IDA: Hybrid Attestation with Support for Interrupts and TOCTOU

Fatemeh Arkannezhad
UCLA
fatemeharkan@ucla.edu

Justin Feng
UCLA
jfeng10@ucla.edu

Nader Sehatbakhsh
UCLA
nsehat@ucla.edu

***Abstract*—Remote attestation has received much attention recently due to the proliferation of embedded and IoT devices. Among various solutions, methods based on hardware-software co-design (hybrid) are particularly popular due to their low overhead yet effective approaches. Despite their usefulness, hybrid methods still suffer from multiple limitations such as strict protections required for the attestation keys and restrictive operation and threat models such as disabling interrupts and neglecting time-of-check-time-of-use (TOCTOU) attacks.**

In this paper, we propose a new hybrid attestation method called IDA, which removes the requirement for disabling interrupts and restrictive access control for the secret key and attestation code, thus improving the system's overall security and flexibility. Rather than making use of a secret key to calculate the response, IDA verifies the attestation process with trusted hardware monitoring and certifies its authenticity only if it was followed precisely. Further, to prevent TOCTOU attacks and handle interrupts, we propose IDA+, which monitors program memory between attestation requests or during interrupts and informs the verifier of changes to the program memory. We implement and evaluate IDA and IDA+ on open-source MSP430 architecture, showing a reasonable overhead in terms of runtime, memory footprint, and hardware overhead while being robust against various attack scenarios. Comparing our method with the state-of-the-art, we show that it has minimal overhead while achieving important new properties such as support for interrupts and DMA requests and detecting TOCTOU attacks.

I. Introduction

With advancements across the hardware-software stack, smart and embedded devices are becoming more popular. Their wide range of applications, such as manufacturing, health care, infrastructure, and homes, have made these devices a prime target for attackers [4], [9], [18], [25], [53], [60].

These resource-constrained low-end devices, typically referred to by various names including internet-of-things (IoT) devices, cyber-physical systems (CPS), embedded systems, and/or "smart" devices[1], often interact with the physical world using various sensors and actuators, and hence are susceptible to a variety of different attacks [4], [9], [41], [60].

[1]For the rest of this paper we refer to this class as IoT devices.

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.2024.23059
www.ndss-symposium.org

Due to hardware and power constraints, securing low-end IoT devices is quite challenging as any feasible solution should be quite low overhead yet fairly effective. One of the key solutions in this realm is remote attestation (RA) mechanisms [12], [13], [30], [42], [44], [52], [57], where the targeted (remote) device can systematically and securely provide information about its software state to a remote and trusted verifier. Successful execution of an RA mechanism allows users to establish trust with a remote user and further interact with it (e.g., sending and/or receiving commands and data, receiving proofs of execution, etc.).

In high-end systems, RA is implemented purely in hardware, often relying on trusted execution environments (TEE) [10], [14], [23], [24], [38]. In low-end IoT devices, however, overheads prevent employing such methods, and hence *software* and/or software-hardware (*hybrid*) methods have been developed [30], [52].

Among the two groups, software attestation approaches (SWATT) rely *only* on software [39], [51], [52], [57]. While quite attractive for low-end devices due to their low overhead and flexibility, lack of any hardware support comes with several critical security issues [16], [57]. Hybrid methods, alternatively, address these issues and provide stronger security guarantees by adding (minimal) hardware support [3], [5], [6], [8], [15], [26], [30], [36], [44], [50].

There are a wide variety of effective hybrid RA methods addressing a range of concerns and important considerations. These techniques, however, commonly suffer from several *limitations* imposed by the methodology. Particularly, to guarantee the secure and correct execution of attestation software, the code and cryptographic key for attestation should be protected at all times (typically by using a ROM). Furthermore, to keep the key secret, all key-related CPU and memory operations should be carefully tainted, tracked, and sanitized during and/or after each attestation operation. Such requirements, make hybrid methods vulnerable to various attack scenarios such as ROP [20] and recently shown attacks by Bognar *et al.* [12]. Additionally, most hybrid RA methods are unable to handle interrupts and/or DMA requests during attestation and are susceptible to time-of-check-(to)-time-of-use (TOCTOU) attacks [26], hence additional requirements are needed to protect them.

To address these limitations, in this paper, we design a new hybrid RA method that does not require secure access

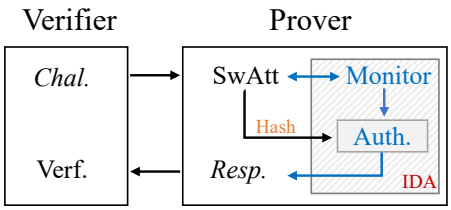

Fig. 1. Using hardware support, IDA monitors the execution of hash computation during attestation and provides authentication if the procedure was followed correctly. This eliminates the need for key access control and allows interrupts to be handled *during* the attestation.

control for the key while being able to handle interrupts, DMA requests, and TOCTOU attacks.

The ***primary idea*** in this work is that instead of using a key during the attestation to verify the hash for each memory address (using an HMAC), we can rely on hardware monitoring to guarantee the correct execution of the attestation protocol. We only use the key to cryptographically sign the final response when the response computation is indeed trustworthy and authentic. The *benefit* of this method is that it provides the flexibility of a keyless software-based RA while achieving similar or better security guarantees of a hybrid-based method.

Fig. 1 shows an overview of our approach called IDA (interruptable hybrid attestation). Upon receiving a remote attestation request, the target device's CPU begins the attestation process by invoking the attestation software (SW-Att). Using various control signals described in Section IV, the hardware module (HWM) ensures the correct execution of the attestation software and its underlying secure hash computation (e.g., SHA3-256). To provide authenticity for the computed hash and prevent the adversary from forging the response, HWM leverages its internal authentication module based on HMAC to authenticate the computed hash. The final response is then sent back to the verifier. This guarantees that a correct response can only be produced when *(a)* the hash computation has been correctly executed (by HWM monitoring); *(b)* the content of memory is correct (by checking the hash value itself); and *(c)* the response is authentic and approved by HWM (by using the authentication module within HWM). Details of SW-Att and HWM units are extensively described in Section IV, and the security and soundness of our method are analyzed in Section VI.

Additionally, by leveraging hardware monitoring and removing the need for secure key access control during attestation, IDA is able to handle interrupts and DMA requests *during* attestation[2]. Requirements for achieving such functionality securely are described in Section III. We design a novel hardware-software algorithm to securely support handling interrupts and DMA requests. The details of the design and implementation are provided in Section IV.

To further support protection against time-of-check-time-of-use attacks, an important and critical attack in existing hybrid methods, as well as improve the system's ability in handling interrupts during attestation, we extend IDA and propose IDA+. It further extends the hardware module to track modifications to the device's program memory *between* attestation requests or *during* interrupts, and proposes a novel mechanism to track and report this information during a remote attestation procedure. We describe the details of IDA+ and how it can help handle interrupts and TOCTOU attacks in Section V.

To evaluate the effectiveness of IDA and IDA+, we implement them on OpenMSP430 [32] and report various critical metrics including hardware, runtime, power, and memory overheads. Further, we compare IDA and IDA+ with several state-of-the-art hybrid attestation methods and highlight the key differences and advantages.

In short, this paper makes the following contributions:

- A new design for hybrid remote attestation based on monitoring hash computation and response authentication using hardware support.
- A novel hybrid mechanism to support interrupts and DMA requests during RA.
- Designing a new mechanism for detecting and reporting TOCTOU attacks in hybrid RA.
- Implementation of the proposed methods on real hardware and comparing the design with state-of-the-art using various metrics. The *open-source* implementation of our framework can be accessed here: https://github.com/ssysarch/IDA.

The rest of this paper is organized as follows: In Section II, we provide a brief background on existing RA methodologies. The threat model, assumptions, and required security properties in our system are described in Section III. The details of our design are provided in Section IV. Further, in Section V, we present IDA+ and show how it can protect the system further. Analysis of the security and soundness of our approach is provided in Section VI. Evaluation and comparisons are presented in Section VII. Related work is discussed in Section VIII, and the paper is concluded in Section IX.

II. Background

Establishing trust for an execution environment remotely is an important problem, and practical solutions for it rely on remote attestation (RA), a security primitive that allows a trusted system (verifier) to verify the integrity of program code, execution environment, data values, etc. in an untrusted remote system (prover).

RA typically relies on a challenge-response paradigm, where the prover is asked to compute a response for a given request. The response computation typically involves measurement (e.g., checksum) of the prover's execution environment (i.e., code and/or data), which the verifier checks against expected values for a "clean" (trustworthy) system. The

[2]This can not be achieved in existing methods since secure cleanup (removing the key and any tainted memories) cannot be properly performed when an interrupt happens.

verifier considers the prover's integrity to not be compromised if *(a)* the response provided by the prover matched with the expected value computed by the verifier, AND *(b)* the verifier believes that the computation that produced the response itself has not been tampered with, e.g., by producing expected values without computing them from the verifier's actual code/data.

In high-end modern processors, the assurance that the response computation itself was not tampered with is typically provided by using a dedicated hardware module called Trusted Execution Environment (TEE) [14], [23], [38]. In low-end IoT devices, however, form factor, battery life, and other constraints prevent the use of hardware-supported enclaves or other hardware supports. Instead, two alternative solutions are proposed, *software* attestation (SWATT) [52] and *hybrid* methods [30].

In SWATT [39], [51], [52], [57] the attestation code is executed normally on the hardware with no protection. To enforce security, *request-to-response time* is leveraged as a way to establish confidence about the integrity of the response computation itself. To implement this, the verifier utilizes the challenge-response paradigm by asking the prover to compute a checksum of its program memory, while measuring the response time to prevent the adversary from computing a correct response, e.g., by temporarily restoring the program memory while the response is computed or by forwarding the challenge to another system that computes the response, etc. The prover passes the attestation test only if it provides the correct response to the challenge without violating the timing requirement. The significant advantage of this method is flexibility since it does not require any hardware support. Fig. 2(a) shows an overview of this approach.

Unfortunately, in SWATT, the overall request-to-response time provides only one coarse-grained measurement, and this method is not able to monitor the prover during the attestation without imposing significant performance and cost overheads on the system. This, in turn, makes the software attestation schemes vulnerable to attacks that have very low latency compared to the overall response time [57]. Moreover, due to the network limitations and/or micro-architectural events, this request-to-response time may be noisy since it includes the round-trip network latency and/or variations caused by the micro-architectural events (e.g., a cache miss) which consequently, makes these schemes even more vulnerable to low-latency attacks.

Alternatively, hybrid methods [3], [5], [6], [8], [15], [26], [30], [36], [44] allow the attestation software to run on the CPU while ensuring its security using some (minimal) hardware support by computing an authenticated integrity check over its memory. The authenticated integrity check can be realized as a Message Authentication Code (MAC) over the prover's memory. However, computing a MAC requires the prover to have a unique secret key, k, shared with the verifier in an offline phase. This key must reside in secure storage, where it is not accessible to any software running on the prover, except for the attestation code. An overview of this approach is shown in Fig. 2(b).

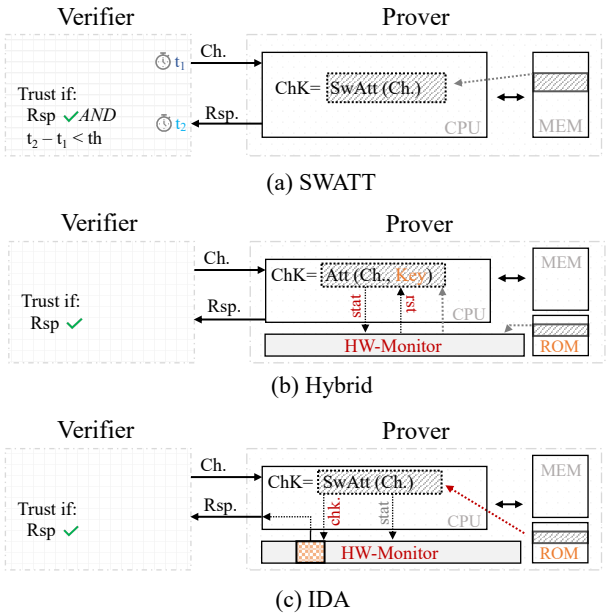

Fig. 2. A comparison between our proposed method, IDA, an established software-based (SWATT) and hybrid-based remote attestation methods. The main difference between the proposed method and existing hybrid methods is that IDA ensures security by hardware monitoring instead of leveraging a key during attestation (*rst*: reset signal).

The requirement for key storage and access control also prevents hybrid methods from supporting interrupts during attestation since memory lines tainted by the secret key need to be first securely cleaned up before the attestation is finished. Bognar *et al.* [12] has recently shown that state-of-the-art hybrid methods are subject to key leakage vulnerabilities as well as novel side-channel DMA and interrupt-based attacks [12], [58]. Other than these issues, hybrid RA methods are also limited in protecting the system against TOCTOU attacks [26].

We propose IDA to address the above issues in software and hybrid RA methods. The goal is to achieve a method that provides the flexibility of a SWATT while addressing the security vulnerabilities that exist in hybrid methods. The main idea relies on utilizing a novel hardware monitoring method that relaxes the requirement for key access control while designing a controller that can handle interrupts and other requests during attestation. Fig. 2(c) depicts our design at a high level. Details of our design and requirements for it are presented in Sections IV and III, respectively.

We also further compare various hardware, hybrid, and software attestation methods in Table I. We compare IDA with various modern software and hybrid methods including Sancus [42], ACES [21], VRASED [44], RATA [26], Real-SWATT [57], GARROTA [3], and PISTIS [33].

To summarize, compared to the state-of-the-art, our method *eliminates* the need for secure key storage and access control, eliminates the need for a secure reset, while allowing the interrupt and DMA requests to be services, and protects the system against TOCTOU attacks.

TABLE I
IDA VS. STATE-OF-THE-ART SOFTWARE AND HYBRID ATTESTATION METHODS (HW-SUPP: HARDWARE SUPPORT, COMP-SUPP: COMPILER SUPPORT).

Model	HW-Supp.	Comp-Supp.	Interrupt	DMA	Need Access Control	Need Reset	TOCTOU	Native Execution
Sancus [42]	Yes	No	x	x	Yes	Yes	x	✓
ACES [21]	No	Yes	✓	✓	No	No	x	x
VRASED [44]	Yes	No	x	x	Yes	Yes	x	✓
RATA [26]	Yes	No	x	x	Yes	Yes	✓	✓
RealSWATT [57]	No	No	x	x	Yes	No	x	✓
GAROTA [3]	Yes	No	x	x	Yes	Yes	x	✓
PISTIS [33]	Yes	Yes	✓	✓	Yes	No	x	x
IDA (ours)	Yes	No	✓	✓	No	No	✓	✓

III. OVERVIEW AND ASSUMPTIONS

A. Threat Model and Scope

We target single-thread, yet multi-tasking bare-metal applications that are the most common in the IoT domain. This class of computing devices (referred to as IoT or MCU) has a single core and features Flash, SRAM, and ROM memories. They execute instructions in place (in physical memory) and have no memory management unit to support virtual memory.

We assume that the adversary, *Adv*, has complete control over the software state, code, and data of the target MCU (prover). In particular, *Adv* can modify any writable code on the device and learn any code and/or data that is in the device's memory and/or CPU. Furthermore, *Adv* has complete control over the communication channel and can use multiple colluding devices in order to pass or subvert attestation.

Similar to prior work, we assume that the attestation code is stored in ROM and cannot be modified by *Adv*. We examine two scenarios: if the adversary's physical access is taken into account, the ROM must be safeguarded using one-time programmable (OTP) memories. Alternatively, if only remote attackers are considered (as assumed in most prior works), then ROM protection is assured provided that our design is installed correctly on the embedded device by a trusted party. *Unlike prior work*, however, we assume that *Adv* can freely read the content of ROM (since no key is stored in ROM)[3]. Additionally, as mentioned earlier, the execution of the attestation code can be interrupted.

We assume that the hardware is implemented correctly and strictly adheres to design specifications. We also assume that the design is free of hardware Trojans [11], hence monitoring signals in IDA that are read from the CPU and/or memory is untampered, and code stored in ROM is protected by the hardware (can be read but not modified). These assumptions are standard in prior methods [33], [42], [44].

We further assume that the hardware monitoring unit, IDA, is correctly implemented and that all code and data within this hardware unit cannot be modified and/or tampered with. We assume that the verifier and the HWM share a secret key. This key is used to verify the authenticity of the final response (i.e., by using an HMAC within HWM[4]). Unlike prior work, this key never leaves HWM. This key can be pre-loaded onto HWM at production time or later. Similar to prior work, we do not address the details of this procedure.

[3]Since IDA is leveraging hash computation instead of HMAC, no key is needed during hash computation.

B. Security Protections and Considerations

In the following, we describe, at a high level, the sub-properties that are required to achieve security in our system. In Section IV, we will present our design and show how IDA can achieve these properties.

Security Properties. For clarity, we show the attestation software as Sw-Att, the hardware module that is part of IDA as HWM, the target device, prover, as Prv, and the verifier as Vrf. Our design should have the following properties:

P1) Immutability: Sw-Att executable should be immutable. Otherwise, the adversary is able to hide the malware by redirecting the memory requests to their desired addresses where the clean copy of the memory is stored (this is referred to as memory copy attacks) and/or moving the malware during the attestation process.

P2) Functional Correctness: Sw-Att must implement the expected behavior of Prv in the attestation protocol. Specifically, Sw-Att should compute the (secure) hash of memory in the address range requested by the challenge using the initialization vector that is sent by either Prv or HWM.

P3) Interrupt/DMA Reset: Unlike prior work, the attestation *can* allow interrupts and DMA requests during the attestation. However, once an interrupt/DMA request is handled and the interrupt request/DMA write flag is cleared, the attestation process should be reset and start fresh using the original nonce. It is important to mention that while the reset is needed for the baseline design in IDA, later in Section V, we *relax* this requirement by designing an improved version called IDA+.

P4) Implementation Correctness: Apart from software execution, HWM and its interaction with CPU and memory should be correctly implemented in hardware and strictly adhere to the design. Specifically, the implementation should be Trojan-free, and the hardware parameters (e.g., memory addresses, ROM, interrupt request flag, PC) monitored and/or used by HWM should be untampered.

[4]Note that the response computation is keyless, i.e., a regular hash function is used for that process. The HMAC is needed to certify the final response. Details will be provided in Section IV.

As will be discussed later, our implementation is able to achieve all these properties.

Relaxed Properties. Apart from these required properties, it is also useful to highlight other requirements that are commonly used in prior work but are not needed in our design. *This is to showcase the versatility and flexibility of our approach compared to the state-of-the-art.*

R1) Atomic Execution: One critical requirement in prior work is the need for atomic execution of SW-Att as any interrupt during the attestation could potentially leak the secret key, and/or affect the integrity of the response. *IDA does not need such a requirement.* We will describe how we achieve this in the next section.

R2) Invocation Flexibility: To avoid secret key leakage, prior work requires enforcing controlled invocation of SW-Att – i.e., Sw-Att must always start from the first instruction and execute until the last instruction. This was needed to avoid ROP attacks where the adversary invokes part of the attestation code and exploits that to read the key (since the key is only accessible via instructions within SW-Att). *For IDA, however, the partial invocation of Sw-Att is allowed* as we assume that the code is publicly known, and no secret value is being accessed by Sw-Att in our design (i.e., no access control is needed).

R3) Flexible Reset: To support interrupts, reset functionality is needed in IDA. However, unlike prior work, this reset does not need secure erasure where all memory lines and registers tainted by the secret key have to be securely erased before the attestation can be restarted again. This is again due to the fact that IDA does not use a particular secret key during the attestation process.

R4) DMA Monitoring: Since the key access control is no longer needed, monitoring the activity of DMA is more relaxed too. Prior work requires monitoring both reads and writes, as well as corresponding addresses. IDA, however, only need to monitor writes during the attestation. Reads can happen as is. In either case, no address monitoring is needed.

IV. System Design

A. Design Overview

To achieve the security properties described in Section III based on the threat model also explained in the same section, we design IDA. There are various components in our proposed systems which will be explained in this section. The high-level overview of IDA is shown in Fig. 2(c). The more detailed system architecture of the hardware module (HWM) with its internal components are shown in Fig. 3.

In addition to the hardware module which is responsible for monitoring the CPU and ensuring security during attestation, our system consists of three other major parts. The *memory system* houses the ROM, program memory, and data memory (they are all part of a unified address space). The *CPU* unit collectively executes programs and sends and receives commands. Lastly, the *memory backbone* is responsible to connect and manage the connection between these units and other I/O components.

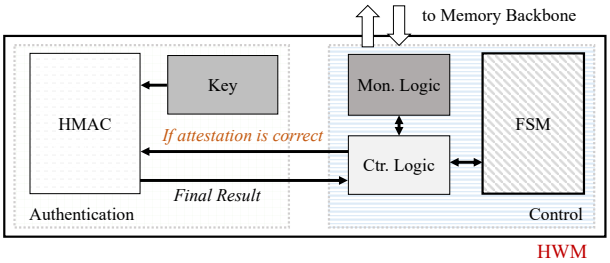

Fig. 3. The systems architecture for the hardware module (HWM) in IDA.

The memory system is connected to the memory backbone which directly interacts with the CPU's front end. The immutability of the attestation software is based on the secure implementation of the ROM. It is important to emphasize that since access control is not needed in IDA, ROM does not need to be directly connected and controlled by HWM hence it can directly be connected to the backbone.

B. System Architecture

Hardware Module (HWM). To enforce secure execution of the attestation code (SW-Att), HWM utilizes a minimal series of signals from and to the CPU and/or backbone. We focus on the attestation functionality of the prover; verification of the entire system architecture is beyond the scope of this paper. Therefore, we assume the system architecture strictly adheres to and correctly implements its specifications. In particular, IDA utilizes the following features in our system:

F1) Program Counter (PC): HWM monitors the PC in every cycle to track the current instruction that is being executed on the CPU. We assume that the PC always contains the address of the instruction being executed in a given cycle.

F2) Memory Address (MemAdr): Similarly, HWM monitors the memory address in every cycle to track the current memory content that is being read by the CPU. We assume that the memory address register always contains the memory address that is being read.

F3) Interrupt Request: HWM also monitors interrupt requests (IRQ). First, HWM receives a signal if and when an interrupt happens. The hardware module also receives a signal when the interrupt is finished and the normal execution of instructions in the CPU has been resumed. Both signals can be received using the same wire. By default, IRQ is low and when an interrupt is activated, the wire becomes high and stays high until the interrupt is handled, and the execution is resumed. Similar to PC, we assume the correct implementation of the interrupt signal in hardware.

F4) DMA Write: Similar to the interrupt requests, HWM should receive a signal when writes are initiated by the DMA (direct memory access) unit. Unlike prior work, *only* writes need to be monitored. We assume that a *DMA write* signal is correctly implemented and is connected to HWM.

F5) Shared Status Register (CSR): To communicate between the hardware module and CPU, a dedicated register is needed. We call this a secure status register (CSR) which is

accessible by both HWM and CPU. The content of CSR does not need to be secret and/or read-only. Both modules can write the register. Writes to CSR can be achieved using memory-mapped I/O (i.e., simple load and store commands) where the address of CSR is hard-coded in SW-Att.

It is important to emphasize that all the above five features are feasible and in fact, satisfied by Open-MSP430 design [32]. More details about the actual implementation are presented in Section VII.

Using the features described above, our hardware module is designed. This module consists of two main units. The *controller unit* and the *authentication unit* as shown in Fig. 3.

The controller unit has a finite-state machine (FSM) that monitors the PC, memory address, and DMA-write and interrupt signals. Depending on these control signals, the FSM transitions between various states. *The objective of the FSM is to ensure that Sw-Att is correctly executed from start to finish.* Further, the FSM monitors DMA and interrupts signals, and *resets* the procedure if either occurs. The detailed description of the FSM is described in Section IV-C and Algorithm 2.

The second part of HWM is the authentication unit. *The objective of the authentication unit is to validate the integrity and authenticity of the attestation process (i.e., the final hash).*

The input sent to authentication in IDA is the final output of the hash function generated by Sw-Att. The idea is that the controller in HWM ensures the correct computation of the hash and the authentication ensures its authenticity and integrity. Together, they guarantee that the procedure is secure and trustworthy. While there are multiple candidates for authentication, we opt for utilizing HMAC-SHA3-256. The final response is then sent back to the verifier. Details of the full algorithm are shown in Algorithm 1 which will be discussed in Section IV-C.

Software Attestation (Sw-Att). The main objective in designing SW-Att is simplicity. It first initializes the hash using the challenge sent by the verifier[5]. It then iteratively goes through the contents of the program memory line-by-line and updates the hash until it covers all lines requested by the verifier. The nonce sent by the verifier guarantees the uniqueness of the computed hash for each challenge, even though the input to the hash function (i.e., program memory) is supposedly the same in each challenge. Such a method is quite popular for storing passwords in databases securely [47] and also used in a prior work [57].

For supporting interrupts and DMA, SW-Att receives an input, $intr$, from HWM. The signal's role is to halt the hash computation when an interrupt and/or DMA has been requested/handled. Once the interrupt/DMA request is complete, the entire process is reiterated to ensure that all memory addresses are correctly hashed without any interruptions. It is important to mention that resetting after each interrupt could create additional performance and/or power overheads. To address this, we propose two solutions. First, we introduce a *watchdog* timer to ensure that the attestation will eventually finish by disabling interrupts after a certain number of interrupts (details later). Second, to further reduce the overhead, we will introduce, IDA+, which eliminates the need for resetting *if* the content of memory is not modified during the interrupt handling. Details will be provided in Section V.

An alternative approach, instead of delaying attestation locally through a reset, is to notify the remote user and let them initiate another attestation request after a period. We note that the "delaying locally" strategy offers several advantages. *First,* it conserves energy since local processing consumes much less power compared to the RF subsystem required for notifying the remote user [31]. Thus, minimizing communication can save energy—trading RF communication with local processing proves beneficial. As explained later, this is particularly advantageous in IDA+ as it eliminates the need for constant restarting. *Second,* most real-time systems continually receive interrupts; hence, notifying the verifier to check back later may not be as effective in terms of energy and performance. The chances that the device is "ready" don't significantly increase after a few seconds/minutes. On the other hand, IDA can initiate the attestation locally as soon as the interrupt is finished—handling this locally transforms the process from having a random chance of initiating interrupts to deterministic, as the interrupt can start right after it's finished.

Full details of the algorithm are presented in Algorithm 1. In the following, we will describe the steps in more detail.

C. Attestation Algorithm

Overall, our attestation framework, IDA, has three parts. The attestation software runs on the CPU without any protection (hence fully controlled by the adversary), and the software and hardware modules (SW-Att and HWM, respectively) together ensure the security and authenticity of the response even in the presence of an adversary.

Algorithm 1 presents the details for the attestation software. As can be seen, the procedure begins by receiving a challenge from the verifier that includes a nonce and optionally an attestation range. Similar to prior work [30], [44], [57], we assume a scenario where, by default, the prover wants to attest the entire program memory. The attestation software then invokes Sw-Att. Note that the attestation's main function is unprotected and controlled by the adversary. This means that the adversary can forge the challenge if desired. We will systematically analyze the security of our system under such an assumption in Section VI.

The attestation procedure continues by invoking SW-Att and passing the control to it. Since IDA can allow interrupts and DMA, this process might need to be repeated multiple times if needed (line 2 in Algorithm 1). We will describe how this part is implemented later.

As described earlier, the goal of SW-Att is to iteratively read memory addresses in the program memory and update the hash. The hash is updated by calling the hash function. Once the hash computation is finished, the value is sent to the hardware module using the special register (CSR). This

[5]The nonce sent by the verifier acts as an initialization vector (IV)/cryptographic salt for the hash function.

Algorithm 1: IDA: Algorithm and SW-Att function

```
1  input: Ch. = {nonce, AR*}, Repeat;
   /* AR: address range to be attested.
      */
2  while Repeat do
3  |  Resp. = Sw-Att(Ch.);
4  end
5  return Resp.;
1  Function SW-Att
2  |  input: nonce, AR, inter;
3  |  CSR = CSR_adr;
   |  /* Address for accessing HWM.      */
4  |  cHash = nonce;
5  |  i = AR_begin;
6  |  while (i < AR_end) do
7  |  |  cHash = hash(Mem[i], cHash);
8  |  |  i++;
9  |  |  if (inter == 1) then  break;
10 |  end
11 |  Mem[CSR] = cHash;
12 |  IdleLoop();
   |  /* To allow time for HWM to
   |     calculate the response.          */
13 |  cHash = Mem[CSR];
14 |  return cHash;
```

is achieved via a memory *store* (line 11 in Algorithm 1), where the address is mapped to the special register (the register itself is implemented inside HWM). The forwarded hash value acts as an input for the authentication unit within HWM (see Section IV-B). The response to the provided input (i.e., the final hash value) is calculated (using HMAC-SHA) and sent back using the same special register (CSR). This is achieved by using a memory *load* (line 13). To consider the delay for response calculation, a delay function is needed (line 12). To implement this securely, an inline delay loop can be implemented where the total number of iterations/instructions decides the appropriate delay time. In Section VII, more details will be provided.

An important optimization is used for SW-Att where the hash computation is terminated when an interrupt and/or DMA request is accepted (line 9). Given that the hash computation should be reset, the *while* loop is terminated as soon as the control signal, $inter$, is activated, indicating that an interrupt/DMA request is being serviced. To repeat the hash computation, SW-Att is repeatedly called within the main function (lines 2-4). This loop stops when the entire hash computation is performed uninterrupted. HWM is responsible to monitor this and controls $Repeat$ control signal.

The details of the controller unit within HWM are presented in Algorithm 2 and shown in Fig. 4. Generally, the controller uses an FSM to monitor the attestation process and issues correct and timely control signals. Initially, the FSM is in "Wait" state where it continuously monitors the PC. Once the control is passed to SW-Att (line 7 in Algorithm 2), the FSM

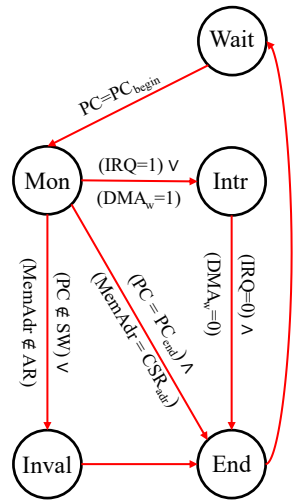

Fig. 4. The control states needed for HWM.

switches to the next state and starts monitoring.

During "Monitor" state, the hardware module monitors the PC and the memory address to ensure that the attestation software is being correctly executed. Any jumps and/or memory redirection will be detected by HWM. Upon detecting such an intrusion, the FSM switches to "Invalid" state (lines 24-27) where it issues a terminate signal for hash computation (i.e., $inter = 0$) and then goes to "End" state (details later). Similarly, if an interrupt or DMA request is detected (lines 13-15), the FSM switches to the corresponding state and waits until the request is cleared. Once the request is cleared/serviced, the FSM switches to "End" state.

There are two ways to reach "End" state. First, through one of the incomplete (i.e., {"Invalid", "Intr"}) states where the hash computation was not correctly computed. Second, via "Monitor" state where hash computation is correctly completed (see line 17). To distinguish between the two, we use a counter, LC, which counts the number of lines being correctly hashed so far. If this is equal to the total number of expected lines (line 29), HWM reads the hash value through CSR and forwards it to its authentication module (i.e., the HMAC). The response is forwarded back to the software and the state resets (i.e., goes back to "Wait") as can be seen in lines 30-32. Additionally, to break the infinite loop in the attestation procedure (see lines 2-4 in Algorithm 1), $Repeat$ is cleared, indicating that the attestation is successfully completed.

Alternatively, if "End" is reached via an incomplete state, the FSM forwards a special response (line 34) indicating that the attestation has failed. Further, FSM switches back to the first state without clearing $Repeat$. This forces the attestation procedure to repeat. This process repeats until the attestation is eventually successful or if it is timed out (lines 36-38). The timeout condition is indicated using a watchdog timer ($WDog$). This value depends on how many interrupts can be serviced during attestation and depends on the application scenario. Choosing a smaller or larger number does not have any correctness and/or security implications and it is only a

Algorithm 2: The controller in the hardware module.

```
1  Module HWM
2      init: State = Wait, CSR = 0, WDog = 0,
        LC = 0, AR, SW;
       /* SW: where SW-Att is stored.    */
3      switch State do
4          case Wait do
5              CSR = 0, LC = 0,
6              inter = 0, Repeat = 1;
7              if (PC == PC_begin) then
8                  State = Monitor // Hash Comp.
9          case Monitor do
10             if (PC not in range SW) OR
11             (MemAdr not in range AR) then
12                 State = Invalid;
13             else if (IRQ == 1 or DMA_w == 1) then
14                 State = Intr;
15                 inter = 1 // Breaking hash
                    loop
16             else if (PC == PC_end) AND
                (MemAdr == CSR_adr) then
17                 State = End
18             LC++ // Counting hashed lines
19         case Intr do
20             CSR = 0;
21             if (IRQ == 0 and DMA_w == 0) then
22                 State = End// To repeat
                    hashing
23                 inter = 0;
24         case Invalid do
25             CSR = 0, LC = 0;
26             State = End;
27             inter = 0;
28         case End do
29             if (LC == LC_AR) then
30                 CSR = AUTH(CSR);
31                 State = Wait, WDog = 0;
32                 Repeat = 0 // Braking the
                    loop
33             else
34                 CSR = NULL;
35                 State = Wait;
36                 if WDog > TH then
37                     Repeat = 0 // No deadlock
38                     WDog = 0;
39                 else WDog++;
40     end
```

performance optimization.

Depending on the user's preference, two options can be provided to ensure the satisfactory operation of this protocol. First, in the event that interrupts keep happening during the attestation, the watchdog timer can just terminate the attestation process (when it is timed out) and send a failure message to the remote user. The user can then take proper action such as trying at some other time and/or performing additional investigations.

Alternatively, the timer can internally fix the issue by temporarily disabling all the interrupts and DMA requests and then running the attestation procedure again. This will ensure that the process will finish but might create practical concerns due to disabling the interrupts for some period of time.

In the next section, we propose a third option where we eliminate the need for resetting after each interrupt. This will satisfy the timeliness requirement without disabling the interrupts which is desired.

We evaluate the security and soundness of our design in Section VI. Before that, we first explain how IDA can be further enhanced by adding support for protecting against TOCTOU attacks and handling interrupts more efficiently.

V. IDA+: Supporting TOCTOU and Interrupts

IDA suffers from a common limitation: IDA and other existing hybrid attestation methods measure the state of the prover's program memory at the time when remote attestation (RA) is executed by the prover. Crucially, they provide no information about the prover's state *before* attestation or *after* (i.e., between two consecutive RA measurements). This is commonly referred to as Time-Of-Check-(to)-Time-Of-Use (TOCTOU) problem [26] where the adversary *hides* the malware *during* the attestation process but manages to recover it *after* passing the attestation test. This, for example, can be achieved by storing the malicious code in other parts of the device's memory (e.g., peripherals [16]).

Note that TOCTOU is different from ensuring temporal consistency between attestation and execution of a binary, which can be solved by runtime attestation approaches [17], [45] (e.g., ensuring that the desired binary is correctly loaded/executed during the attestation process). TOCTOU is important when static RA is needed or when static RA is used as a basic block for other functionalities such as code updates, secure reset, etc.

The *second* concern with IDA is that supporting interrupts requires resetting the attestation process. While for rare and/or occasional interrupts this can be tolerated, for a complex real-time system with many deadlines this will lead to inefficiencies as either the attestation needs to be repeated many times (power and latency overheads), or even worse, the attestation never finishes due to a recurring interrupt. Therefore, *a more practical solution for handling interrupts is needed.*

In this section, we propose IDA+, an enhanced version of our baseline design to protect the system against TOCTOU attacks and to support interrupts while eliminating the need for resets. The ***main idea*** in IDA+ is to monitor the program memory even when an attestation procedure is not being executed where HWM tracks any changes to the program memory and takes proper actions during the attestation. Particularly, in addition to creating a response for a given correctly computed hash, HWM alternatively computes a response (HMAC) when the verifier's nonce is used as the input. The trick, however, is that HWM sends this as a response only if the program hasn't

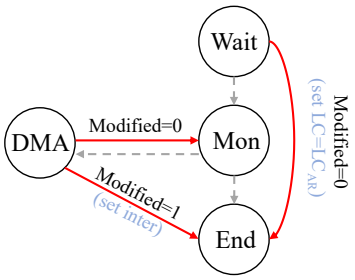

Fig. 5. The changes needed for IDA+ (shown in red) in the FSM within HWM to support TOCTOU.

been modified since the last successful attestation request. Otherwise, the attestation procedure is performed normally and a *NULL* (e.g., a randomly chosen response) is forwarded if the procedure is not followed correctly.

Consequently, based on the response received, the verifier can check the current status of the prover. The correct response when nonce was used indicates that the prover has not been modified. Alternatively, the correct response when the hash was used as input means that the prover has been modified, but the attestation was performed (and passed if the response is correct). Lastly, an incorrect response means that the prover has been modified, and the attestation couldn't be performed. Based on the received response, the verifier can then decide on the proper action. The changes needed are shown in Fig. 5 and Algorithm 3.

To support interrupts, the idea is to monitor the memory during the interrupt handling period. If it is not modified, then the attestation procedure could be continued normally without the need for a reset. Otherwise, the process will be reset (i.e., how it was handled in IDA). There are a few additional considerations for security. Details will be provided later.

To track changes to the program memory, HWM is augmented with an additional control signal named $MemAdr_w$ that becomes active whenever a write to the program memory has been initiated. To also eliminate DMA-based modifications, DMA_w and DMA_{adr} should be tracked. A one-bit flag is then used in HWM to track the current status of the program memory. By default, this flag, named $modified$, is clear. However, as soon as either DMA_w or $MemAdr_w$ are set while the corresponding address is within the program memory range, $modified$ will be set. Note that this check happens in parallel with HWM's FSM, thus $modified$ can be set during any of the possible states in the FSM or even when attestation is not active (i.e., FSM is in "Wait" state).

In addition to this tracking, further security checks need to be added to ensure the integrity of response during an interrupt. Specifically, while memory can be monitored by HWM, internal registers within the CPU cannot. Therefore, IDA+ is potentially vulnerable to attacks where an adversary modifies the registers and/or times the interrupt perfectly to hide the malware.

Looking closer to the Sw-Att shown in Algorithm 1, the

Algorithm 3: IDA+: Changes needed to support TOCTOU and Interrputs

1 **Function** *SW-Att*
2 **input:** $nonce, AR, inter, Repeat$;
3 $CSR = CSR_{adr}$;
4 $Mem[CSR] = nonce$;
5 $IdleLoop()$;
6 **if** *(Repeat == 0)* **then**
7 **return** *Mem[CSR];*
8 $Mem[CSR] = nonce$;
9 $i = AR_{begin}$;
10 **while** $(i < AR_{end})$ **do**
11 $Mem[CSR] = hash(Mem[i], Mem[CSR])$;
// Interrupts are disabled during this line but can happen before or after
12 $i++$;
13 **if** *(inter == 1)* **then** break;
14 **end**
/* Rest is the same. */

adversary could potentially modify any variable in this code during an interrupt without being tracked. As a result, IDA+ needs to ensure *(a)* critical values are properly saved before an interrupt is serviced and *(b)* the interrupt does not happen while a memory line is still being read. More specifically, we need to make sure that the interrupt does not happen during the computations in line 11 of Algorithm 3 (i.e., where the memory is being read or the result is being calculated).

To ensure these, we make *three* changes. First, we replace $cHash$ with $Mem[CSR]$ in the internal loop of Sw-Att (see line 11). This will ensure that the intermediate hash value cannot be modified. Second, we move the interrupt trigger logic to HWM. This will ensure that interrupts can only happen before or after a single memory line read and its corresponding hash calculation. Third, HWM records the last address that was read by SW-Att (i) before the interrupt and checks if the next address requested (i.e., after the interrupt has been handled) matches with the expected address (i.e., $i + 1$).

An additional optimization that is made in IDA+ is that the attestation procedure could resume normally when a DMA request is handled if during the DMA request, program memory within the attestation range (AR) is not modified (i.e., similar to interrupts).

A recent work, called RATA [26], has also recently addressed this problem in hybrid RA. Compared to that work IDA+ has the following advantages. First, IDA+ does not need extra storage overhead to store logs. Only one flag bit is needed to track the memory modifications. Second, similar to other hybrid methods, RATA faces similar vulnerabilities related to storing the key. IDA+, however, is immune against those. Third, IDA+ can support interrupts and DMA requests while the current version of RATA is unable to achieve them.

VI. Security Analysis

The IDA attestation framework uses several new techniques to perform reliable software-based attestation of low-end IoT and embedded devices. To ensure the integrity of an attested device, IDA (and IDA+) must satisfy multiple premises. In the following, we discuss the formal criteria for possible attack models and describe how IDA/IDA+ can protect the system against them.

Attack Model. The goal of the attacker is to win an attestation *game* by providing a correct response to a given challenge while hiding all her malicious traces. Recall that we assume that the attacker has full control over all software, memory, and hardware on the prover except parts that are explicitly protected by IDA, hence the attacker can launch denial of service (DoS) attacks by deciding not to participate in the game. However, this indicates that the prover is compromised and hence further interactions with the device will be terminated. Similar to prior work, IDA can not prevent DoS attacks and here we only focus on scenarios where the prover participates in the game but plans to forge the response in the presence of a malicious memory modification, i.e., $Resp_{Forged} == Resp_{Expected}$ while $M_{Prover} \neq M_{Original}$.

To win the attestation game, the attacker needs to find and implement a method to forge the response. More concretely, the attacker can launch any of the following attacks (T):

T1) Collision Attack: In this type of attack, the goal is to find a collision for the final response by changing the inputs to the attestation procedure. More concretely, the attacker has three options to conduct such an attack.

T1.1.) First, the attacker can change multiple lines in M' such that $Hash(nonce, M') == Hash(nonce, M)$, where M and M' refers to the original and modified (malicious) program memory.

T1.2.) Second, the attacker can find a different nonce, $nonce'$, such that $Hash(nonce', M') == Hash(nonce, M)$, i.e., when both the nonce and multiple lines in the memory are changed to find a collision.

T1.3.) Third, the attacker can find a new hash, H' such that $AUTH(H') == AUTH(H)$, where H refers to the expected hash when M is not modified and correct nonce was used. To find such a hash, the attacker can change either the nonce or the contents of the memory (or both).

T2) Substitution Attack: Alternatively, instead of trying to find a collision, the attacker could try to *alter* critical variables during the attestation process with the ultimate goal of outputting a correct hash (i.e., cheating in the process rather than changing the values). To achieve this, the attacker needs to find a way to substitute values *during* the attestation. This can be achieved via four possibilities:

T2.1.) The attacker can substitute memory contents (program memory and/or CSR) during each memory read and/or in the background after a given line has been visited.

T2.2.) The attacker can substitute/modify the contents of the register file during the attestation process.

T2.3.) The attacker can alter control signals such as PC, memory address, interrupt request, DMA request, etc. which are monitored by HWM.

T2.4.) The attacker can forge the final response by substituting the output of HWM with their own generated response (i.e., completely bypassing attestation by not participating and creating their own responses).

Considering the above attack scenarios, we will analyze how IDA can protect against all scenarios. To analyze this, we first explain our main security assertions and then show how, based on the validity of these assertions, security can be achieved in IDA. We will then extend it to IDA+ by analyzing TOCTOU attacks.

Security Assertions. Our security argument is based on the following assertions:

1) Hash computation in IDA (SHA3-256) is secure against preimage attacks. Informally, given y, it is hard to find x s.t. $y = H(x)$ (first pre-image), where in our setup $x = \{nonce, M\}$. Additionally, given x, it is hard to find $x' \neq x$ s.t. $H(x') = H(x)$ (second pre-image).
2) Each challenge has a unique nonce that has not been used before. Further, using a different nonce guarantees that $H(x)$ is different for each challenge even for unmodified program memory.
3) Authentication module, $AUTH$ (HMAC-SHA3-256) is secure against forgery attacks. Further, the key, K, for the authentication never leaves the tamper-resistant HWM.
4) SW-Att resides in ROM and cannot be modified.
5) IDA strictly adheres to and correctly implements its specifications. In addition to the correct implementation of system hardware specification (e.g., CPU, memory, backbone, and all connections), this also implies that control signals including $PC, MemAdr, MemAdr_w, DMA_w, DMA_{adr}, IRQ$, and CSR, as well as all internal logic in HWM are correctly implemented based on the specification.

Analysis. Using the assertions described above and attack scenarios presented earlier, we now focus on how IDA can successfully defend against those attack scenarios:

Both *T1.1* and *T1.2* are protected based on assertion A1. Informally, finding a new $x' = \{nonce', M'\}$ s.t. $H(x) = H(x')$ is hard when either $nonce' \neq nonce$ and/or $M' \neq M$ – i.e., $H(x) = H(x')$ is true only when both $nonce$ and M are not modified. Note that the second preimage resistance implies that, the adversary can have full knowledge of the requested nonce and the original memory (which is the case given our threat model) and still cannot find a collision.

T1.3 is protected based on assertions A2 and A3. Specifically, using a unique nonce guarantees that the challenge sent to HWM is unique (hence replay attacks are ineffective). Secondly, the forgery resistance of the authentication unit using HMAC asserts that the response is unique thus the final response is different for two different hashes (inputs).

T2.1 is protected based on A4 and A5. First, the hardware is correctly implemented, thus memory requests can not be forwarded during attestation without changing the memory address (which is monitored by HWM). Second, any attempt to modify the memory *during* attestation via interrupts will be detected by HWM and cause a reset in response computation. Third, the final response is forwarded by HWM only if the entire hash computation is uninterrupted (IDA) or interrupted but unchanged (IDA+). Fourth, attempts to change the memory in the background via DMA are detected by HWM and cause a reset. Lastly, attempts to forward the memory requests and/or execute a different application (i.e., a different PC or a different memory address) other than SW-Att are also detected by HWM and cause a reset.

T2.2 is protected since the register file cannot be externally modified given that the system is single-threaded. For IDA+, the critical register, $cHash$, was substituted with $Mem[CSR]$, which resides in HWM and cannot be modified. Any other changes during the interrupt (e.g., changing i) will be detected by HWM since they have been saved prior to handling the interrupt and will be monitored once the attestation is resumed.

A similar argument can be used for *T2.3* thus we don't provide an additional description for brevity.

Lastly, *T2.4* is protected based on A3, and in particular, uniqueness and forgery resistance.

Additionally, IDA+ is resistant to TOCTOU attacks due to the following reasons. First, IDA+ monitors the changes to the program memory. Based on A5, it is guaranteed that any changes to the program memory will be detected by HWM. Further, A3 implies that the response generated by HWM cannot be forged (same as *T2.4*). Together, this guarantees that changes to the program memory cannot stay hidden.

Remarks. It is important to mention that our security argument is informal. A more substantial argument (or proof) would require formal analysis and verification of the entire system and various components of IDA/IDA+, which is planned as part of future work. Here, the goal was to be systematic and comprehensive and clearly explain all possible attack scenarios and how IDA/IDA+ can defend them at a high-level.

VII. Evaluation

Implementation. To implement IDA and IDA+, we use OpenMSP430 [32], an open-source implementation of the MSP430 architecture, as a representative of a low-end IoT/embedded system. OpenMSP430 is written in the Verilog hardware description language (HDL) and can execute software generated by any MSP430 toolchain with near-cycle accuracy. OpenMSP430 has been also widely used in numerous prior hybrid attestation prototypes. Note that, while our choice is motivated by the availability of a well-maintained open-source MSP430 hardware design [32] and its popularity in other similar prototypes, our framework is applicable to other low-end MCUs in the same class as MSP430 (e.g., ARM M0).

We modify the baseline OpenMSP430 implementation to add required functionalities in IDA/IDA+. Specifically, we add a ROM (8KB) to store SW-Att, a new hardware module (HWM), and adapt the design accordingly to properly connect these modules. This is achieved by modifying the memory backbone module. Further, the CPU unit and memory backbone is modified to connect control signals such as PC, memory address, etc., to HWM. The correct implementation of hardware ensures the integrity of ROM against a remote user. For IDA+, the interrupt handling is modified where the global interrupt flag is directly connected to HWM instead of the CPU. To allow interrupts, a new signal called $sGIE$ (secure global interrupt enable) is connected to the CPU. This will ensure that interrupts only happen when it is allowed.

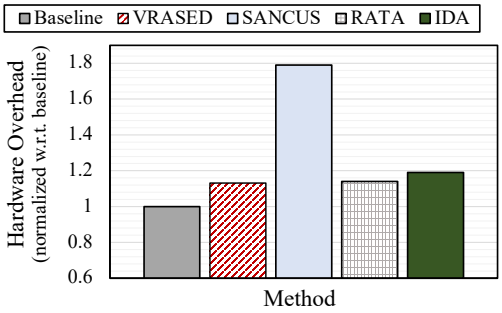

Fig. 6. Hardware overhead for IDA and other state-of-the-art hybrid attestation techniques. Results are normalized w.r.t. the original baseline OpenMSP430 design.

For SW-Att, SHA3-256 from the HACL library [35] is used to compute the hash. For HWM, HMAC-SHA-256 implemented by Secworks [54] is used to authenticate the final response. The unoptimized version of SHA in our hardware simulations takes about 60 cycles (for an 8 MHz clock).

The entire design is implemented on a ZYNQ FPGA using Xilinx/AMD Vivado toolchain. SW-Att is compiled using msp430gcc compiler and then properly loaded to ROM and RAM units on the FPGA.

Hardware/Logic Overhead. The hardware overhead in IDA and IDA+ is primarily due to adding HWM. This module consists of a controller (an FSM and control logic surrounding it) and the authentication module. Comparing the two, the authentication module takes about 54.4% of the entire area in HWM.

To provide a more detailed comparison, we compare our hardware overhead results with the baseline OpenMSP430 design and three recent hybrid methods: VRASED [44], SANCUS [43], and RATA [26]. The overhead is measured as the number of gates (combinational and register) divided by the total number of gates for the baseline design. All numbers are based on the report provided by Vivado. The raw numbers are provided in Table II (see Appendix B).

The results are shown in Fig. 6. To make comparison easier, all values are normalized against the baseline OpenMSP430 design (no attestation). IDA has about 19% overhead compared to the baseline which is mainly due to the logic needed for the FSM and authentication with the authentication being the dominant component.

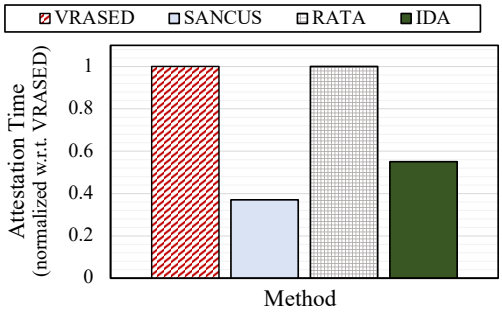

Fig. 7. Attestation time for IDA and other state-of-the-art hybrid attestation techniques. Results are normalized w.r.t. VRASED [44].

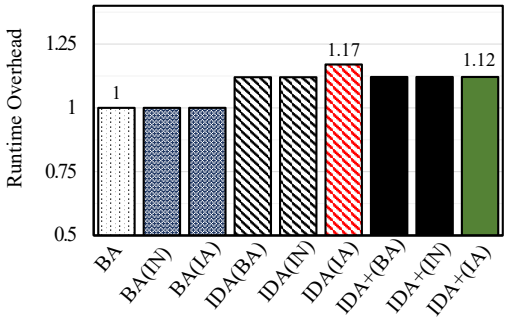

Fig. 8. Average runtime overhead for three applications from Mibench benchmark suite [34]. Numbers are normalized w.r.t. baseline.

Compared to the state-of-the-art, IDA achieves relatively similar overhead results, indicating that our approach is a viable choice. The overhead is slightly higher due to the need to implement the HMAC function in hardware (as opposed to its hybrid implementation in prior work). Apart from the authentication function, the control logic in IDA incurs less overhead compared to the state-of-the-art since it is simpler (up to 1.8x). Specifically, by eliminating functionalities such as secure reset and cleanup, which are needed in prior work, some logic is saved. *More importantly, IDA achieves similar overhead while providing more features including support for interrupt and DMA activity while completely eliminating the need for storing a secure key.*

Attestation Time. The results for attestation (the total time to attest an 8 KB program memory with an 8 MHz clock uninterrupted) are shown in Fig. 7 (raw numbers can be found in Table II in Appendix B). Similar to the hardware overhead results, we compare IDA with other modern techniques. Since the baseline design does not have any support for attestation, results are shown for VRASED [44], SANCUS [43], and RATA [26]. As shown in the figure, *IDA achieves significantly (around 2x) faster attestation time compared to VRASED [44].* This is due to the fact that IDA uses HMAC for the final response instead of verifying the hash for each memory access. Similar to prior work, the majority of the runtime is spent in the hash computation loop. IDA achieves runtime close to SANCUS [43] which is purely implemented in hardware. Comparing the overhead between the two, IDA has a much smaller overhead compared to SANCUS while achieving additional functionalities such as support for interrupts.

It is important to mention that since IDA handles interrupts by issuing a reset, the actual interrupted runtime of IDA could be significantly higher than the uninterrupted version since a recurring interrupt could cause many resets before the watchdog timer is triggered. In this experiment, we did not consider such a scenario and only report the uninterrupted version. However, we will later consider this scenario when comparing the results between IDA and IDA+ and show how IDA+ can solve the reset problem.

Memory Overhead. Similar to prior work, IDA requires around 4 KB of ROM which is needed for storing SW-Att application and particularly, HACL SHA3-256 code. For a commercial-of-the-shelf MSP430 microcontroller with around 128 KB of flash and 8 KB of RAM, this constitutes about 3% memory overhead of the entire system.

Compared to state-of-the-art, IDA achieves similar or smaller overhead. For example, for VRASED [44], the overhead is around 4.5 KB which is for storing the HMAC function and some additional functionality such as secure reset and secure cleanup. *Combining the logic and memory overheads, IDA/IDA+ achieves a very similar area overhead compared to VRASED and RATA.*

Runtime Overhead. To measure the impact of attestation and interrupts on runtime overhead for IDA and IDA+, we do the following experiment. We pick three representative applications (*FFT, Basicmath,* and *dijkstra*) from MiBench[6] benchmark [34] and measure the overall runtime of the application on our hardware. We consider the following scenarios: *baseline (BA):* where only the original application is executed on the CPU, uninterrupted. *Baseline-Interrupted-Normal (BA(IN)):* where the original application is executed on the CPU and a timer interrupt are set to interrupt the program every 500ms (each interrupt takes approximately 4-6 μs). *Baseline-Interrupted-Agressive (BA(IA)):* Similar to the previous case, this time the timer interrupt is set at every 10ms. *IDA (baseline) (IDA (BA)):* Total runtime when the original application is interrupted to compute the attestation. *IDA-Interuppted (IDA (IN)):* Overall runtime when timer interrupt (500ms) is also enabled. *IDA-Interrupted-Aggressive (IDA (IA)):* Interrupt is set to every 10ms. Similar cases are considered for IDA+.

Results for this experiment are shown in Fig. 8 and Table III in Appendix C. In the baseline setup, interrupts have minimal impact as their duration is relatively short (tens of microseconds). The noteworthy observation is the runtime difference between IDA (IN) vs. IDA+ (IN) and IDA (IA) vs. IDA+ (IA). In the former case, the runtime is nearly identical for both (about 12% overhead compared to the baseline due to the time it takes to attest 8 KB of memory, i.e., approximately 488 ms). However, in the latter case, IDA experiences a notable

[6]MiBench is a popular, free, commercially representative embedded benchmark suite.

increase in runtime, while IDA+'s runtime remains unchanged. The reason behind this difference lies in the interval between two consecutive interrupts for IA, which is much shorter than the time needed to complete the attestation (i.e., 10ms vs. 488ms). As per Algorithm 2, this causes the attestation to reset, and it keeps failing until the watchdog timer aborts the request (in this scenario, we set the timer to twenty attempts). On the other hand, for IDA+, this issue doesn't arise, as the attestation can continue after the interrupt. This ensures that performance remains unaffected by frequent interrupts, as the attestation can be completed normally in between them.

Power Overhead. Results for estimated power consumption are reported in Table II (see Appendix B). Overall, we observe minimal changes to power consumption in most designs. Regarding energy consumption, the results will be affected by the additional runtime overhead for each method. In our setup, we note that in the worst case (i.e., when interrupt intervals is $\approx$ attestation time), the energy overhead for IDA could increase by as much as 2.5x compared to other methods due to the waste caused by resets forced by frequent interrupts. We further analyze the impact of supporting interrupts in IDA+ and also report the result for VRASED [44] and RATA [26]. Results are shown in Fig. 9. As can be seen in the figure, by eliminating the need for resets while supporting interrupts, even in the worst case IDA+ achieves very similar energy consumption to that of VRASED and RATA, confirming its usefulness, capability, and feasibility.

Comparisons. We finish this section by summarizing the comparisons among different approaches. Overall, IDA achieves very similar hardware overhead compared to the state-of-the-art with better runtime while providing additional protection and features. However, it suffers from potential persistent interrupts. Further, IDA+ solves this problem by eliminating the need for resetting. It can achieve similar power, area, and runtime overhead results compared to the state-of-the-art while adding another security protection (TOCTOU) and supporting interrupts. *Together, these results confirm that IDA and IDA+ are excellent candidates for achieving remote attestation capability in low-end devices.*

Further, while in this paper our primary focus lies on remote attestation, we acknowledge that this doesn't cover all attack vectors, including dynamic attacks. Remote attestation and memory integrity, however, remain crucial issues, with several concerns addressed in the paper, such as support for interrupts, DMA requests, and TOCTOU. We also believe that the proposed support for TOCTOU could enhance the applicability of our approach, transforming IDA from a purely passive method to a dynamic one, constantly monitoring memory modifications to detect any changes at ANY time.

Though not presented in this paper, IDA and IDA+ can be utilized for applications directly extending memory integrity. These applications encompass proof of execution, secure reset, secure code update, and secure data sensing and/or actuation (e.g., verifying the integrity of sending/receiving commands remotely). Their implementation is left for future work.

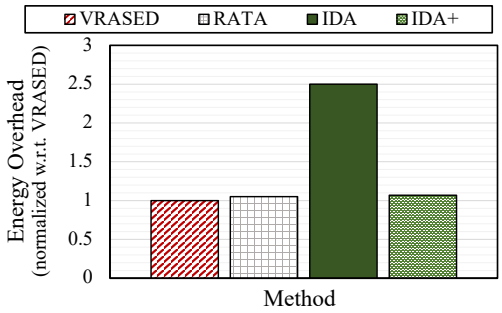

Fig. 9. Energy overhead for IDA, IDA+, and other state-of-the-art hybrid attestation techniques. Results are normalized w.r.t. VRASED [44]. IDA and IDA+ can support interrupt while others cannot.

VIII. Related Work

Throughout the paper, and specifically, in Section II, we extensively discussed various remote attestation methods including software, hardware, and hybrid methods. We explained the advantages and disadvantages of each approach. We also discussed the contributions of our work compared to state-of-the-art (e.g., see Table I). Here we review other methods for achieving trust remotely. We also discuss other important and relatively relevant topics to RA.

An important class of solutions for RA is software-based memory isolation techniques to isolate the device's root of trust software from the rest of other untrusted software modules running in the same address space. Examples are Sμv [7] and PISTIS [33]. The key idea is to use software-only isolation established from the boot. Particularly, they initially deploy an initial code, called a Trusted Computing Module (TCM), that occupies part of the Flash memory including the bootloader area. The main goal of the TCM is to guarantee memory protection by isolating its memory area from other memory parts, creating two logically isolated memory zones: secure and insecure. To execute applications, a customized compiler is needed to recompile a given application such that the created binary correctly follows the expected memory format. Achieving this requires a secure compiler.

The key advantage of such an approach is eliminating the need for hardware support (i.e., similar to SWATT) while being more secure than SWATT approaches. The downside, however, is an inevitable runtime overhead due to the software isolation, and additional memory overhead.

A related approach is methods based on compartmentalization and overlays [21], [22], [61]. These methods are based on pure software and compiler co-design solutions to achieve isolation and security by design. Generally, they share very similar advantages and disadvantages compared to software-based isolation mechanisms.

Also related to remote attestation are methods for control flow and data integrity attestation [1], [2], [27], [28], [40], [46], [56], [59]. Unlike RA, these techniques aim for precise attestation of the execution path of an application running on an embedded device. This is achieved by measuring the program's execution path at the binary level, capturing its

runtime behavior. The goal is to ensure security during the runtime and protect the application, and system, against dynamic attacks such as buffer overflow, ROP, etc. Similarly, the same approaches could be applied to ensure data integrity during runtime. Compared to RA, they provide stronger security guarantees at the expense of requiring more hardware support and other mechanisms to track the program during its execution.

Lastly, another relevant category of work to this paper is methods that leverage PUFs for authentication and attestation [19], [29], [37], [48], [49], [55]. The closest are methods that leveraged PUFs for software remote attestation (SWATT) [37], [49]. The main idea of these methods is to leverage PUFs to create unique random values during the software attestation process.

IX. Conclusions

In this paper, we proposed a new hybrid attestation method called IDA, which removes the requirement for disabling interrupts and restrictive access control for the secret code and attestation code. We showed how removing such requirements could improve the system's security and flexibility.

The key insight was to verify the attestation process via a trusted hardware module instead of directly enforcing it by disabling interrupts and leveraging a secret key. Further, we presented IDA+ which allows us to track program memory changes even after attestation is done. We showed how such a feature could be used to detect TOCTOU attacks and handle interrupts.

We analyzed the security of our approach and evaluated its effectiveness. Comparing IDA and IDA+ with the state-of-the-art, we showed that our approach has minimal hardware and runtime overhead while achieving important properties such as support for interrupts and DMA requests, and detecting TOCTOU attacks.

Acknowledgement

We thank our reviewers for their guidance. This work has been supported, in part, by NSF grants CNS-2211301, CNS-2303115, and CNS-2312089. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[2] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.

[3] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. {GAROTA}: Generalized active {Root-Of-Trust} architecture (for tiny embedded devices). In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2243–2260, 2022.

[4] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*, pages 1362–1380. IEEE, 2019.

[5] Muhammad Naveed Aman, Mohamed Haroon Basheer, Siddhant Dash, Jun Wen Wong, Jia Xu, Hoon Wei Lim, and Biplab Sikdar. Hatt: Hybrid remote attestation for the internet of things with high availability. *IEEE Internet of Things Journal*, 7(8):7220–7233, 2020.

[6] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. Sana: secure and scalable aggregate network attestation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 731–742, 2016.

[7] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. S\μ v—the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[8] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[9] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[10] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Cure: A security architecture with customizable and resilient enclaves. In *USENIX Security Symposium*, pages 1073–1090, 2021.

[11] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[12] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1638–1655. IEEE, 2022.

[13] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd annual design automation conference*, pages 1–6, 2015.

[14] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[15] Xavier Carpent, Gene Tsudik, and Norrathep Rattanavipanon. Erasmus: Efficient remote attestation via self-measurement for unattended settings. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1191–1194. IEEE, 2018.

[16] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, 2009.

[17] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. Asap: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 721–726, 2022.

[18] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. Program analysis of commodity iot applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys (CSUR)*, 52(4):1–30, 2019.

[19] Urbi Chatterjee, Vidya Govindan, Rajat Sadhukhan, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, Debashis Mahata, and Mukesh M Prabhu. Building puf based authentication and key exchange protocol for iot without explicit crps in verifier database. *IEEE transactions on dependable and secure computing*, 16(3):424–437, 2018.

[20] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[21] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *USENIX Security Symposium*, volume 2018, pages 65–82, 2018.

[22] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303. IEEE, 2017.

[23] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[24] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[25] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[26] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the toctou problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[27] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[28] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[29] Mohammad Ebrahimabadi, Mohamed Younis, and Naghmeh Karimi. A puf-based modeling-attack resilient authentication protocol for iot devices. *IEEE Internet of Things Journal*, 9(5):3684–3703, 2021.

[30] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.

[31] Justin Feng, Timothy Jacques, Omid Abari, and Nader Sehatbakhsh. Everything has its bad side and good side: Turning processors to low overhead radios using side-channels. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, pages 288–301, 2023.

[32] Olivier Girard. openmsp430, 2018, https://github.com/olgirard/openmsp430.

[33] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. {PISTIS}: Trusted computing architecture for low-end embedded systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3843–3860, 2022.

[34] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[35] HACL-star. Hacl*, a formally verified cryptographic library written in f*, 2023, https://github.com/hacl-star/hacl-star.

[36] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. Seed: secure non-interactive attestation for embedded devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 64–74, 2017.

[37] Joonho Kong, Farinaz Koushanfar, Praveen K Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. Pufatt: Embedded platform attestation based on novel processor-based pufs. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[38] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[39] Yanlin Li, Jonathan M McCune, and Adrian Perrig. Viper: Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16, 2011.

[40] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[41] Arsalan Mosenia and Niraj K Jha. A comprehensive study of security of internet-of-things. *IEEE Transactions on emerging topics in computing*, 5(4):586–602, 2016.

[42] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, pages 479–494, 2013.

[43] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–33, 2017.

[44] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. Vrased: A verified hardware/software co-design for remote attestation. In *USENIX Security Symposium*, pages 1429–1446, 2019.

[45] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Apex: A verified architecture for proofs of execution on remote devices under full software compromise. In *USENIX Security Symposium*, pages 771–788, 2020.

[46] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Dialed: Data integrity attestation for low-end embedded devices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 313–318. IEEE, 2021.

[47] OWASP. Password storage cheat sheet, 2023, https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[48] Mahmood Azhar Qureshi and Arslan Munir. Puf-rake: A puf-based robust and lightweight authentication and key establishment protocol. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2457–2475, 2021.

[49] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. Short paper: Lightweight remote attestation using physical functions. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 109–114, 2011.

[50] Nader Sehatbakhsh, Alireza Nazari, Haider Khan, Alenka Zajic, and Milos Prvulovic. Emma: Hardware/software attestation framework for embedded systems using electromagnetic signals. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 983–995, 2019.

[51] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, 2006.

[52] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282. IEEE, 2004.

[53] Saleh Soltan, Prateek Mittal, and H Vincent Poor. Blackiot: Iot botnet of high wattage devices can disrupt the power grid. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 15–32, 2018.

[54] Joachim Strömbergson, Olof Kindgren, and Sanjay A Menon. sha256, 2023, https://github.com/secworks/sha256.

[55] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual design automation conference*, pages 9–14, 2007.

[56] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[57] Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. Realswatt: Remote software-based attestation for embedded devices under realtime constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2890–2905, 2021.

[58] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[59] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *USENIX Security Symposium*, pages 1805–1821, 2019.

[60] Yuchen Yang, Longfei Wu, Guisheng Yin, Lijie Li, and Hongbin Zhao. A survey on security and privacy issues in internet-of-things. *IEEE Internet of things Journal*, 4(5):1250–1258, 2017.

[61] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren. Opec: operation-based security isolation for bare-metal embedded systems. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 317–333, 2022.

APPENDIX

A. Attestation Algorithm

Remote attestation is typically realized as a challenge-response protocol between a verifier and a prover. The definition shows the attestation protocol in more detail.

> ***Definition** 1 (Attestation Protocol):* Remote attestation (RA) is a tuple (Request, Attest, Verify) of algorithms between a prover (Prv) and a verifier (Vrf):
>
> - Request: algorithm initiated by Vrf to request a measurement of Prv memory range AR (attested range). As part of Request, Vrf sends a challenge Chal to Prv.
> - Attest: algorithm executed by Prv upon receiving Chal from Vrf. Computes an authenticated integrity-ensuring function over AR content. It produces attestation token H, which is returned to Vrf, possibly accompanied by auxiliary information to be used by the Verify algorithm (see below).
> - Verify: algorithm executed by Vrf upon receiving H from Prv. It verifies whether Prv current AR content corresponds to some expected value M (or one of a set of expected values). To compute M, PRV only needs the current content of AR. Verify outputs: 1 if H is valid, and 0 otherwise.

To acquire the ground truth for Verify, we assume the verifier possesses applications/binaries that existed on the remote device. It is worth noting that the target application is single-core bare-metal IoT devices, usually with a few applications installed. Real systems exemplifying this model encompass health monitoring devices, industrial IoTs, drones, robotic arms, and more. The application is pre-loaded in these scenarios, and the remote verifier intermittently checks the device's integrity.

B. Detailed Power, Attestation Time, and Area Results

The raw numbers for power, attestation runtime (interrupt-free), and area (total number of logic cells) are reported in Table II.

TABLE II
POWER, RUNTIME (ATTESTATION PER 1 KB), AND AREA (LOGIC CELLS) NUMBERS FOR DIFFERENT HYBRID ATTESTATION TECHNIQUES.

	Power	Attestation Time	Area
Baseline	25.21 mW	N/A	2891
VRASED	25.98 mW	110.9 ms	3269
RATA	26.03 mW	110.9 ms	3301
SANCUS	26.96 mW	41.0 ms	5174
IDA	26.05 mW	61.0 ms	3440
IDA+	26.06 mW	61.8 ms	3506

C. Runtime Overhead Results for MiBench

To study the impact of interrupts on the overall runtime and how IDA+ can improve the overall performance, we report our results for three standard benchmark applications under three different configurations (baseline with no interrupts, occasional interrupts, and frequent interrupts). The chosen applications signify typical tasks for embedded systems (e.g., signal processing, networking, basic mathematical operations). The experiment is repeated for IDA and IDA+.

TABLE III
RUNTIME FOR THREE DIFFERENT APPLICATIONS FROM MIBENCH UNDER THREE DIFFERENT SETTINGS: BASELINE = BA, BASELINE-INTERRUPTED = BA(IN) (500MS), BASELINE-INTERRUPTED-AGGRESSIVE= BA(IA) (10 MS). THE EXPERIMENT IS REPEATED FOR IDA AND IDA+ WITH SIMILAR SETTINGS.

	Basicmath	Dijkstra	FFT
BA	15.6 s	4.25 s	2.31 s
BA(IN)	15.6 s	4.25 s	2.31
BA(IA)	15.606 s	4.252 s	2.31 s
IDA (BA)	16.088 s	4.738 s	2.798 s
IDA (IN)	16.088 s	4.739 s	2.799 s
IDA (IA)	16.28 s	4.938 s	2.998 s
IDA+ (BA)	16.09 s	4.738 s	2.8s
IDA+ (IN)	16.09 s	4.738 s	2.8s
IDA+ (IA)	16.091 s	4.738 s	2.81s