parameters and have little effect on protection effectiveness. As long as the poisoned pairs can affect the training phases of the attackers, MCU-Token works well. The key point is the ratio of the normal pair the attackers can get and how they use it. These settings have been shown in the experiments above.

*3) **Tampering Attack**:* In MCU-Token, an attacker may tamper with the operation or payloads of requests. We assume that the attacker knows the message mapping algorithm installed on the client side and the attacker tries to modify the requests and keep the tasks the same (and the tokens will be the same). To simplify, we set parameters as below. $totalNum$ is 2, the number of operation types is 200 (the car key BLE operation types in Telsa are around 40), the size of payloads is 32 bit and the size of the nonce is 16 bit. We test the attack success rates with various numbers of arguments (i.e., the output size of the message mapping algorithm). The results are shown in Figure 11.

The line whose label is "only h_1" means that in Algorithm 1 we only use h_1 to generate arguments (the same for others). Figure 11a shows the success rate. Figure 11b shows the average times of modifying the request to launch a successful attack. Results prove that our algorithm is immune to tampering attacks. With a 10,000 output space size, the attack success rate is less than 1% and the number of arguments used in ESP32S2 is about 20,000 (i.e, the output space size is 20,000). What's more, comparing the results between "only h_1" and "only h_3", we observe that it is more difficult to modify the operation than the payloads. "h_1 h_2" shows that h_2 raises SR as attackers can modify payloads to keep the digest the same. With h_3 the success rate reduces greatly and the success average times are almost the squared values of those without h_3.

E. Case Studies on Various Usage Scenarios

To show MCU-Token's usability on different IoT devices, we choose some typical scenarios to perform case studies to evaluate the energy consumption of reasonable tasks number.

Table VII: Power and time for fingerprint generation.

	Encrypt	Voltage	FPU	Clock	Storage
ESP32S2	0.23W 2ms	0.22W 23ms	0.22W 97ms	0.19W 10ms	0.17W 10ms
STM32F429	0.74W 2ms	0.79W 39ms	0.76W 8ms	0.79W 47ms	0.71W 1ms
STM32F103	0.15W 5ms	0.16W 114ms	0.16W 17ms	0.15W 8ms	0.15W 1ms

*1) **Smart Home**:* Smart home devices adopt trigger-action platforms [30] to execute automation rules, which usually adopt token-based authentication and may be abused to maliciously trigger rules [30], [29]. We use the STM32F429 device as an IoT temperature sensor which can report the current temperature to trigger a rule of "if the temperature is higher than 32°C, open the window". After adopting MCU-Token, the trigger action platform can check the extra hardware access token to verify if the temperature data is actually from the sensor rather than attackers' phantom device [67]. Since the temperature data may be uploaded very frequently, we only use 4 fingerprints in the token to find a tradeoff between security and energy consumption.

*2) **PKE/BLE Key Fob**:* Existing PKE key fob uses rolling codes [9] for authentication and the risk is that attackers can record some codes to perform cryptographic attacks [36] to reveal the generating of rolling code or reuse the rolling code. As shown in Figure 4, we generate the MCU-Token's access token based on the command and use the rolling code as the nonce. We prototype the PKE rolling code mechanism on the ESP32S2 device and use two fingerprints for each request which only increases 32 bits to the existing payload. For the BLE key fob using RSA, we can use more fingerprints (e.g., 8) as BLE can send longer payloads. By verifying the extra access token, we can prevent cryptographic attacks [36] and relay attacks on these devices.

*3) **Hardware Security Token (HST) for FIDO-U2F**:* MCU-Token can be easily integrated into the existing FIDO-U2F [12] service for verifying if the FIDO-U2F HSTs are trusted devices. We use the STM32F103 devices as a HST which implements the FIDO-U2F client and deploys the FIDO-U2F server on the PC. FIDO-U2F's existing counter can be used as our nonce for message mapping and generate a hardware fingerprint based token based on the response payload. Since the HSTs have a high security requirement and are less sensitive to performance, we can generate 8 fingerprints and half of them are poisoned data. This token can be added as extra information in the attestation certificate and the server can verify this item to check the authenticity of HSTs. As a result, attackers attempting to clone the HST [7] still cannot impersonate the real device, even if they have stolen the private keys.

Table VII shows the energy consumption when authentication with MCU-Token, which varies on different fingerprinting tasks. Compared to the baseline of default token-based authentication using AES encryption, our fingerprinting generation incur an extra energy consumption of less than 4% on average. The extra time consumption is less than 31ms (2 fingerprints) and 115ms (8 fingerprints) on average.

VII. Related Work

Hardware Fingerprints are widely explored on various platforms such as mobile, PC, and IoT devices, to distinguish and track devices [37], [52], [65] or to authenticate devices [36], [26], [56], [24], [21], [22]. Unfortunately, most of these fingerprinting features require special hardware support, such as GPU [37], [52], mobile sensors [65], [32], and NADA flash [60], [24], which are absent in MCU-based embedded devices. For the approaches target IoT devices, HODOR [36] and [26] employ RFID signal features to fingerprint devices which is not a general solution for other kinds of IoT devices. DeMiCPU [21] and [22] do not consider the attackers can MitM mimic or forward the fingerprint. Our approach (i.e., MCU-Token) aims at proposing a general fingerprint framework for all kinds of COTS MCUs that can resist MitM advisories. IoT-ID [56] is the closest work but they use invariant fingerprints as the device identifier, which is vulnerable to both software mimic (ML attacks) and MitM interference. Our work first extends IoT-ID's solution to generate variable fingerprints based on different inputs and then proposes an arguments mapping protocol to bind the inputs with specific commands to ensure the integrity of messages.

Hardware-backed Authentication. Various PUF mechanisms [41] are proposed to enforce IoT authentication by dynamically reproducing cryptographic keys from devices rather than storing the keys on the firmware which can reduce the risks of key stolen. Priyanka et al. [41] investigate the performance and security of different PUF mechanisms and find that these approaches do not take both MitM attacks and software/physical impersonation attacks into consideration at the same time. For instance, the existing approach proposes Challenge-Response (CRP) based PUF protocols [45], [53], [15] to prevent replay attacks and software impersonation attacks. are proposed to solve this problem. However, they cannot protect the integrity of the commands and thus are vulnerable to several MitM attacks [40], [41]. Most of these PUF approaches require extra hardware supports (e.g., special circuits) which are not supported by our target devices, i.e., COTS MCU. MCU-Token aims to provide a general fingerprint framework that can be easily extended by adding new PUF-based fingerprint features (e.g., SRAM, Flash) that do not require extra hardware support. Moreover, we ensure the security of fingerprints by proposing argument mapping and data poison approaches to defend against MitM attacks and impersonation attacks.

Embedded Device Authentication Security. Embedded device authentication has been long regarded as vulnerable [67], [25]. Mirai [16] exploits weak passwords to compromise millions of devices. BIAS [18] and KNOB [17] can perform MitM attacks on almost all Bluetooth devices by impersonating validate devices. Existing USB hardware tokens [48], [13] are also proved to be insecure and can be cloned or impersonated during manufacturing or shipments. To protect these devices, various authentication or pairing approaches are proposed. T2Pair [38], [49], and [59] utilize the sensing operations (e.g., knob, button, or touch screen) to secure the pair of IoT devices. Their limitation is requiring Human-in-the-loop to generate sensor data. Using hardware features (e.g., fingerprint [21], [24], PUF [28], [43]) to secure the authentication is widely discussed in various platforms. However, IoT-ID [56] is the only work that focuses on MCU-based devices. We address several applicable issues of IoT-ID by proposing a new hardware fingerprint authentication framework MCU-Token, which can work on all kinds of MCU devices and can resist traditional token compromise and MitM attacks.

VIII. Discussion

Attackers Compromise the Devices. MCU-Token aims to mitigate device impersonation attacks due to credential theft, weak cryptographic support, or insecure authentication implementation. For attackers who have compromised the system locally or remotely, they may manipulate this device to send requests with valid authentication data (e.g., MCU-Token's hardware fingerprints), which is outside the scope of authentication security and not the design goal of MCU-Token. If they want to clone [7], [1] this device for off-path exploitation (e.g., via Phantom Client [67]), they still need to collect enough fingerprint data to mimic the real hardware features. However, the specific fingerprinting parameters used for data collection are unknown to attackers. Therefore, attackers need to explore a large number of fingerprinting parameters to obtain the set of training data, which is time-consuming and can be easily detected by the backends or the device owner.

The Maximum Limit of Requests Supported. In MCU-Token, the maximum number of requests that can be issued is determined by the range of hardware task arguments. In fact, the argument range for hardware tasks is typically large enough to accommodate the number of device requests in practical scenarios. For example, in a PKE/BLE key fob scenario, there are 20,000 different argument values on the ESP32S2 device. Assuming a person sends five distinct requests per day (e.g. unlocking the door) and 4 fingerprints are used per authentication, MCU-Token can provide protection for approximately 1,000 days. In addition, we can extend the current lifetime of MCU-Token by extracting new fingerprints from existing hardware features and retraining the backend. For instance, changing the SRAM address ranges can generate different fingerprints. All fingerprinting tasks and arguments can be changed to get more fingerprints for new requests.

Device Aging. In practice, hardware fingerprints may change due to device damage, aging, or other factors, resulting in failed server authentication at the backend side. Regarding this issue, customers can securely return their devices to the backend for re-collection of fingerprint data. Thus, these devices can be successfully authenticated when they are re-deployed in the customers' environments.

MCU-Token with Fewer Hardware Features. Hardware features used in MCU-Token may not be available on all devices, such as the DAC. But MCU-Token can still work, because fewer hardware features do not mean fewer fingerprints. On the one hand, SRAM and RTCs are available on almost all types of MCUs. Also, SRAM and RTCs can provide sufficient fingerprints. On the other hand, we can increase the number of fingerprints by modifying the fingerprinting tasks to generate as many fingerprints as possible.

Future Works. MCU-Token provides a general hardware fingerprinting scheme. It is promising to extend MCU-Token by supporting more hardware features and PUFs [42].

IX. Conclusion

We introduce MCU-Token, a hardware fingerprint based authentication mechanism to enhance the security of existing token-based authentication approaches for MCU-based IoT devices. MCU-Token can protect device authentication when traditional cryptography-based approaches (e.g., message encryption and signature) are compromised by attackers. With its simplicity, MCU-Token can be applied in diverse scenarios to authenticate different MCU-based IoT devices with high accuracy and can resist common attacks. The MCU-Token solution can be easily integrated into all kinds of existing IoT devices as its client runtime supports major COTS MCUs and its backend authentication service can be deployed on common IoT devices or on the cloud.

Acknowledgment

We would like to thank our shepherd and the anonymous reviewers for their comments. This work is supported in part by the Natural Science Foundation of China under Grant U20B2049, U21B2018, 62302452, and 62132011; Zhejiang Provincial Natural Science Foundation of China under Grant LQ23F020019. Kun Sun's work is supported in part by National Centers of Academic Excellence in Cybersecurity under Grant #H98230-22-1-0311.

References

[1] A product for clone PKE fob. https://clonemykey.com/.

[2] Android safetynet attestation. https://developer.android.com/training/safetynet/attestation.

[3] Hackers can clone tesla key fobs in seconds. https://www.esat.kuleuven.be/cosic/news/fast-furious-and-insecure-passive-keyless-entry-and-start-in-modern-supercars/.

[4] IoT Authentication. https://www.nabto.com/iot-device-authentication-comparison-guide/.

[5] JSON Web Tokens. https://jwt.io/.

[6] Millions of IoT Devices Using Same Hard-Coded CRYPTO Keys. https://thehackernews.com/2015/11/iot-device-crypto-keys.html.

[7] New Attack Could Let Hackers Clone Your Google Titan 2FA Security Keys. https://thehackernews.com/2021/01/new-attack-could-let-hackers-clone-your.html.

[8] NXP adds PUF to its next generation SmartMX2 microcontroller. https://www.intrinsic-id.com/nxp-adds-puf-anti-cloning-technology-next-generation-smartmx2-microcontroller/.

[9] Rolling Codes and Encryption. https://electronics.howstuffworks.com/gadgets/automotive/unlock-car-door-remote1.htm.

[10] Tesla cars and smart home locks vulnerable to bluetooth low energy relay attacks. https://www.spiceworks.com/it-security/threat-reports/news/bluetooth-low-energy-relay-attack/.

[11] Use a U2F Key to Secure Your Crypto Accounts. https://news.bitcoin.com/how-to-use-u2f-key-crypto/.

[12] Webinar: Securing iot with fido authentication. https://fidoalliance.org/securing-iot-with-fido-authentication/.

[13] The impostor among US(B): Off-Path injection attacks on USB communications. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.

[14] Milad Taleby Ahvanooey, Mark Xuefang Zhu, Qianmu Li, Wojciech Mazurczyk, Kim-Kwang Raymond Choo, Birij B. Gupta, and Mauro Conti. Modern authentication schemes in smartphones and iot devices: An empirical survey. *IEEE Internet of Things Journal*, 9(10):7639–7663, 2022.

[15] Muhammad Naveed Aman, Kee Chaing Chua, and Biplab Sikdar. Mutual authentication in iot systems using physical unclonable functions. *IEEE Internet of Things Journal*, 4(5):1327–1340, 2017.

[16] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[17] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The knob is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, August 2019.

[18] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias: Bluetooth impersonation attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2020.

[19] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. Breakmi: Reversing, exploiting and fixing xiaomi fitness tracking ecosystem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 330–366, 2022.

[20] Baibhab Chatterjee, Debayan Das, and Shreyas Sen. Rf-puf: Iot security enhancement through authentication of wireless nodes using in-situ machine learning. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 205–208, 2018.

[21] Yushi Cheng, Xiaoyu Ji, Juchuan Zhang, Wenyuan Xu, and Yi-Chao Chen. DeMiCPU: Device fingerprinting with magnetic signals radiated by cpu. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1149–1170, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[23] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[24] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. Timeprint: Authenticating usb flash drives with novel timing fingerprints. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1002–1017, 2022.

[25] Adrian Dabrowski, Katharina Pfeffer, Markus Reichel, Alexandra Mai, Edgar R. Weippl, and Michael Franz. Better keep cash in your boots - hardware wallets are the new single point of failure. In *Proceedings of the 2021 ACM CCS Workshop on Decentralized Finance and Security*, DeFi '21, page 1–8, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Boris Danev, Thomas S Heydt-Benjamin, and Srdjan Capkun. Physical-layer identification of rfid devices. In *USENIX security symposium*, pages 199–214, 2009.

[27] Daniel Díaz-Sánchez, Andrés Marín-Lopez, Florina Almenárez Mendoza, Patricia Arias Cabarcos, and R. Simon Sherratt. Tls/pki challenges and certificate pinning techniques for iot and m2m secure communications. *IEEE Communications Surveys Tutorials*, 21(4):3502–3531, 2019.

[28] Moneer Fakroon, Fayez Gebali, and Mohammad Mamun. Multifactor authentication scheme using physically unclonable functions. *Internet of Things*, 13:100343, 2021.

[29] Jingwen Fan, Yi He, Bo Tang, Qi Li, and Ravi Sandhu. Ruledger: Ensuring execution integrity in trigger-action iot platforms. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[30] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *22nd Network and Distributed Security Symposium (NDSS 2018)*, February 2018.

[31] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2011.

[32] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating physical-layer ble location tracking attacks on mobile devices. 05 2022.

[33] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 461–472, New York, NY, USA, 2016. Association for Computing Machinery.

[34] ID Intrinsic. White paper-sram-puf: The secure silicon fingerprint, 2017.

[35] Anu Jagannath, Jithin Jagannath, and Prem Sagar Pattanshetty Vasanth Kumar. A comprehensive survey on radio frequency (rf) fingerprinting: Traditional approaches, deep learning, and open challenges. *Computer Networks*, 219:109455, 2022.

[36] Kyungho Joo, Wonsuk Choi, and Dong Hoon Lee. Hold the door! fingerprinting your car key to prevent keyless entry car theft. In *NDSS*, 2020.

[37] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. DRAWNAPART: A device identification technique based on remote GPU fingerprinting. In *Network and Distributed Security Symposium (NDSS 2022)*, 2022.

[38] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. T2pair: Secure and usable pairing for heterogeneous iot devices. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 309–323, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Chi-Wei Lien and Sudip Vhaduri. Challenges and opportunities of biometric user authentication in the age of iot: A survey. *ACM Comput. Surv.*, 2023.

[40] Karim Lounis and Mohammad Zulkernine. Lessons learned: Analysis of puf-based authentication protocols for iot. *Digital Threats*, feb 2022.

[41] Priyanka Mall, Ruhul Amin, Ashok Kumar Das, Mark T. Leung, and Kim-Kwang Raymond Choo. Puf-based authentication and key agreement protocols for iot, wsns, and smart grids: A comprehensive survey. *IEEE Internet of Things Journal*, 9(11):8205–8228, 2022.

[42] Thomas McGrath, Ibrahim Ethem Bagci, Zhiming M. Wang, Utz Roedig, and Robert James Young. A puf taxonomy. *Applied Physics Reviews*, 2019.

[43] Reem Melki, Hassan N. Noura, and Ali Chehab. Lightweight multi-factor mutual authentication protocol for iot devices. *International Journal of Information Security*, 19(6):679–694, dec 2019.

[44] John V Monaco. Device fingerprinting with peripheral timestamps. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1018–1033. IEEE, 2022.

[45] K. Nimmy, Sriram Sankaran, and Krishnashree Achuthan. A novel lightweight puf based authentication protocol for iot without explicit crps in verifier database. *Journal of Ambient Intelligence and Humanized Computing*, 08 2021.

[46] TJ OConnor, Dylan Jessee, and Daniel Campos. *Through the Spyglass: Towards IoT Companion App Man-in-the-Middle Attacks*, page 58–62. Association for Computing Machinery, New York, NY, USA, 2021.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[48] Katharina Pfeffer, Alexandra Mai, Adrian Dabrowski, Matthias Gusenbauer, Philipp Schindler, Edgar Weippl, Michael Franz, and Katharina Krombholz. On the usability of authenticity checks for hardware security tokens. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 37–54. USENIX Association, August 2021.

[49] Kasper Bonne Rasmussen, Marc Roeschlin, Ivan Martinovic, and Gene Tsudik. Authentication using pulse-response biometrics. In *The Network and Distributed System Security Symposium (NDSS)*, 2 2014.

[50] Jason Reid, Juan M Gonzalez Nieto, Tee Tang, and Bouchra Senadji. Detecting relay attacks with timing-based protocols. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 204–213, 2007.

[51] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 237–249, New York, NY, USA, 2010. Association for Computing Machinery.

[52] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1502–1514, 2018.

[53] Nimesh Shah, Durba Chatterjee, Brojogopal Sapui, Debdeep Mukhopadhyay, and Arindam Basu. Introducing recurrence in strong pufs for enhanced machine learning attack resistance. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):319–332, 2021.

[54] Mridula Singh, Patrick Leu, and Srdjan Capkun. Uwb with pulse reordering: Securing ranging against relay and physical-layer attacks. *Cryptology ePrint Archive*, 2017.

[55] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[56] Girish Vaidya, Akshay Nambi, T.V. Prabhakar, Vasanth Kumar T, and Suhas Sudhakara. IoT-ID: A novel device-specific identifier based on unique hardware fingerprints. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 189–202, 2020.

[57] Arunkumar Vijayakumar, Vinay C. Patil, Charles B. Prado, and Sandip Kundu. Machine learning resistant strong puf: Possible or a pipe dream? In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 19–24, 2016.

[58] Sergei Volokitin and Riscure Alyssa Milburn. Software Attacks on Hardware Wallets. In *Blackhat*, February 2018.

[59] Wei WANG, Lin Yang, and Qian Zhang. Resonance-based secure pairing for wearables. 17(11):2607–2618, nov 2018.

[60] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G Edward Suh, and Edwin C Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *2012 IEEE Symposium on Security and Privacy*, pages 33–47. IEEE, 2012.

[61] Henry Wong and Tony Luo. Man-in-the-middle attacks on mqtt-based iot using bert based adversarial message generation. In *KDD 2020 AIoT Workshop*, 08 2020.

[62] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, May 2019.

[63] Xinyi Xie, Kun Jiang, Rui Dai, Jun Lu, Lihui Wang, Qing Li, and Jun Yu. Access your tesla without your awareness: Compromising keyless entry system of model 3. In *NDSS*, 2023.

[64] Christian Zajc, Markus Haberler, Gerald Holweg, and Christian Steger. Generating a puf fingerprint from an on-chip resistive ladder dac and adc. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–7, 2021.

[65] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 638–655. IEEE, 2019.

[66] Jiliang Zhang and Chaoqun Shen. Set-based obfuscation for strong pufs against machine learning attacks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(1):288–300, 2021.

[67] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

APPENDIX A
DETAILS OF DESIGNED TASKS

The following are the designed details of the tasks for 6 features, along with their arguments and corresponding outputs (i.e., fingerprints):

DAC/ADC. We use DAC to convert a number to an analog voltage and use ADC to read it, then calculate the error

between the read voltage and the theory voltage as the output. The arguments are, (1) the value of DAC input; (2) the working state of voltage drain drain; (3) the format of ADC voltage, the raw value or the corrected value; (4) the output mode, including different error representations and different pins of ADC.

FPU. We use the calculation of Mandelbrot fractal calculating, which is a way to test the speed of FPU. The arguments are, (1) whether FPU is used; (2) the x-bound of Mandelbrot set; (3) the y-bound of Mandelbrot set. The output is time spent of calculating.

PWM. We utilize ADC to measure voltages generated by PWM, and the output is calculated as the sum of voltages over multiple periods. The arguments are, (1) the clock source of PWM; (2) the frequency of the clock; (3) the number of measured period; (4) the working state of voltage drain drain; (5) the duty ratio of PWM.

RTCFre and RTCPha.During frequency testing, we measure the time it takes to complete several periods as the output. The arguments are: (1) the clock source; (2) the number of clock division; (3) the adjusting value, which is used to adjust clock during different environments; (4) the number of measured period. Also, we measure the instantaneous phase of source clock. The arguments are: (1) the clock source; (2) the number of clock divisions; (3) the supposed period of clock ticking.

SRAM. The input is a target address (required to be 4-aligned) of SRAM as the start address. And we make the following 32-bit (contains the start address) into an integer as output.

APPENDIX B
SECURITY PROOF OF MCU-TOKEN

MCU-Token relies on data poisoning to defend against software mimic attacks and uses the message mapping algorithm to defend against tampering attacks. In this section, we provide theoretical proofs of the security of these two key designs. Combined with the analysis in § V, we can demonstrate the security of the entire system of MCU-Token.

A. Proof of Data Poisoning Effectiveness

Influence of Poisoned Data on Model Learning. We formulate how the poisoned data can affect the model to learn a linear mapping as a regression problem and solve it:

Problem statement: There is mapping $Y = a * X + b$, and pairs in the mapping can be formatted as (x, y). Now, we transform all (x, y) to $(x, y'), y' = cy + d$. If an attacker learns with the modified pairs and aims to make the mean square error as small as possible, what mapping will be learned?

Solution: The mapping learned by the attacker is still a linear one, which can be formatted as $Y' = a' * X + b'$. According to the least square method, we can calculate a', b'.

$$\begin{aligned} a' &= \frac{\Sigma(x-\overline{x})(y'-\overline{y'})}{\Sigma(x-\overline{x})^2} \\ &= \frac{\Sigma(x-\overline{x})(cy+d-c\overline{y}-d)}{\Sigma(x-\overline{x})^2} \\ &= ca \\ b' &= \overline{y'} - a'\overline{x} \\ &= c\overline{y} - ca\overline{x} + d \\ &= cb + d \end{aligned} \tag{4}$$

The Effectiveness of MCU-Token's Poisoned Fingerprints Identification. We show how MCU-Token identifies the poisoned fingerprints based on the solution. The backend checks whether a fingerprint is legal or not by comparing it to the raw fingerprint and can tolerate the hardware bias. If a fingerprint is within the bias, the backend accepts it, otherwise the fingerprint is rejected. We use y for the fingerprint and x for the arguments. According to the proof, when it comes to a new x, an attacker will give a y'. Compared to the original y we have $\Delta = |\frac{y-y'}{y}| = (c-1) + |\frac{d}{ax+b}|$. As long as Δ is greater than the hardware bias, y' will not match the original y and will be rejected by the backend.

B. Proof of Message Mapping Algorithm' Security

Proof of the Hash Collision Security. To analyze the security of Algorithm 1, we construct and solve a hash collision problem and prove it can ensure collision security under our settings.

Problem statement: There is hash function H whose output space size is d. Payloads are p_0, p_1. c is a command. The results are $h_0 = H(c||p_0||p_1), h_1 = H(c||p_1||p_0)$. The purpose of the attacker is to modify c to c' and to keep $h'_0 = h_0, h'_1 = h_1$. Solve the probability.

Solution: The c' is fixed and the attacker modifies p_0, p_1. For a fixed p'_0, as the output of H is uniformed,

$$P\{h_0 = h'_0 | p'_0, p'_1\} = \frac{1}{d} \tag{5}$$

Although in $h_0.h_1$ the difference is only the position of p_0, p_1, the outputs will be completely different and also be seen as uniform.

$$P\{h_0 = h'_0, h_1 = h'_1 | p'_0, p'_1\} = \frac{1}{d^2} \tag{6}$$

We assume there are n kinds of combinations for p_0, p_1, Then

$$P\{h_0 = h'_0, h_1 = h'_1\} = 1 - (\frac{d^2-1}{d^2})^n \tag{7}$$

A Step-by-step Analysis of Algorithm 1. Based on this proof, we explain the security of the whole message mapping algorithm. We use the case where there are only two tasks, i.e. the algorithm produces 2 tasks, each of which has d_0 different possible values (we use "space size" to represent the number of possible values). The attacker aims to tamper with the request and keep the output tasks the same.

With only a single hash function: if the generation of two tasks is independent, the attacker only needs to tamper

Table VIII: Different combinations of models.

	RandomForest	ExtraTree	DecisionTree
RandomForest	0.85,0,09	0.83,0.08	0.83,0.08
ExtraTree	0.85,0.09	0.84,0.08	0.84,0.08
DecisionTree	0.85,0,09	0.83,0.08	0.84,0.08

with tasks whose space size is d_0 twice. In total, the attacker's attempt times are at most $2d_0$.

With h_1 (line 8): h_1 links the generation of the two tasks. If the attacker replaces the operation or the nonce, both tasks will change, which means that the attacker needs to consider the space size of the combination of two tasks. The space size of the combination of two tasks is $d_0 * d_0$, i.e. the attempt times are at most d_0^2.

With h_1, h_2 (line 10): h_2 provides integrity protection for payloads, but allows the attacker to modify payloads to keep the same tasks. The attacker can modify h_2 respectively for the two tasks, and the attempt times are at most $2d_0$.

With h_1, h_2, h_3 (line 12): To solve the problem posed by h_2, we add h_3. According to the proof, with h_3 the attacker's attempt times are at most d_0^2. Furthermore, if there are more generations of tasks linked together, the difficulty for attackers to manipulate the request will increase.

In MCU-Token, the output space of ESP32S2 devices is 20,000 (i.e. d), and the success rate for an attacker is about 2% (i.e. P) with 10^7 (i.e. n) attempt times.

APPENDIX C
PERFORMANCE OF DIFFERENT MODEL COMBINATION

We compare the effectiveness of different models in predictors and verifiers of MCU-Token backend's authentication service. These models are tested on the DAC/ADC features of ESP32S2 devices. The TPRs and FPRs are shown in Table VIII. The predictor models are in the rows and the verifier models are in the columns. The results show that different models have little effect on MCU-Token.

APPENDIX D
ARTIFACT APPENDIX

This artifact contains the source code of MCU-Token and the instructions to run it. MCU-Token is designed for authenticating embedded devices via hardware fingerprinting. To evaluate the basic functionality, you need at least one of these development boards: ESP32S2, STM32F429, or STM32F103. If you do not have any of these devices, we offer a demo that can be executed on the Renode emulator to showcase the functionality. Furthermore, we provide a dataset obtained from our physical devices, allowing you to reproduce the paper's experimental results without necessitating any IoT hardware.

A. Description & Requirements

1) How to access: All the documents and source code are available on github: https://github.com/IoTAccessControl/MCU-Token/tree/master. And the DOI link is https://zenodo.org/doi/10.5281/zenodo.10117167.

2) Hardware dependencies: MCU-Token is implemented on the following devices, ESP32S2, STM32F429, and STM32F103. Make sure you have at least one of them to collect fingerprint data and validate the results.

3) Software dependencies: In summary, to compile the source code and deploy MCU-Token, you need to install one of the following software.

- ESP32-idf for ESP32S2
- Keil for STM32F429 and STM32F103
- Renode for emulation

4) Benchmarks: None.

B. Artifact Installation & Configuration

1) Install Python(3.8) and other required software.
2) Clone the source code from the repo.
3) Install the requirements from requirements.txt.

C. Experiment Workflow

The detailed steps for generating fingerprints for a device and evaluating them are as follows:

1. Install MCU-Token on your devices and ensure that the wires are connected correctly (according to Device-porting/README.md).

2. Collect training data through the serial port. We provide shell scripts to collect fingerprint data (see MCU-Token/server). For example,

```
python bat_generator.py
    --device_number 0 --port COM3
```

3. Evaluate the accuracy of fingerprint verification. We provide shell scripts for generating logs which contain the core results for generating the figures in our paper (see reproducable). Such as,

```
bash 0_generate_ref_log.sh
```

D. Major Claims

- (C1): We can generate a hardware-based access token for each command and collect fingerprints for each device. This is proven by the experiment (E1).
- (C2): We evaluate the performance of the tokens (hardware fingerprints) with different settings. Including the accuracy of authentication, the robustness in different environments and the effectiveness of defense against three types of attacks. This is proven by the experiment (E2).

E. Evaluation

1) Experiment (E1): [30 human-minutes]: Generate hardware-based access tokens for commands and extract hardware fingerprints for devices. Details are shown in Device-porting/README.md.

[Preparation] Install MCU-Token on your device or open Renode. If you are using a physical device, make sure the wire connection is correct.

[Execution] For a physical device, open the serial port and use the "token_gen" command to generate a token for the command. For example,

```
token_gen SET_TEM SEAT1 25
```

Use the "fp_gen" to extract hardware fingerprints, for example,

```
fp_gen STM32 11010 0 0
```

If you do not have a physical device, you can follow the steps in the document to use Renode.

[Results] If you use "token_gen", the command token is printed to the serial port. If you use "fp_gen", the results of fingerprint tasks are printed to the serial port. In a physical device, you will see u8 serials (unreadable). And in Renode you will get readable results (strings).

2) Experiment (E2): [30 human-minutes + 2 compute-hours]: Evaluate the performance of the hardware tokens (fingerprints). The details are shown in document Reproducable.

TL;DR Run Step-2 and get the results in our paper.

[Preparation] Install Python(3.8).

[Execution] Step-1: You can generate the evaluation results with the provided "*_log.sh" scripts (may take several hours). After replacing the original results with yours, you can use the plotting programs to get the figures and tables that are similar to or the same as those in the paper. You can train your own models based on the dataset provided by us and evaluate the data of your devices. Step-2: You can run the plotting programs to get the figures and tables based on our data, for example,

```
python3 Fig-2_plot.py
```

[Results] With Step-1, you can get the raw results logs and get the evaluation results. With Step-2, you can reproduce all the tables and figures in our paper.