Toward Remotely Verifiable Software Integrity in Resource-Constrained IoT Devices

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik

The authors provide a holistic and systematic treatment of this family of architectures.

Abstract

Lower-end IoT devices typically have strict cost constraints that rule out usual security mechanisms available in general-purpose computers or higher-end devices. To secure low-end devices, various low-cost security architectures have been proposed for remote verification of their software state via integrity proofs. These proofs vary in terms of expressiveness, with simpler ones confirming correct binary presence, while more expressive ones support verification of arbitrary code execution. This article provides a holistic and systematic treatment of this family of architectures. It also compares (qualitatively and quantitatively) the types of software integrity proofs, respective architectural support, and associated costs. Finally, we outline some research directions and emerging challenges.

Introduction

Micro-Controller Units (MCUs) that perform actuation and/or sensing are the *de facto* interfaces between the analog and digital worlds. On actuators, digital commands are converted into physical actions, while sensors convert analog ambient quantities into digital form. They represent the point where data is "born" and first processed. At the same time, since MCUs are programmable, their software can be compromised and subsequently corrupt data, e.g., by modifying software to forge/spoof a sensed values or "lie" about having performed actuation commands. One naïve defense strategy is to make all software non-writable, e.g. by housing it in ROM. While this obviates software compromise, it also precludes legitimate software updates.

In the last decade, the research community has actively identified and examined this issue [1]. Earlier results proposed methods to allow a trusted party called *Verifier* to remotely check if the correct binary is currently installed on a remote and untrusted *Prover*. This security service is commonly known as Remote Attestation ($\mathcal{R}$A) [2–5].

A related notion is Proofs of Execution (PoX) [6] which extends $\mathcal{R}$A to prove correct execution of the attested binary or parts thereof, i.e., functions within the binary. In line with PoX, Control Flow Attestation ($\mathcal{C}$FA) allows *Verifier* to also verify the sequence of executed instructions. This detects software exploits that corrupt execution path by changing the program control flow without modifying the actual binary (aka code-reuse attacks [7]). Data-Flow Attestation ($\mathcal{D}$FA) further extends $\mathcal{C}$FA with detection of data-only attacks that exploit vulnerabilities to corrupt data without modifying the program control flow.

This article overviews a series of recent low-cost techniques, based on HW/SW co-design, that create unforgeable proofs of software integrity (encompassing aforementioned services) for the MCUs commonly used in low-end IoT devices. Each of these technique tackles one of the following questions:

1. How to prove that an MCU of interest is currently installed with the correct software/firmware binary?
2. How to extend this proof with historical context, i.e., how to determine "since when" the expected software has been installed on the device?
3. Upon receiving a result from a remote MCU (e.g., a sensed value), how to ensure that it was indeed obtained through the proper execution of expected software on the expected device?
4. Can we verify that instructions were executed in the intended/legal order? In other words, how to ensure the absence of control flow attacks during the execution?
5. In addition, how to ensure the absence of (non-control) data-only attacks during execution?

Naturally, approaches that provide more expressive evidence (thereby detecting stealthier attacks) also incur higher hardware and run-time overhead.

Folklorically and historically, the common wisdom holds that the typical/usual low-end MCUs (such as TI MSP430 or AVR ATMega) are incapable of supporting software integrity verification. While this is true for unmodified MCUs, recent research results show that it can indeed be achieved with minimal hardware overhead, low overall cost, and strong security guarantees. This article overviews and compares (qualitatively and quantitatively) a sequence of five techniques, each incrementally addressing one of the above questions. We also identify several outstanding challenges that need to be tackled by future work.

Background

Resource-Constrained/Low-End MCUs

This article focuses on resource-constrained embedded/smart/IoT sensors and actuators (or hybrids thereof). These are some of the simplest and

Ivan De Oliveira Nunes is with Rochester Institute of Technology, USA; Sashidhar Jakkamsetti is with Robert Bosch LLC, USA; Norrathep Rattanavipanon (corresponding author) is with College of Computing, Prince of Songkla University, Phuket, Thailand; Gene Tsudik is with the University of California Irvine, USA.

Digital Object Identifier: 10.1109/MCOM.001.2300514

0163-6804/24/$25.00 © 2024 IEEE

smallest computing devices, based on low-power single-core MCUs with only a few KBytes of memory. Figure 1 illustrates a typical MCU architecture, featuring a CPU core, an Interrupt Control Logic module, and a Direct Memory Access (DMA) controller connected to main memory via a bus. The MCU includes four memory types:

1. Program memory (PMEM)
2. Read-only memory (ROM)
3. Data memory (DMEM)
4. Peripheral memory

Application software is stored in PMEM, usually realized as non-volatile physical memory such as Flash or FRAM. Runtime data is stored in volatile DMEM, implemented using RAM. ROM contains the bootloader and any software fixed at the time of manufacturing or provisioning, and remains immutable thereafter. DMA can read and write memory in parallel with the core. Generally, low-end MCUs run software atop "bare metal," i.e., execute software directly from PMEM, without relying on memory management units (MMU) and often not even memory protection units (MPU). Examples of such MCUs include Atmel AVR ATmega, TI MSP430, and ARM Cortex-M, featuring 8/16/32-bit single-core CPUs, running at clock frequencies of 1–48MHz, with up to 128 KB of addressable memory.

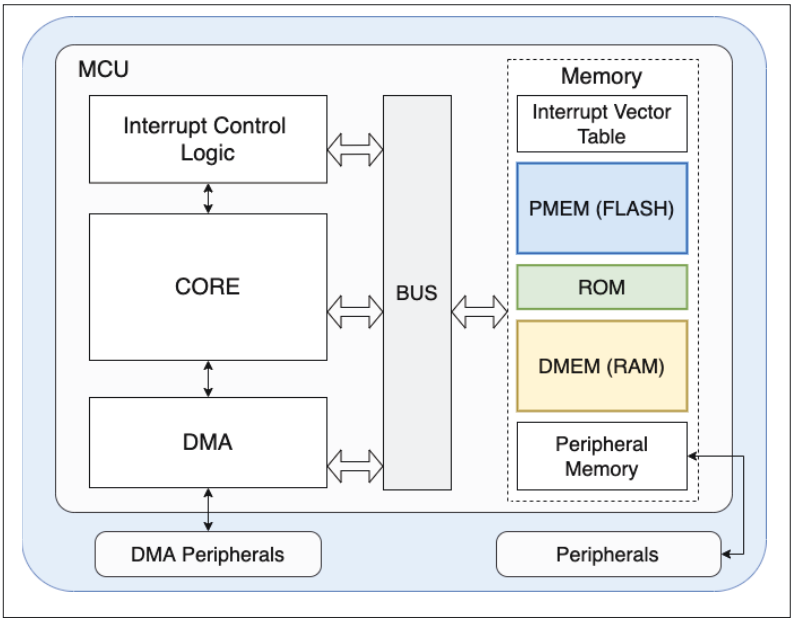

FIGURE 1. Architecture of low-end IoT devices.

Attack Vectors & Threat Model

To reason about software integrity one must consider that all modifiable memory could be tampered with by the adversary ($\mathcal{A}$dv), unless explicitly protected by the hardware architecture. Therefore, $\mathcal{A}$dv is assumed to control the entire software state of *Prover*. This allows $\mathcal{A}$dv to read and write any memory that is not explicitly protected by hardware, program DMA controllers, and trigger interrupts at any given moment.

While $\mathcal{A}$dv may reprogram *Prover* software in PMEM via a wired interface (e.g., USB or J-TAG), we consider invasive attacks physically altering hardware and ROM code to be out of scope. Protection against these attacks can be obtained through orthogonal tamper-resistance techniques, e.g., employing internal power regulators, implementing anomaly detection for the MCU's behavior, enclosing the MCU with additional metal layers or enforcing physical access control to the devices.

Verifying Software/Firmware Integrity with $\mathcal{R}$A

Completely preventing illegal code modifications in low-end devices is a challenging task due to the need to perform code updates. As an alternative, $\mathcal{R}$A is an inexpensive and effective technique which detects attacks that modify *Prover* code. $\mathcal{R}$A allows *Verifier* to remotely assess software integrity of *Prover*, in an on-demand fashion. This is typically realized as a *Verifier*-initiated challenge-response protocol where:

1. *Verifier* sends an attestation request containing a cryptographic challenge to *Prover*.
2. *Prover* performs an *authenticated integrity check* based on the received challenge over its own PMEM.
3. *Prover* returns the result to *Verifier*.
4. *Verifier* validates whether the result matches a valid PMEM state by comparing it to the expected (benign) value.

The purpose of the challenge in step 1 is to ensure that *Prover*'s response is fresh, i.e., reflects its current software state (rather than an old replayed response). The *authenticated integrity check* in step 2 can be implemented as a Message Authentication Code (MAC) or signature. We refer to the function that implements the authenticated integrity check as the integrity-ensuring function (IEF). To implement IEF, *Prover* must maintain a secret key ($\mathcal{K}$), confidentiality of which must be preserved even if *Prover* software is compromised. Consequently, main challenges in designing secure $\mathcal{R}$A revolve around:

- Secure storage of $\mathcal{K}$
- Establishment of an immutable secure runtime environment that accesses $\mathcal{K}$ to compute the IEF without leaking $\mathcal{K}$ to any other software in the *Prover*.

SANCUS [8] protects the IEF and $\mathcal{K}$ by implementing them entirely in hardware inaccessible to untrusted software. This eliminates the need for any software component of the trusted computing base (TCB). However, this approach incurs a significant hardware cost, which may be prohibitive for budget-coscious low-end MCUs. SMART [2] is designed to minimize hardware overhead by using a hybrid (HW/SW co-design) $\mathcal{R}$A approach. SMART implements its IEF in software, while a small amount of hardware is used to detect any violation that attempts to leak $\mathcal{K}$ or tamper with IEF execution. The main trade-off between SMART and SANCUS is speed *vs.* cost — being all hardware, the latter is faster, while the former is cheaper.

Building upon SMART design principles in a less *ad hoc* fashion, recently proposed *VRASED* [3] technique is a formally verified hardware/software $\mathcal{R}$A co-design. Figure 2 shows its hardware and software components. The trusted software module (`SW-Att`) contains IEF code and $\mathcal{K}$ stored in ROM. This way, neither `SW-Att` nor $\mathcal{K}$ values can be modified after manufacturing or provisioning. *VRASED* also includes a hardware monitor that tracks several MCU signals to determine:

1. *PC* value, i.e., address of currently executing instruction.

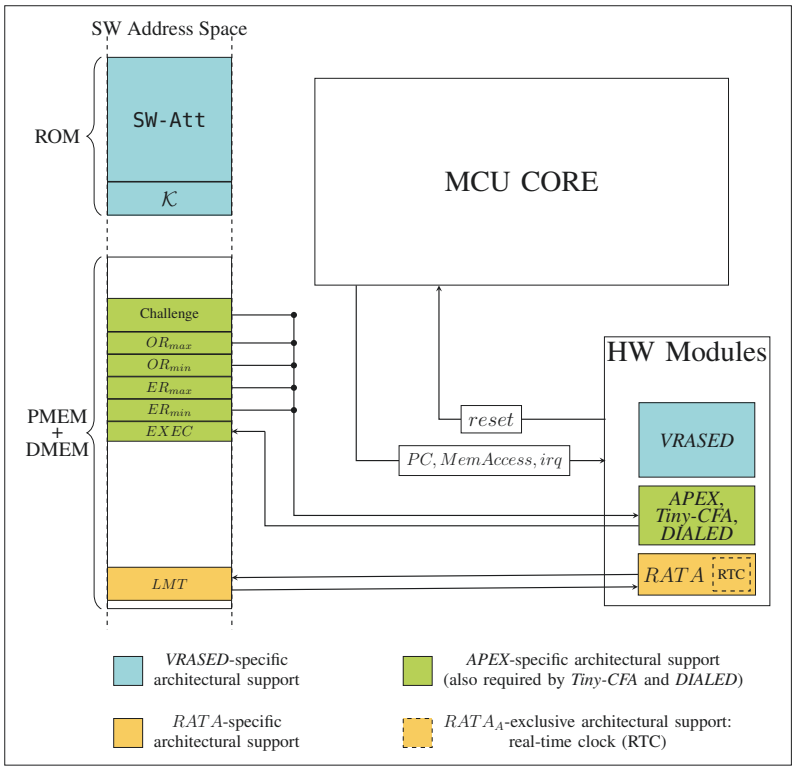

FIGURE 2. Architectural requirements of various integrity proofs.

2. *MemAccess*, memory address currently being read or written by either MCU Core or DMA.
3. *irq*, a one-bit signal indicating whether an interrupt is currently being triggered.

Using these signals, *VRASED* hardware monitor detects violations that try to violate secrecy of $\mathcal{K}$ or `SW-Att` execution integrity through:

- Illegal accesses to $\mathcal{K}$ by any software other than `SW-Att`.
- Any incomplete or interrupted `SW-Att` execution that could lead to forgery of an attestation result.

Upon detecting a violation, *VRASED* triggers an immediate MCU *reset*, promptly preventing the violation.

To facilitate formal verification, *VRASED* avoids state explosion problems by structuring its implementation as a collection of sub-modules, each guaranteeing a specific set of formal sub-properties. Each sub-module undergoes individual verification, and the combination of all sub-modules is then verified for end-to-end notions of $\mathcal{R}$A soundness and security. Informally, $\mathcal{R}$A soundness ensures correct IEF computation over current PMEM, while $\mathcal{R}$A security guarantees that IEF execution produces an unforgeable authenticated PMEM measurement and prevents $\mathcal{K}$ leakage before, during, or after $\mathcal{R}$A. Further details on *VRASED* formal verification can be found in [3].

TOCTOU-Security & Efficient $\mathcal{R}$A

Recall that each $\mathcal{R}$A instance is initiated by *Verifier*. An authentic $\mathcal{R}$A result received from *Prover* reflects *Prover* code *only at the time when it is computed*. In particular, it provides no information about *Prover* software *before* $\mathcal{R}$A execution or between successive $\mathcal{R}$A instances. This issue is commonly termed as Time-of-Check Time-of-Use (TOCTOU). In the context of $\mathcal{R}$A, TOCTOU means that transient malware can not be detected by $\mathcal{R}$A. Concretely, if malware infects *Prover*, performs its malicious tasks, and erases itself prior to the next $\mathcal{R}$A instance, its ephemeral presence would remain unnoticed.

RATA [9] is another recent technique to address the TOCTOU problem. It extends *VRASED* with a minimal and formally verified hardware component that additionally provides historical context about the state of PMEM. RATA consists of two alternative designs. Shown as yellow components in Fig. 2, the first version — $RATA_A$ — is a verified hardware module that operates as follows:

- It monitors MCU *PC* and *MemAccess* signals and uses this information to detect whether PMEM is currently being modified.
- When a PMEM modification is detected, $RATA_A$ retrieves the current time from a real-time clock (RTC) and stores it in a designated and secure memory area, called the Latest Modification Time (LMT) region.
- LMT region is always covered by IEF (i.e., included in each attestation result, along with PMEM) and read-only to all software and DMA.

To verify the attestation result, *Verifier* compares the received LMT value with the time of the last authorized PMEM modification (usually, time of latest legitimate code update) to check whether any unauthorized activity occurred since then.

In practice, RTCs are usually unavailable on resource-constrained devices and secure clock synchronization in distributed systems poses a significant challenge, particularly for such devices. $RATA_A$ is thus useful only to demonstrate the general approach. To make it practical, the second version ($RATA_B$) eliminates the RTC requirement. $RATA_B$ relies on *Verifier*'s own notion of time by associating each attestation challenge to the time of its issuance by *Verifier*. *Prover* logs the latest received challenge to LMT. For this to work, each *Verifier*'s challenge must be unique for each $\mathcal{R}$A instance, to prevent replay attacks. This allows $RATA_B$ to uniquely associate each challenge to *Verifier*'s notion of time. $RATA_B$ hardware component is similar $RATA_A$, except that LMT is now updated with the current challenge if and only if a PMEM modification occurred since the previous $\mathcal{R}$A instance.

An important side benefit of *RATA* is its ability to significantly reduce $\mathcal{R}$A execution time on *Prover* since it is no longer necessary to compute the IEF over the entire PMEM most of the time. Assuming that *Verifier* already knows PMEM contents from a previous $\mathcal{R}$A result, it is adequate to demonstrate that no changes have occurred since then. This can be achieved by attesting only LMT as opposed to the entire PMEM, resulting in a substantial reduction of computation time. In fact, this time is constant in size of LMT, instead of increasing linearly with PMEM size. For instance, on an MSP430 MCU running at 8MHz with 8 kBytes PMEM, *RATA* takes roughly 50ms on *Prover*, as opposed to 1sec in VRASED. (See[9] for details).

From $\mathcal{R}$A to Proofs of Execution

$\mathcal{R}$A yields an indication of whether *Prover* PMEM contains expected code. However, it does not guarantee that any operation (i.e., sensing/actuation functions in that code) was executed correctly. Also, it does not bind results (e.g., sensor

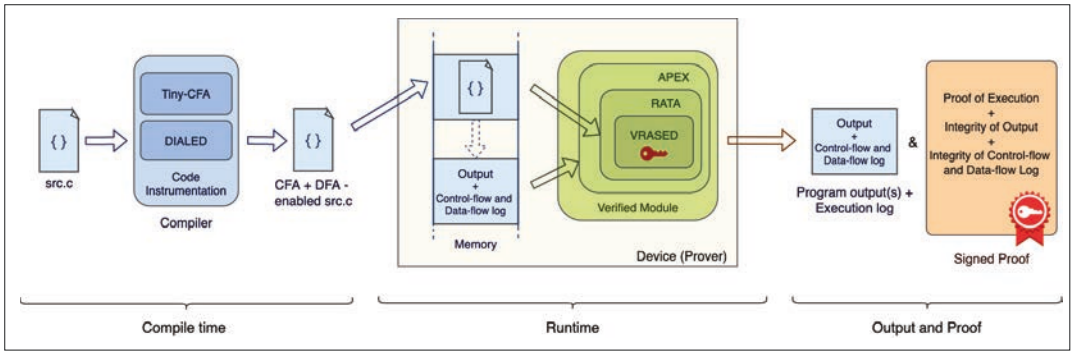

FIGURE 3. Phases in embedded software execution integrity.

This can be accomplished by securely recording the destination of every control flow altering instruction, such as jumps, branches, and returns, during the program's execution.

readings) to the correct execution of appropriate code. In other words, $\mathcal{R}$A gives no secure association between data received by *Verifier* from the *Prover* and *Prover*'s execution of a specific application-dependent operation. Thus, $\mathcal{A}$dv can tamper with or spoof data even if *Prover* PMEM contains correct code.

For this reason, *APEX* [6] proposes the concept of "Proofs of Execution" (PoX) by augmenting $\mathcal{R}$A to prove to *Verifier* that:

1. The function of interest exists within a specific region of PMEM;
2. This function was indeed executed in a timely manner, upon *Verifier*'s request; and
3. Any claimed output was indeed produced by this timely execution of the desired function on *Prover*.

Hence, PoX enables authentication of data from its "birth," i.e., at the point when it becomes digital, through the interaction of code with sensing ports (e.g., general purpose I/O).

APEX [6] structure is shown in green in Fig. 2. It is built atop a secure $\mathcal{R}$A technique, such as *VRASED*, by introducing an additional hardware module that controls a 1-bit flag, called *EXEC*, that can not be modified by any software. The key is to use the high value of *EXEC* to inform *Verifier* that the intended portion of attested code (a *Verifier*-defined code section in PMEM) was executed successfully between the time when *Verifier* challenge was issued and the time when IEF on *Prover* executed. Similarly, a value of 0 for *EXEC* indicates that execution of that code section did not occur, or that it was tampered with.

APEX IEF covers: *EXEC* flag itself; the region where output must be saved (called output region or *OR*); and code stored in a *Verifier*-defined section of PMEM (called executable region or *ER*). Security of the $\mathcal{R}$A architecture guarantees the contents of these memory regions (including *EXEC*) can not be spoofed. Therefore, as long as *APEX* hardware properly controls *EXEC*, the $\mathcal{R}$A result constitutes unforgeable proof that code in *ER* was executed and produced the results stored in *OR*. *APEX* considers that code section executed properly (setting *EXEC* to 1) if and only if:

1. Execution of *ER*-resident code is atomic (i.e., uninterrupted), from *ER*'s first, to its last, instruction.
2. Neither code in *ER*, nor its output in *OR* is modified between execution and the next IEF computation.
3. During execution, DMEM is not modified by DMA or by other software functions except *ER*.

These conditions ensures that when *EXEC* is 1, both *ER* and *OR* remain consistent between *ER* code execution and subsequent IEF computation, plus execution itself is not tampered with. *ER* and *OR* locations and sizes are configured using values of ER_{min}, ER_{max}, OR_{min} and OR_{max} in DMEM, as shown in Fig. 2. This allows *APEX* to support PoX of arbitrary code and output sizes.

APEX hardware is verified to conform to formal specifications of the abovementioned *EXEC* behavior. These specifications, along with the underlying verified guarantees of *VRASED*, are proven to guarantee a security definition for unforgeable PoX, as discussed in [6].

$\mathcal{C}$FA on Resource-Constrained MCUs: Augmenting PoX to Verify Control Flow Paths

PoX assumes that the code for which execution is being proven (i.e., the code in *ER*) is free of memory-safety vulnerabilities, such as those leading to buffer overflows and similar attacks. However, when these vulnerabilities (unintentionally) exist in the executable, they can be exploited at the time of execution to launch well-known control flow attacks (such as return- and jump-oriented programming) that change the order in which the instructions are executed to cause unintended behavior, without modifying the program's code. As a consequence, these attacks would remain oblivious to $\mathcal{R}$A or PoX.

Control Flow Attestation ($\mathcal{C}$FA) aims to detect control flow attacks by also providing *Verifier* with a report that shows the exact order in which the instructions that form a software operation of interest have executed on *Prover*. This can be accomplished by securely recording the destination of every control flow altering instruction, such as jumps, branches, and returns, during the program's execution.

A number of $\mathcal{C}$FA techniques have been proposed in recent years (e.g., C-FLAT [10] and LiteHAX [11]). However, they target higher-end embedded devices (e.g., those featuring application CPUs, such as Raspberry Pi). Unfortunately, these techniques are prohibitively expensive for resource-constrained MCUs.

Tiny-CFA [12] was recently developed to address the $\mathcal{C}$FA problem in the context of resource-constrained MCUs by leveraging inexpensive PoX as its only hardware requirement. As shown in Fig. 3, *Tiny-CFA* introduces an additional compilation-time phase where the code to be executed by *Prover* is instrumented with addition-

Scheme	Baseline MCU	*VRASED* [3]	*APEX* [6]	*Tiny-CFA* [12]	*DIALED* [13]	*RATA* [9]
Detection of Modified Code	×	✓	✓	✓	✓	✓
Provable Execution	×	×	✓	✓	✓	×
Detection of Control Flow Attack	×	×	×	✓	✓	×
Detection of Data Flow Attack	×	×	×	×	✓	×
TOCTOU Security	×	×	×	×	×	✓

TABLE 1. Qualitative comparison.

al instructions that generate a log (referred to as *CF-Log*), containing the control flow path taken during execution.

During a PoX of the instrumented code, execution yields *CF-Log*, in addition to its regular result/output. *Tiny-CFA* ensures authentication and integrity of *CF-Log* by making *CF-Log* a part of the PoX output, which is located within *APEX*'s output region *OR* and covered by the IEF. As a result, *Verifier* can use this new evidence (*CF-Log*) to determine the validity of the execution control flow path and verify the absence of control flow hijacking attacks.

In more detail, *Tiny-CFA* instruments the executable to ensure that *CF-Log* contains all information required by *Verifier* to reconstruct the control flow path by:

- *Securely logging control flow instructions*: all control flow altering instructions are prepended with additional instructions to log their destinations to *CF-Log*.
- *Ensuring append-only CF-Log*: direct writes to *CF-Log* are replaced at compile time while indirect writes are instrumented to check whether their destination is within *CF-Log* at runtime. Upon detecting an illegal write to *CF-Log*, the PoX is halted, implying an invalid control flow.

Due to the resource-constrained nature of MCUs, $\mathcal{C}$FA schemes should have minimal hardware and runtime overheads. *Tiny-CFA* minimizes hardware requirements by requiring no hardware support other than PoX from *APEX*. It also implements several optimizations to keep the runtime overhead and *CF-Log* size within practical limits. We discuss these overheads later and revisit opportunities for future work on reducing $\mathcal{C}$FA runtime costs.

MCU Data Flow Integrity atop $\mathcal{C}$FA

Aside from control flow attacks (detected by $\mathcal{C}$FA), stealthier attacks known as "data-only" attacks can still originate from memory safety vulnerabilities. Specific vulnerabilities (see example in [13]) allow attacks to corrupt intermediate data variables in DMEM without even altering the control flow of the program (hence "data-only").

Detection of such data-only attacks still remains elusive and requires verifying the data-flow integrity during execution — a service known as Data-Flow Attestation ($\mathcal{D}$FA). Prior work in $\mathcal{D}$FA such as OAT [14] requires user annotations and relatively expensive trusted hardware support.

DIALED [13] presents the first $\mathcal{D}$FA architecture aimed at resource-constrained MCUs by following an approach similar to *Tiny-CFA*. As shown in Fig. 3, at compile time, *DIALED* uses *Tiny-CFA* for $\mathcal{C}$FA-related instrumentation. Additionally, it adds its own instrumentation to log all data inputs to a dedicated memory region, called I-Log. *DIALED*'s instrumenter defines any non-local variables as data inputs, i.e., any value located *outside* of the attested program's current stack.

Following this definition, any instructions that access data from arguments, peripherals, network, or general-purpose I/O are considered data inputs and recorded to I-Log. Conversely, reads occurring during regular computation, e.g., instructions that make use of local variables are excluded from I-Log, as they are not inputs to this program. This approach helps keep the size of I-Log relatively small.

Recall that *Tiny-CFA* instruments the executable to produce *CF-Log*. In the context of *DIALED*, both *CF-Log* and I-Log are included in *APEX*'s authenticated output region *OR*. As *OR* is covered by the IEF, *Verifier* is assured of the integrity of these logs. With the code, its execution's control flow path, and all inputs, *Verifier* can locally emulate execution and its data flow. Therefore, it can verify all steps in this computation, and detect data-only and control flow attacks.

As illustrated in Fig. 2 and similar to *Tiny-CFA*, *DIALED* requires no hardware support other than PoX. *DIALED*'s instrumentation overhead includes logging inputs (which are typically small in number). Similarly, I-Log size depends on the number of arguments the application receives when it is invoked and the inputs it processes during its execution.

A Comparison of Software Integrity Verification Methods

This section compares the architectures discussed thus far. Table 1 presents a qualitative comparison, highlighting the type of security service offered by each architecture. Meanwhile, Fig. 4 reports a quantitative comparison, depicting hardware and software overheads.

As we target low-cost/low-power MCUs, we compare these architectures by instantiating them on OpenMSP430, an open-source version of the TI MSP430 MCUs. Therefore, the unmodified MSP430 is used as a reference baseline for comparison. Although our evaluation is conducted on MSP430, we emphasize that the suitability of these architectures extends to other MCUs within the same class, e.g., AVR ATMega and ARM Cortex-M MCUs. However, the proprietary nature of these designs precludes direct evaluation. Following the common practice in this space [2, 11], the hardware overhead is reported in terms of additional Look-Up Tables (LUTs) and registers. Added LUTs represent increase in combinatorial logic, whereas added registers are due to sequential logic required in each case.

Figure 4a depicts percentage increase in hardware relative to the CPU core cost. *VRASED* hybrid $\mathcal{R}$A architecture offers the simplest type of integrity evidence, i.e., whether *Prover* is currently loaded with the correct software image. Atop the baseline MCU, it incurs around 2% additional LUTs and 4.5% additional registers. *APEX*, which is a superset of *VRASED*'s hardware support, offers both $\mathcal{R}$A and PoX and requires 16% extra LUTs and 6% extra registers.

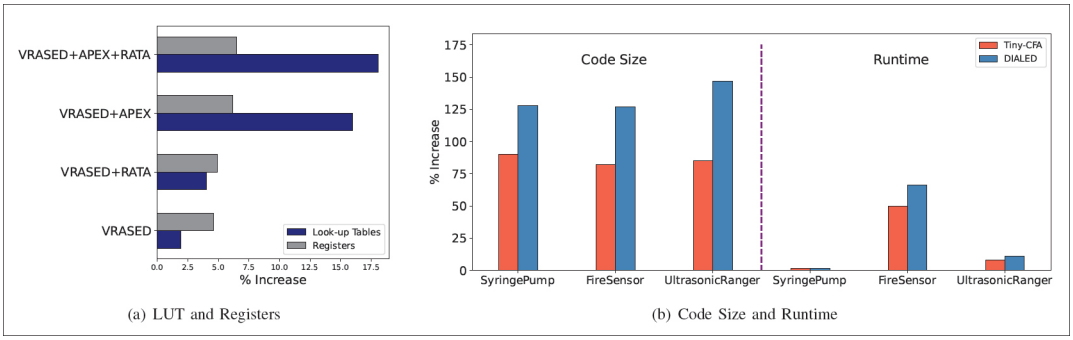

FIGURE 4. Hardware and software overhead of different architectures.

Tiny-CFA and *DIALED* add support for $\mathcal{C}$FA and $\mathcal{D}$FA. Since they rely on *APEX* hardware support for PoX "*as is*," they do not incur additional hardware costs. However, they lead to notable code size and runtime overhead as a result of the instrumentation phase. While the exact overhead is application-dependent (due to the variable number of branch instructions in different applications), we report *Tiny-CFA* and *DIALED* overheads on three real-world open-source MCU applications: Open Syringe Pump, Fire Sensor, and Ultrasonic Ranger. Details of these applications can be found in [13].

Figure 4b shows the percentage code size and runtime increase caused by instrumentation. On average, *Tiny-CFA* increases the code size by 85% and *DIALED* by 130%. Whereas, the runtime increase is approximately between 2–65% over their non-instrumented counterparts.

RATA can be viewed as an add-on to any of the aforementioned architectures to provide TOCTOU-security and reduced $\mathcal{R}$A computation time. When added to *VRASED* alone it adds 4% LUTs and 6% registers.

When considered in conjunction (as depicted in Fig. 2), all discussed features add up to 18% LUTs and 7% registers to the baseline MCU. Together they provide TOCTOU-Secure and faster $\mathcal{R}$A, PoX, as well as hardware support required by instrumentation-based $\mathcal{C}$FA and $\mathcal{D}$FA.

Open Problems and Opportunities

This section outlines open challenges and future research directions in this area.

1. *Formal Verification & Provable Security:* The trustworthiness of software integrity proofs also heavily depends on the correct implementation of the underlying architectures. *Formal verification* is a common approach to prove the correctness of the system implementation with respect to formal design specifications. *VRASED*, *APEX*, and *RATA* already employed this approach to ensure security/correctness in their hardware and software implementations. Nonetheless, there are no formally verified architectures for $\mathcal{C}$FA or $\mathcal{D}$FA. The main challenge in verifying *Tiny-CFA* and *DIALED* lies in how to verify security and correctness of the instrumentation phase. Given the absence of this phase in *VRASED/APEX/RATA*, the verification methods employed by these architectures cannot be directly applied here. Addressing this presents an interesting avenue for future research.
2. *Higher-End Devices:* This article examines software integrity techniques in resource-constrained MCUs. One avenue for future research involves extending these guarantees to higher-end devices. General-purpose CPUs (e.g., those featured in smartphones or desktops) are not as cost-prohibitive and thus often come equipped with more sophisticated hardware (e.g., MMUs or Trusted Execution Environments) or software modules (e.g., micro-kernels or monolithic operating systems). A promising opportunity for future research is to leverage the added hardware and software support to obtain similar guarantees to those considered in this article.
3. *Efficiency of $\mathcal{C}$FA and $\mathcal{D}$FA:* While *Tiny-CFA* and *DIALED* provide relatively low-cost $\mathcal{C}$FA and $\mathcal{D}$FA, the code size and run-time increases are still significant. Furthermore, as attested operations increase in size and complexity, the generated evidence traces (*CF-Log* and I-Log) also increase accordingly. As a consequence, $\mathcal{C}$FA and $\mathcal{D}$FA are still limited to simple self-contained operations in which associated evidence can be stored and transmitted by a resource-constrained MCU. An interesting direction for future work lies in the management and reduction of $\mathcal{C}$FA and $\mathcal{D}$FA associated costs.
4. *Attesting vs. Auditing Software Integrity:* Current architectures enable only *detection* of software compromises. They cannot guarantee that *Verifier* ever receives the produced evidence, in case of software attacks. While this suffices to detect if the *Prover* is compromised in a yes/no manner (in general, the absence of a signed report from the *Prover* indicates that something is wrong), it precludes *auditing* the generated evidence to pinpoint the source of compromises (i.e., to determine what is wrong with the *Prover*'s software). Auditing is non-trivial because a compromised *Prover* might ignore the protocol and simply refuse to send back evidence that indicates a compromise. Resolving this issue remains an open problem for future work.
5. Multi-Device Settings: This article focuses of software integrity techniques in a single-*Prover* setting. However, many IoT systems rely on a large group ("swarm") of interconnected devices. Simply applying single-*Prover* solutions to the swarm setting faces scalability issues. To address this, in the context of $\mathcal{R}$A, several "swarm/collective $\mathcal{R}$A" techniques have been proposed to efficiently

This article focuses of software integrity techniques in a single-*Prover* setting. However, many IoT systems rely on a large group ("swarm") of interconnected devices.

perform $\mathcal{R}$A across a multitude of devices. These techniques vary in their target settings, considering different factors, e.g., swarm topologies, swarm dynamics, and software/hardware heterogeneity. We refer to [15] for an in-depth overview of swarm $\mathcal{R}$A. Notably, *RATA* holds the potential to enhance existing swarm attestation schemes. With its advantages of constant runtime and TOCTOU security, *RATA* can serve as a building block in swarm $\mathcal{R}$A schemes, yielding faster overall swarm attestation while ensuring synchronized TOCTOU security across the swarm. Although swarm $\mathcal{R}$A has been extensively studied, the extension of other integrity services, i.e., PoX/$\mathcal{C}$FA/$\mathcal{D}$FA, to a swarm of devices remains largely unexplored, presenting an opportunity for future work.

Conclusions

This article overviews a series of techniques for remote verification of software integrity on resource-constrained IoT devices. Services provided by each technique vary, starting from simply verifying code installed on a remote device, to detection of transient malware, and detection of runtime attacks that arise due to vulnerabilities in installed code. Not surprisingly, techniques that provide more sophisticated service are also accompanied by increased complexity and costs. To assess these trade-offs, we compare them qualitatively and quantitatively, considering the security services provided in each case, additional hardware cost, and runtime overhead. Finally, we present new research directions and open problems.

Acknowledgment

We thank IEEE Communications' reviewers for constructive feedback. Gene Tsudik was supported in part by funding from NSF Award SATC-1956393, NSA Awards H98230-20-1-0345 and H98230-22-1-0308, as well as a subcontract from Peraton Labs. Ivan De Oliveira Nunes was supported by NSF Award SaTC- 2245531. Norrathep Rattanavipanon was supported by the National Science, Research and Innovation Fund (NSRF) and Prince of Songkla University (Grant No. COC6701016S).

References

[1] B. Kuang *et al.*, "A Survey of Remote Attestation in Internet of Things: Attacks, Countermeasures, and Prospects," *Computers & Security*, vol. 112, 2022, p. 102498.
[2] K. Eldefrawy *et al.*, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust," *NDSS*, 2012.
[3] I. De Oliveira Nunes *et al.*, "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation," *USENIX Security*, 2019.
[4] M. Grisafi *et al.*, "PISTIS: Trusted Computing Architecture for Low-End Embedded Systems," *USENIX Security*, 2022.
[5] P. Koeberl *et al.*, "Trustlite: A Security Architecture for Tiny Embedded Devices," *EuroSys*, 2014.
[6] I. De Oliveira Nunes *et al.*, "APEX: A Verified Architecture for Proofs of Execution on Remote Devices Under Full Software Compromise," *USENIX Security*, 2020.
[7] L. Szekeres *et al.*, "Sok: Eternal War in Memory," *IEEE S&P*, 2013.
[8] J. Noorman *et al.*, "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices," ACM TOPS, vol. 20, no. 3, 2017, pp. 1–33.
[9] I. De Oliveira Nunes *et al.*, "On the TOCTOU Problem in Remote Attestation," *ACM CCS*, 2021.
[10] T. Abera *et al.*, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," *ACM CCS*, 2016.
[11] G. Dessouky *et al.*, "LiteHAX: Lightweight Hardware-Assisted attestation of Program Execution," *ICCAD*, 2018.
[12] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution," *DATE*, 2021.
[13] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "DIALED: Data Integrity Attestation for Low-End Embedded Devices," *DAC*, 2021.
[14] Z. Sun *et al.*, "OAT: Attesting Operation Integrity of Embedded Devices," *IEEE S&P*, 2020.
[15] M. Ambrosin *et al.*, "Collective Remote Attestation at the Internet of Things Scale: State-of-the-Art and Future Challenges," *IEEE Commun. Surveys & Tutorials*, vol. 22, no. 4, 2020, pp. 2447–61.

Biographies

Ivan De Oliveira Nunes (ivanoliv@mail.rit.edu) is an Assistant Professor at the Rochester Institute of Technology (RIT). Before RIT, he received a Ph.D. from the University of California Irvine. His research interests span the fields of Security & Privacy, Computer Networking, Computing Systems, and especially their intersection.

Sashidhar Jakkamsetti (sashidhar.jakkamsetti@us.bosch.com) is a Research Scientist at Robert Bosch LLC - Research and Technology Center. He obtained his Ph.D. from the University of California, Irvine. Engineer at Microsoft in India (2016–2018). His research focuses on IoT Security/Privacy, Applied Cryptography, and Privacy-Preserving Technologies.

Norrathep Rattanavipanon (norrathep.r@phuket.psu.ac.th) received a Ph.D. from the University of California, Irvine. He is an Assistant Professor at College of Computing, Prince of Songkla University, Phuket, Thailand. His research interests include IoT security as well as software and binary analysis.

Gene Tsudik (gene.tsudik@uci.edu) received all his degrees eons ago. He's a fellow of all the usual societies. He dabbles in numerous security/privacy and applied crypto topics. He also occasionally composes atrocious crypto-poetry.