arXiv:2205.02963v1 [cs.CR] 5 May 2022

`Privacy-from-Birth`: Protecting Sensed Data from Malicious Sensors with `VERSA`

Ivan De Oliveira Nunes
Rochester Institute of Technology
ivanoliv@mail.rit.edu

Seoyeon Hwang
UC Irvine
seoyh1@uci.edu

Sashidhar Jakkamsetti
UC Irvine
sjakkams@uci.edu

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

Abstract—**With the growing popularity of the Internet-of-Things (IoT), massive numbers of specialized devices are deployed worldwide, in many everyday settings, including homes, offices, vehicles, public spaces, and factories. Such devices usually perform sensing and/or actuation. Many of them handle sensitive and personal data. If left unprotected, ambient sensing (e.g., of temperature, motion, audio, or video) can leak very private information. At the same time, some IoT devices use low-end computing platforms with few (or no) security features.**

There are many well-known techniques to secure sensed data, e.g., by authenticating communication end-points, encrypting data before transmission, and obfuscating traffic patterns. Such techniques protect sensed data from external adversaries while assuming that the sensing device itself is secure. Meanwhile, both the scale and frequency of IoT-focused attacks are growing. This prompts a natural question: *how to protect sensed data even if all software on the device is compromised?* Ideally, in order to achieve this, sensed data must be protected from its genesis, i.e., from the time when a physical analog quantity is converted into its digital counterpart and becomes accessible to software. We refer to this property as `PfB`: `Privacy-from-Birth`.

In this work, we formalize `PfB` and design Verified Remote Sensing Authorization (`VERSA`) – a provably secure and formally verified architecture guaranteeing that only correct execution of expected and explicitly authorized software can access and manipulate sensing interfaces, specifically, General Purpose Input/Output (GPIO), which is the usual boundary between analog and digital worlds on IoT devices. This guarantee is obtained with minimal hardware support and holds even if all device software is compromised. `VERSA` ensures that malware can neither gain access to sensed data on the GPIO-mapped memory nor obtain any trace thereof. `VERSA` is formally verified and its open-sourced implementation targets resource-constrained IoT edge devices, commonly used for sensing. Experimental results show that `PfB` is both achievable and affordable for such devices.

I. INTRODUCTION

The impact and importance of embedded (aka IoT or “smart”) devices is hard to overestimate. They are increasingly popular and becoming pervasive in many settings: from homes and offices to public spaces and industrial facilities. Not surprisingly, they also represent increasingly attractive attack targets for exploits and malware. In particular, low-end (cheap, small, and simple) micro-controller units (MCUs) are designed with strict cost, size, and energy limitations. Thus, it is hard to offer any concrete guarantees for tasks performed by these MCUs, due to their lack of sophisticated security and privacy features, compared to higher-end computing devices, such as smartphones or general-purpose IoT controllers, e.g., Amazon Echo or Google Nest. As MCUs increasingly permeate private spaces, exploits that abuse their sensing capabilities to obtain sensitive data represent a significant privacy threat.

Over the past decade, the IoT privacy issues have been recognized and explored by the research community [1], [2], [3], [4], [5]. Many techniques (e.g., [6], [7]) were developed to secure sensor data from active attacks that impersonate users, IoT back-ends, or servers. Another research direction focused on protecting private data from passive in-network observers that intercept traffic [8], [9], [10], [11] or perform traffic analysis based on unprotected packet headers and other metadata, e.g., sizes, timings, and frequencies. However, security of sensor data **on the device** which originates that data has not been investigated. We consider this to be a crucial issue, since all software on the device can be compromised and leak (exfiltrate) sensed data. Whereas, aforementioned techniques assume that sensing device runs the expected **benign** software.

We claim that in order to solve this problem, privacy of sensed data must be ensured “from birth”. This corresponds to two requirements: (1) access to sensing interfaces must be strictly controlled, such that only authorized code is allowed to read data, (2) sensed data must be protected as soon as it is converted to digital form. Even the simplest devices (e.g., motion sensors, thermostats, and smart plugs) should be protected since prior work [12], [13], [14], [15] amply demonstrates that private – and even safety-critical – information can be inferred from sensed data. It is also well-known that even simple low-end IoT devices are subject to malware attacks. This prompts a natural question: *Can privacy of sensed data be guaranteed if the device software is compromised?* We refer to this guarantee as `Privacy-from-Birth` (`PfB`).

Some previous results considered potential software compromise in low-end devices and proposed methods to enable security services, such as remote verification of device software state (remote attestation) [16], [17], [18], [19], [20], [21], proofs of remote software execution [22], control- & data-flow attestation [23], [24], [25], [26], [27], [28], [29], as well as proofs of remote software updates, memory erasure, and system reset [30], [31], [32].

Regardless of their specifics, such techniques only detect of violations or compromises **after the fact**. In the context of `PfB`, that is too late since leakage of private sensed data likely already occurred. Notably, SANCUS [17] specifically

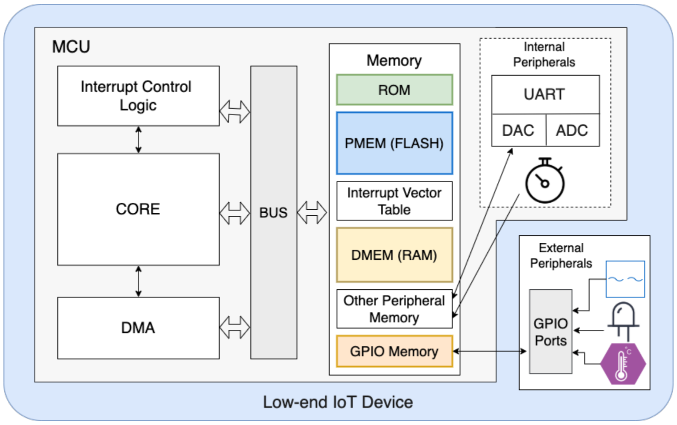


Fig. 1. System Architecture of an MCU-based IoT Device

discusses the problem of access control to sensor peripherals (e.g., GPIO) and proposes attestation of software accessing (or controlling access to) these peripherals. However, this only allows detection of compromised peripheral-accessing software and does not prevent illegal peripheral access.

To bridge this gap and obtain PfB, we construct the Verified Remote Sensing Authorization (VERSA) architecture. It provably prevents leakage of private sensor data even when the underlying device is software-compromised. At a high level, VERSA combines three key features: (1) *Mandatory Sensing Operation Authorization*, (2) *Atomic Sensing Operation Execution*, and (3) *Data Erasure on Boot* (see Section III). To attain these features, VERSA implements a minimal and formally verified hardware monitor that runs independently from (and in parallel with) the main CPU, without modifying the CPU core. We show that VERSA is an efficient and inexpensive means of guaranteeing PfB.

This work makes the following contributions:

- Formulates PfB with a high-level specification of requirements, followed by a game-based formal definition of the PfB goal.
- Constructs VERSA, an architecture that guarantees PfB.
- Implements and deploys VERSA on a commodity low-end MCU, which demonstrates its cost-effectiveness and practicality.
- Formally verifies VERSA implementation and proves security of the overall construction, hence obtaining provable security at both architectural and implementation levels. VERSA implementation and its computer proofs are publicly available in [33].

II. PRELIMINARIES

A. *Scope & MCU-based devices*

This work focuses on low-end CPS/IoT/smart devices with low computing power and meager resources. These are some of the smallest and weakest devices based on low-power single-core MCUs with only a few kilobytes (KB) of program and data memory. Two prominent examples are Atmel AVR ATmega and TI MSP430: 8- and 16-bit CPUs, respectively, typically running at 1-16MHz clock frequencies, with $\approx$ 64KB of addressable memory.

Figure 1 illustrates a generic architecture representing such MCUs. The CPU core and the Direct-Memory Access (DMA) controller access memory through a bus.[1] Memory can be divided into 5 logical regions: (1) Read-only memory (ROM), if present, stores critical software such as a bootloader, burnt into the device at manufacture time and not modifiable thereafter; (2) program memory (PMEM), usually realized as flash, is non-volatile and stores program instructions; (3) interrupt vector table (also in flash and often considered as part of PMEM), stores interrupt configurations; (4) data memory (DMEM), usually implemented with DRAM, is volatile and used to store program execution state, i.e., its stack and heap; and, (5) peripheral memory region (also in DRAM and often considered as a part of DMEM), contains memory-mapped I/O interfaces, i.e., addresses in the memory layout that are mapped to hardware components, e.g., timers, UART, and GPIO. In particular, GPIO are peripheral memory addresses hardwired to physical ports that interface with external circuits, e.g., analog sensors/circuits.

We note that small MCUs usually come in one of two memory architectures: Harvard and von Neumann. The former isolates PMEM and DMEM by maintaining two different buses and address spaces, while the latter keeps both PMEM and DMEM in the same address space and accessible via a single bus.

Low-end MCUs execute instructions in place, i.e., directly from flash memory. They have neither memory management units (MMUs) to support virtualization/isolation, nor memory protection units (MPUs). Therefore, privilege levels and isolation used in higher-end devices and generic enclaved execution systems (e.g., Intel SGX [34] or MIT SANCTUM [35]) are not applicable.

We believe that a PfB-agile architecture that is sufficiently inexpensive and efficient for such low-end devices can be later adapted to more powerful devices. Whereas, going in the opposite direction is more challenging. Furthermore, simpler devices are easier to model and reason about formally. Thus, we believe that they represent a natural starting point for the design and verification of a PfB-agile architecture. To this end, our prototype implementation of VERSA is integrated with MSP430, due in part to public availability of an open-source MSP430 hardware design from OpenCores [36].[2]

B. *GPIO & MCU Sensing*

A GPIO port is a set of GPIO pins arranged and controlled together, as a group. The MCU-addressable memory for a GPIO port is physically mapped (hard-wired) to physical ports that can be connected to a variety of external circuits, such as analog sensors and actuators, as shown in Figure 1. Each GPIO pin can be set to function as either an input or output, hence called "general purpose". Input signals produced by external circuits can be obtained by the MCU software by reading from GPIO-mapped memory. Similarly, egress electric signals (high or low voltage) can be generated by the MCU software by writing (logical 1 or 0) to GPIO-mapped memory.

---

[1] DMA is a hardware controller that can read/write to memory in parallel with the CPU.

[2] Nevertheless, the generic machine model and methodology of VERSA are applicable to other low-end MCUs of the same class, e.g., Atmel AVR ATmega.

Remark: "GPIO-mapped memory" includes the set of all software-readable memory regions connected to external sensors. In some cases, this set may even include multiple physical memory regions for a single physical pin. For instance, if a given GPIO pin is also equipped with an Analog-to-Digital Converter (ADC), a GPIO input could be reflected on different memory regions depending on whether the ADC is active or inactive. All such regions are considered "GPIO-mapped memory" and we refer to it simply as GPIO. Using this definition, in order to access sensor data, software running on the MCU must read from GPIO.

We also note that various applications require different sensor regimes [37]: event-driven, periodic, and on-demand. Event-driven sensors report sensed data when a trigger event occurs, while periodic sensors report sensor data at fixed time intervals. On-demand (or query-driven) sensors report sensor data whenever requested by an external entity. Although we initially consider on-demand sensing, as discussed in Section III, the proposed design is applicable to other regimes.

C. Remote Attestation & VRASED

Remote Attestation ($\mathcal{R}\mathsf{A}$) allows a trusted entity (verifier = $\mathcal{V}\mathsf{rf}$) to remotely measure current memory contents (e.g., software binaries) of an untrusted embedded device (prover = $\mathcal{P}\mathsf{rv}$). $\mathcal{R}\mathsf{A}$ is usually realized as a challenge-response protocol:

1) $\mathcal{V}\mathsf{rf}$ sends an attestation request containing a challenge ($\mathcal{C}\mathsf{hal}$) to $\mathcal{P}\mathsf{rv}$.
2) $\mathcal{P}\mathsf{rv}$ receives the request and computes an authenticated integrity check over its memory and $\mathcal{C}\mathsf{hal}$. The memory region can be either pre-defined or explicitly specified in the attestation request.
3) $\mathcal{P}\mathsf{rv}$ returns the result to $\mathcal{V}\mathsf{rf}$.
4) $\mathcal{V}\mathsf{rf}$ verifies the result and decides if it corresponds to a valid $\mathcal{P}\mathsf{rv}$ state.

VRASED [18] is a verified hybrid (hardware/software) $\mathcal{R}\mathsf{A}$ architecture for for low-end MCUs. It comprises a set of (individually) verified hardware and software sub-modules; their composition provably satisfies formal definitions of $\mathcal{R}\mathsf{A}$ soundness and security. *VRASED* software component implements the authenticated integrity function computed over a given "Attested Region" (AR) of $\mathcal{P}\mathsf{rv}$'s memory. *VRASED* hardware component assures that its software counterpart executes securely and that no function of the secret key is ever leaked. In short, $\mathcal{R}\mathsf{A}$ soundness states that the integrity measurement must accurately reflect a snapshot of $\mathcal{P}\mathsf{rv}$'s memory in AR, disallowing any modifications to AR during the actual measurement. $\mathcal{R}\mathsf{A}$ security defines that the measurement must be unforgeable, implying protection of secret key $\mathcal{K}$ used for the measurement.

In order to prevent DoS attacks on $\mathcal{P}\mathsf{rv}$, the RA protocol may involve authentication of the attestation request, before $\mathcal{P}\mathsf{rv}$ performs attestation. If this feature is used, an authentication token must accompany every attestation request.[3] For example, in *VRASED*, $\mathcal{V}\mathsf{rf}$ computes this token as an HMAC over $\mathcal{C}\mathsf{hal}$, using $\mathcal{K}$. Since $\mathcal{K}$ is only known to $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$, this token is unforgeable. To prevent replays, $\mathcal{C}\mathsf{hal}$ is a monotonically increasing counter, and the latest $\mathcal{C}\mathsf{hal}$ used to successfully authenticate $\mathcal{V}\mathsf{rf}$ is stored by $\mathcal{P}\mathsf{rv}$ in persistent and protected memory. In each attestation request, incoming $\mathcal{C}\mathsf{hal}$ must be greater that the stored value. Once an attestation request is successfully authenticated, the stored value is updated accordingly.

VRASED software component is stored in ROM and realized with a formally verified HMAC implementation from the HACL* cryptographic library [38], which is used to compute: $H = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), AR)$, where $KDF(\mathcal{K}, \mathcal{C}\mathsf{hal})$ is a one-time key derived from the received $\mathcal{C}\mathsf{hal}$ and $\mathcal{K}$ using a key derivation function.

As discussed later in Section VI, in VERSA, *VRASED* is used as a means of authorizing a binary to access GPIO.

D. LTL, Model Checking, & Verification

Our verification and proof methodologies are in-line with prior work on the design and verification of security architectures proving code integrity and execution properties for the same class of MCUs [18], [22], [39], [40]. However, to the best of our knowledge, no prior work tackled formal models and definitions, or designed services, for guaranteed sensed data privacy. This section overviews our verification and proof methodologies that allow us to later show that VERSA achieves required PfB properties and end-goals.

Computer-aided formal verification typically involves three steps. First, the system of interest (e.g., hardware, software, or communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system retains them. This can be done via Theorem Proving [41] or Model Checking [42]. We use the latter to verify the implementation of system sub-modules, and the former to prove new properties derived from the combination (conjunction) of machine model axioms and sub-properties that were proved for the implementation of individual sub-modules.

In one instantiation of model checking, properties are specified as *formulae* using Linear Temporal Logic (LTL) and system models are represented as FSMs. Hence, a system is represented by a triple: (σ, σ_0, T), where σ is the finite set of states, $\sigma_0 \subseteq \sigma$ is the set of possible initial states, and $T \subseteq \sigma \times \sigma$ is the transition relation set, which describes the set of states that can be reached in a single step from each state. Such usage of LTL allows representing a system behavior over time.

---

[3] By saying "this feature is used", we mean that its usage (or lack thereof) is fixed at the granularity of a $\mathcal{V}\mathsf{rf}$-$\mathcal{P}\mathsf{rv}$ setting, and not per single $\mathcal{R}\mathsf{A}$ instance.

Our verification strategy benefits from the popular model checker NuSMV [43], which can verify generic hardware or software models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV models is simple. Furthermore, it can be automated [44] as the standard RTL design already relies on describing hardware as FSMs. LTL specifications are particularly useful for verifying sequential systems. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL extends propositional logic with **temporal quantifiers**, thus enabling sequential reasoning. In this paper, we are interested in the following LTL quantifiers:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
- ϕ $\mathbf{U}$ ψ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- ϕ $\mathbf{W}$ ψ – ϕ Weak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. Or, more formally: $\phi \; \mathbf{W} \; \psi \equiv (\phi \; \mathbf{U} \; \psi) \vee \mathbf{G}(\phi)$.
- ϕ $\mathbf{B}$ ψ – ϕ Before ψ: holds if the existence of state where ψ holds implies the existence of at least one earlier state where ϕ holds. Equivalently: $\phi \; \mathbf{B} \; \psi \equiv \neg(\neg\phi \; \mathbf{U} \; \psi)$.

NuSMV works by exhaustively enumerating all possible states of a given system FSM and by checking each state against LTL specifications. If any desired specification is found not to hold for specific states (or transitions between states), the model checker provides a trace that leads to the erroneous state, which helps correct the implementation accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale due to the so-called "state explosion" problem. To cope with it, our verification approach is to specify smaller LTL sub-properties separately and verify each respective hardware sub-module for compliance. In this process, our verification pipeline automatically converts digital hardware, described at RTL using Verilog, to Symbolic Model Verifier (SMV) [45] FSMs using Verilog2SMV [44]. The SMV representation is then fed to NuSMV for verification. Then, the composition of LTL sub-properties (verified in the model-checking phase) is proven to achieve a desired end-to-end implementation goal, also specified in LTL. This step uses an LTL theorem prover [46].

In our case, we show that the end-to-end goal of VERSA, in composition with *VRASED*, is sufficient to achieve PfB via cryptographic reduction from the formal security definition of *VRASED*. These steps are discussed in detail in Section VII.

III. VERSA OVERVIEW

VERSA involves two entities: a trusted remote controller ($\mathcal{C}$trl) and a device ($\mathcal{D}$ev). We expect $\mathcal{C}$trl to be a relatively powerful computing entity, e.g., a home gateway, a backend server or even a smartphone. VERSA protects sensed data on $\mathcal{D}$ev by keeping it (and any function thereof) confidential. This implies: (1) controlling GPIO access by blocking attempted reads by unauthorized software, and (2) keeping execution traces (i.e., data allocated by GPIO-authorized software) confidential. Therefore, access to GPIO is barred by default. GPIO is unlocked only for benign binaries that are pre-authorized by $\mathcal{C}$trl. Whenever a binary is deemed to be authorized on $\mathcal{D}$ev, VERSA creates for it an ephemeral isolated execution environment and permits its one-time execution. This isolated environment lasts until execution ends, which corresponds to reaching the legal exit point of the authorized binary. Therefore, by including a clean-up routine immediately before the legal exit, we can assure that all execution traces, including all sensitive information, are erased. Any attempt to interrupt, or tamper with, isolated execution causes an immediate system-wide reset, which erases all data traces.

We use the term "Sensing Operation", denoted by $\mathcal{S}$, to refer to a self-contained and logically independent binary (e.g., a function) that is responsible for processing data obtained through one or more reads from GPIO.

VERSA achieves PfB via three key features:

[A] *Mandatory Sensing Operation Authorization* requires explicit authorization issued by $\mathcal{C}$trl before any $\mathcal{D}$ev software reads from GPIO. Recall that access to GPIO is blocked by default. Each authorization token (ATok) coming from $\mathcal{C}$trl allows one execution of a specific sensing operation $\mathcal{S}$, although a single execution of $\mathcal{S}$ can implement several GPIO reads. ATok has the following properties:

1) It can be authenticated by $\mathcal{D}$ev as having been issued by $\mathcal{C}$trl; this includes freshness;
2) It grants privileges **only to a specific** $\mathcal{S}$ to access GPIO during its execution; and
3) It can only be used once.

$\mathcal{C}$trl can authorize multiple executions of $\mathcal{S}$ by issuing a batch of tokens, i.e., $\mathsf{ATok}_1, ..., \mathsf{ATok}_n$, for up to n executions of $\mathcal{S}$. Although supporting multiple tokens is unnecessary for on-demand sensing, it might be useful for periodic or event-driven sensing regimes discussed in Section II-B.

[B] *Atomic Sensing Operation Execution* ensures that, once authorized by $\mathcal{C}$trl, $\mathcal{S}$ is executed with the following requirements:

1) $\mathcal{S}$ execution starts from its legal entry point (first instruction) and runs until its legal exit point (last instruction). This assumes a single pair of entry-exit points;
2) $\mathcal{S}$ execution can not be interrupted and its intermediate results cannot be accessed by external means, e.g., via DMA controllers; and
3) An immediate MCU reset is triggered if either (1) or (2) above is violated.

[C] *Data Erasure on Reset/Boot* works with **[B]** to guarantee that, sensed data (or any function thereof) obtained during $\mathcal{S}$ execution is not leaked due to errors or violations of security properties, which cause MCU reset per item (3) above. This feature must guarantee that all values that remain in RAM after a hard reset and the subsequent boot process, are erased before any unprivileged software can run. While some architectures already provide memory erasure on boot, for those MCUs that do not do so, it can be obtained by calling a secure RAM erasure function at boot time, e.g., as a part of a ROM-resident

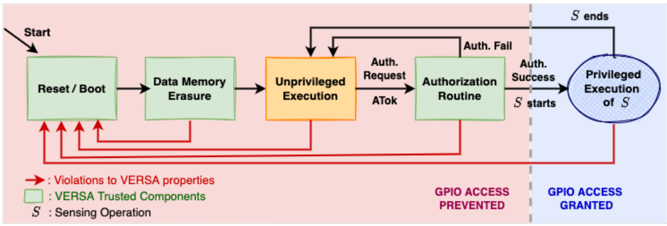


Fig. 2. MCU execution workflow with VERSA.

bootloader code. Appendix B discusses this further.

At a high level, correct implementation of aforementioned three features suffices to obtain PfB, because:

- Any compromised/modified binary can not access GPIO since it has no authorization from $\mathcal{C}$trl.
- Any authorized binary $\mathcal{S}$ must be invoked properly and run atomically, from its first, and until its last, instruction.
- Since $\mathcal{S}$ is invoked properly, intended behavior of $\mathcal{S}$ is preserved. Code reuse attacks are not possible, unless they occure as a result of bugs in $\mathcal{S}$ implementation itself. $\mathcal{C}$trl can always check for such bugs in $\mathcal{S}$ prior to authorization; see Section V-B.
- $\mathcal{S}$ runs uninterrupted, meaning that it can erase all traces of its own execution from the stack before passing control to unprivileged applications. This guarantees that no sensor data remains in memory when $\mathcal{S}$ terminates.
- VERSA assures that any violation of aforementioned requirements causes an MCU reset, triggering erasure of all data memory. Therefore, malware that attempts to interrupt $\mathcal{S}$ before completion, or tamper with $\mathcal{S}$ execution integrity, will cause all data used by $\mathcal{S}$ to be erased.

Support for Output Encryption: $\mathcal{S}$ might process and use sensor data locally as part of its own execution, or generate some output that needs to be returned to $\mathcal{C}$trl. In the latter case, encryption of $\mathcal{S}$ output is necessary. For this reason, VERSA supports the generation of a fresh key derived from ATok (thus implicitly shared between $\mathcal{C}$trl and $\mathcal{S}$). This key is only accessible to $\mathcal{S}$ during authorized execution. Hence, $\mathcal{S}$ can encrypt any data to be exported with this key and ensure that encrypted results can only be decrypted by $\mathcal{C}$trl.

Since we assume that the encryption function is part of $\mathcal{S}$, it cannot be interrupted (or tampered with) by any unprivileged software or external means. Importantly, the encryption key is only accessible to $\mathcal{S}$ (similar to GPIO) and shielded from all other software. Furthermore, the choice of the encryption algorithm is left up to the specific $\mathcal{S}$ implementation.

Figure 2 illustrates MCU execution workflow discussed in this section.

IV. MCU MACHINE MODEL

A. Execution Model

To enable formal specification of PfB guarantees, we formulate the MCU execution model in Definition 1. It represents MCU operation as a discrete sequence of MCU states, each corresponding to one clock cycle – the smallest unit of time in the system. We say that the subsequent MCU state is defined based on the current MCU state (which includes current values in memory/registers, as well as any hardware signals and effects, such as external inputs, actions by DMA controller(s), and interrupts) and the current instruction being executed by the CPU core. Similarly, the instruction to be executed in the next state is determined by the current state and the current instruction being executed.

For example, an arithmetic instruction (e.g., `add` or `mult`) causes the program counter (PC) to point to the subsequent address in physical memory. However, an interrupt (which is a consequence of the current MCU state) may occur and deviate the normal execution flow. Alternatively, a branching instruction may be executed and cause PC to jump to some arbitrary instruction that is not necessarily located at the subsequent position in the MCU flash memory.

In order to reason about events during the MCU operation, we say that each MCU state can belong to one or more sets. Belonging to a given set implies that the state has a given property of interest. Definition 1 introduces six sets of interest, representing states in which memory is read/written by CPU or DMA, as well as states in which an interrupt or reset occurs.

B. Hardware Signals

We now formalize the effects of execution, modeled in Definition 1, to the values of concrete hardware signals that can be monitored by VERSA hardware in order to attain PfB guarantees. Informally, we model the following simple axioms:

[A1] PC: contains the memory address containing the instruction being executed at a given cycle.

[A2] CPU Memory Access: Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. A data read-enable bit (R_{en}) must be set for a read access and a data write-enable bit (W_{en}) must be set for a write access.

[A3] DMA: Whenever a DMA controller attempts to access the main memory, a DMA-address signal (DMA_{addr}) contains the address of the accessed memory location and a DMA-enable bit (DMA_{en}) must be set.

[A4] Interrupts: When hardware interrupts or software interrupts happen, the irq signal is set.

[A5] MCU reset: At the end of a successful reset routine, all registers (including PC) are set to zero before restarting software execution. The reset handling routine cannot be modified, as resets are handled by MCU in hardware. When a reset happens, the corresponding $reset$ signal is set. The same signal is also set when the MCU initializes for the first time.

This model strictly adheres to MCU specifications, assumed to be correctly implemented by the underlying MCU core.

Definition 2 presents formal specifications for aforementioned axioms in LTL. Instead of explicitly quantifying time, LTL embeds time within the logic by using temporal quantifiers (see Section II). Hence, rather than referring to execution

Definition 1 (MCU Execution Model)**.**

***1 – Execution** is modeled as a sequence of MCU states $S := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$ and a sequence of instructions $I := \{\mathsf{i}_0, ..., \mathsf{i}_n\}$. Since the next MCU state and the next instruction to be executed are determined by the current MCU state and the current instruction being executed, these discrete transitions are denoted as shown in the following example:*

$$(\mathsf{s}_1, \mathsf{i}_j) \leftarrow \textit{EXEC}(\mathsf{s}_0, \mathsf{i}_0); \quad (\mathsf{s}_2, \mathsf{i}_k) \leftarrow \textit{EXEC}(\mathsf{s}_1, \mathsf{i}_j); \quad ... \quad (\mathsf{s}_m, \perp) \leftarrow \textit{EXEC}(\mathsf{s}_{m-1}, \mathsf{i}_l)$$

The sequence I represents the physical order of instructions in memory, which is not necessarily the order of their execution. The next instruction and state are also affected by current external inputs, current data-memory values, and current hardware events, e.g., interrupts or resets, which are modeled as properties of each execution state in S. The MCU always starts execution (at boot or after a reset) from state s_0 and initial instruction i_0. EXEC produces $\perp$ as the next instruction if there is no instructions left to execute.

***2 – State Properties as Sets:** sets are used to model relevant execution properties and characterize effects/actions occurring within a given state s_t. We are particularly interested in the behaviors corresponding to the following sets:*

1) ***READ:** all states produced by the execution of an instruction i that reads the value from memory to a register.*
2) ***WRITE:** all states produced by the execution of an instruction i that writes the value from a register to memory.*
3) ***DMA^R:** all states produced as a result of DMA reading from memory.*
4) ***DMA^W:** all states s_t produced as a result of DMA writing to memory.*
5) ***IRQ:** all states s_t where an interrupt is triggered.*
6) ***RESET:** all states s_t wherein an MCU reset is triggered.*

Note that these sets are not disjoint, i.e., s_t can belong to multiple sets. Also, the aforementioned sets do not aim to model all possible MCU behaviors, but only the ones relevant to PfB. Finally, we further subdivide sets that model memory access into subsets relating to memory regions of interest. For example, considering a contiguous memory region $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$, $READ_{\mathcal{M}}$ is a subset of READ containing only the states produced through EXEC of instructions that read from the memory region $\mathcal{M}$. We use the same notation to refer to other subsets, e.g., $WRITE_{\mathcal{M}}$, $DMA^R_{\mathcal{M}}$, and $DMA^W_{\mathcal{M}}$.

Definition 2 (Hardware Model)**.**

$\mathcal{M}$ denotes a contiguous memory region within addresses $\mathcal{M}_{min}$ and $\mathcal{M}_{max}$ in physical memory of $\mathcal{D}$ev, i.e., $\mathcal{M} := [\mathcal{M}_{min}, \mathcal{M}_{max}]$.
s represents the system execution state at a given CPU cycle.

Program counter & instruction execution:

$$\mathbf{G} : \{[\mathbf{X}(\mathsf{s}) \leftarrow \textit{EXEC}(\mathsf{s}, \mathsf{i}_k) \land \mathsf{i}_k \in \mathcal{M})] \rightarrow (PC \in \mathcal{M})\} \quad (1)$$

Memory Reads/Writes:

$$\mathbf{G} : \{\mathbf{X}(\mathsf{s}) \in \textit{READ}_{\mathcal{M}} \rightarrow (R_{en} \land D_{addr} \in \mathcal{M})\} \quad (2)$$

$$\mathbf{G} : \{\mathbf{X}(\mathsf{s}) \in \textit{WRITE}_{\mathcal{M}} \rightarrow (W_{en} \land D_{addr} \in \mathcal{M})\} \quad (3)$$

$$\mathbf{G} : \{(\mathbf{X}(\mathsf{s}) \in \textit{DMA}^R_{\mathcal{M}} \lor \mathbf{X}(\mathsf{s}) \in \textit{DMA}^W_{\mathcal{M}}) \rightarrow (DMA_{en} \land DMA_{addr} \in \mathcal{M})\} \quad (4)$$

Interrupts (irq) and Resets:

$$\mathbf{G} : \{\mathsf{s} \in \textit{IRQ} \leftrightarrow irq\} \quad (5)$$

$$\mathbf{G} : \{\mathsf{s} \in \textit{RESET} \leftrightarrow reset\} \quad (6)$$

states using temporal variables (i.e., state t, state $t+1$, state $t+2$), a single variable (s) and LTL quantifiers suffice to specify, e.g., "current", "next", "future" system states (s). For this part of the model, we are mostly interested in: (1) describing MCU state at the next CPU cycle ($\mathbf{X}(\mathsf{s})$) as a function of the MCU state at the current CPU cycle (s), and (2) describing which particular MCU signals must be triggered in order for $\mathbf{X}(\mathsf{s})$ to be in each of the sets defined in Definition 1.

LTL statements in Definition 2 formally model axioms **[A1]-[A5]**, i.e., the subset of MCU behavior that is relevant to, and sufficient for formally verifying, VERSA. LTL (1) models **[A1]**, (2) and (3) model **[A2]**, and each (4), (5), and (6) models **[A3]**, **[A4]**, and **[A5]**, respectively.

V. PfB DEFINITIONS

Based on the specified machine model, we now proceed with the formal definition of PfB.

A. PfB Syntax

A PfB scheme involves two parties: $\mathcal{C}$trl and $\mathcal{D}$ev. $\mathcal{C}$trl authorizes $\mathcal{D}$ev to execute some software $\mathcal{S}$ which accesses GPIO. It should be impossible for any software different from $\mathcal{S}$ to access GPIO data, or any function thereof (see Definition 4). $\mathcal{C}$trl is trusted to only authorize functionally correct code. The goal of a PfB scheme is to facilitate sensing-dependent execution while keeping all sensed data private from all other software.

Definition 3 specifies a syntax for PfB scheme composed of three functionalities: Authorize, Verify, and XSensing. Authorize is invoked by $\mathcal{C}$trl to produce an authorization token, ATok, to be sent to $\mathcal{D}$ev, enabling $\mathcal{S}$ to access GPIO. Verify is executed at $\mathcal{D}$ev with ATok as input, and it checks whether ATok is a valid authorization for the software on $\mathcal{D}$ev. If and only if this check succeeds, Verify returns $\top$. Otherwise, it returns $\perp$. The verification success indicates one execution of $\mathcal{S}$ granted on $\mathcal{D}$ev via XSensing. XSensing is considered successful (returns $\top$), if there is at least one MCU state produced by XSensing where a GPIO read occurs without causing an MCU $reset$, i.e., $(\mathsf{s} \in \textit{READ}_{GPIO}) \land \neg(\mathsf{s} \in \textit{RESET})$. Otherwise, XSensing returns $\perp$. That is, XSensing models execution of any software in the MCU and its return symbol indicates whether a GPIO read occurred during its execution. Therefore, invocation of XSensing on any input software that does not read from GPIO returns $\perp$. Figure 3 illustrates a benign PfB interaction between $\mathcal{C}$trl and $\mathcal{D}$ev.

Definition 3 (Syntax: `PfB` scheme)**.**

A `Privacy-from-Birth` *(*`PfB`*) scheme is a tuple of algorithms* $[\mathsf{Authorize}, \mathsf{Verify}, \mathsf{XSensing}]$*:*

1) $\mathsf{Authorize}^{\mathcal{C}\mathsf{trl}}(\mathcal{S}, \cdots)$*: an algorithm executed by* $\mathcal{C}\mathsf{trl}$ *taking as* *input* *at least one executable* $\mathcal{S}$ *and* *producing* *at least one authorization token* ATok *which can be sent to* $\mathcal{D}\mathsf{ev}$ *to authorize one execution of* $\mathcal{S}$ *with access to* `GPIO`*.*
2) $\mathsf{Verify}^{\mathcal{D}\mathsf{ev}}(\mathcal{S}, \mathit{ATok}, \cdots)$*: an algorithm (with possible hardware-support), executed by* $\mathcal{D}\mathsf{ev}$*, that takes as* *input* $\mathcal{S}$ *and* ATok*. It uses* ATok *to check whether* $\mathcal{S}$ *is pre-authorized by* $\mathcal{C}\mathsf{trl}$ *and outputs* $\top$ *if verification succeeds, and* $\perp$ *otherwise.*
3) $\mathsf{XSensing}^{\mathcal{D}\mathsf{ev}}(\mathcal{S}, \cdots)$*: an algorithm (with possible hardware-support) that executes* $\mathcal{S}$ *in* $\mathcal{D}\mathsf{ev}$*, producing a sequence of states* $E := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$*. It returns* $\top$*, if sensing successfully occurs during* $\mathcal{S}$ *execution, i.e.,* $\exists \mathsf{s} \in E$ *such that* $(\mathsf{s} \in \mathit{READ}_{\mathit{GPIO}}) \wedge (\mathsf{s} \notin \mathit{RESET})$*); it returns* $\perp$*, otherwise.*

Remark: *In the parameter list,* $(\cdots)$ *means that additional/optional parameters might be included depending on the specific* `PfB` *construction.*

Definition 4 (`PfB` Game-based Definition)**.**

4.1 Auxiliary Notation & Predicate(s):

- *Let* $\mathcal{K}$ *be a secret string of bit-size* $|\mathcal{K}|$*; and* λ *be the security parameter, determined by* $|\mathcal{K}|$*, i.e.,* $\lambda = \Theta(|\mathcal{K}|)$*;*
- *Let* $\mathsf{atomicExec}$ *be a predicate evaluated on some sequence of states* S *and some software – i.e., some sequence of instructions* I*.*
  - $\mathsf{atomicExec}(S := \{\mathsf{s}_1, ..., \mathsf{s}_m\}, I := \{\mathsf{i}_0, ..., \mathsf{i}_n\}) \equiv \top$ *if and only if the following hold; otherwise,* $\mathsf{atomicExec}(S, I) \equiv \perp$*.*
    1) ***Legal Entry Instruction:*** *The first execution state* s_1 *in* S *is produced by the execution of the first instruction* i_0 *in* I*.* *i.e.,* $(\mathsf{s}_1 \leftarrow \mathit{EXEC}(\mathsf{i}_0, \mathsf{s}*)) \ \vee \ (\mathsf{s}_1 \in \mathit{RESET})$*, where* $\mathsf{s}*$ *is any state prior to* s_1*.*
    2) ***Legal Exit Instruction:*** *The last execution state* s_m *in* S *is produced by the execution of the last instruction* i_n *in* I*.* *i.e.,* $(\mathsf{s}_m \leftarrow \mathit{EXEC}(\mathsf{i}_n, \mathsf{s}_{m-1})) \ \vee \ (\mathsf{s}_m \in \mathit{RESET})$*.*
    3) ***Self-Contained Execution:*** *For all* s_j *in* S*,* s_j *is produced by the execution of an instruction* i_k *in* I*, for some* k*.* *i.e.,* $(\mathsf{s}_j \leftarrow \mathit{EXEC}(\mathsf{i}_k, \mathsf{s}_{j-1})) \ \vee \ (\mathsf{s}_j \in \mathit{RESET})$*, for some* $\mathsf{i}_k \in I$*.*
    4) ***No Interrupts, No DMA:*** *For all* s_j *in* S*,* s_j *is neither in the IRQ or DMA. i.e.,* $[(\mathsf{s}_j \notin \mathit{IRQ}) \wedge (\mathsf{s}_j \notin \mathit{DMA})] \ \vee \ (\mathsf{s}_j \in \mathit{RESET})$*.*

4.2 **`PfB`*-Game:*** *The challenger plays the following game with* $\mathcal{A}\mathsf{dv}$*:*

1) $\mathcal{A}\mathsf{dv}$ *is given full control over* $\mathcal{D}\mathsf{ev}$ *software state, implying* $\mathcal{A}\mathsf{dv}$ *can execute any (polynomially sized) sequence of arbitrary instructions* $\{\mathsf{i}_0^{\mathcal{A}\mathsf{dv}}, ..., \mathsf{i}_n^{\mathcal{A}\mathsf{dv}}\}$*, inducing the associated changes in* $\mathcal{D}\mathsf{ev}$*'s sequence of execution states;*
2) $\mathcal{A}\mathsf{dv}$ *has oracle access to polynomially many calls to* Verify*.* $\mathcal{A}\mathsf{dv}$ *also has access to the set of software executables,* $\mathsf{SW} := \{S_1, ..., S_l\}$*, and the set of all corresponding authorization "tokens",* $\mathsf{T} := \{\mathit{ATok}_1, ..., \mathit{ATok}_l\}$*, ever produced by any prior* $\mathcal{C}\mathsf{trl}$ *calls to* $\mathsf{Authorize}$ *up until time* t*. i.e.,* $\mathit{ATok}_j \leftarrow \mathsf{Authorize}(\mathcal{S}_j, ...)$*, for all* j*.*
3) *Let* $\mathsf{U} \subset \mathsf{T}$ *be the set of all "used" authorization tokens up until time* t*, i.e.,* $\mathit{ATok}_j \in \mathsf{U}$*, if a call to* $\mathsf{XSensing}(\mathcal{S}_j, ...)$ *returned* $\top$ *up until time* t*; Let* P *be the set of "pending" (issued but not used) authorization tokens, i.e.,* $\mathsf{P} := \mathsf{T} \setminus \mathsf{U}$*.*
4) *At any arbitrary time* t*,* $\mathcal{A}\mathsf{dv}$ *wins if it can perform an* ***unauthorized or tampered sensing execution****, i.e.:*
   - $\mathcal{A}\mathsf{dv}$ *triggers an* $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}, ...)$ *operation that returns* $\top$*, for* $\forall \mathcal{S}_{\mathcal{A}\mathsf{dv}} \notin \mathsf{SW}$*, or*
   - $\mathcal{A}\mathsf{dv}$ *triggers* $(S, \top) \leftarrow \mathsf{XSensing}(\mathcal{S}_j, ...)$ *such that* $\mathsf{atomicExec}(S, \mathcal{S}_j) \equiv \perp$*, for some* $\mathcal{S}_j \in \mathsf{SW}$ *and* $\mathit{ATok}_j \in \mathsf{U}$ *.*

4.3 **`PfB`*-Security:*** *A scheme is considered* `PfB`*-Secure iff, for all PPT adversaries* $\mathcal{A}\mathsf{dv}$*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \mathit{PfB}\text{-}\mathit{Game}] \leq \mathsf{negl}(\mathsf{l})$$

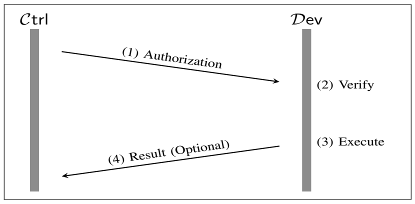


Fig. 3. `PfB` interaction between $\mathcal{C}\mathsf{trl}$ and $\mathcal{D}\mathsf{ev}$

B. Assumptions & Adversarial Model

We consider an adversary, $\mathcal{A}\mathsf{dv}$, that controls the entire software state of $\mathcal{D}\mathsf{ev}$, including PMEM (flash) and DMEM (DRAM). It can attempt to modify any writable memory (including PMEM) or read any memory, including peripheral regions, such as `GPIO`, unless explicitly protected by verified hardware. It can launch code injection attacks to execute arbitrary instructions from PMEM or even DMEM (if the MCU architecture supports execution from DMEM). It also has full control over any DMA controllers on $\mathcal{D}\mathsf{ev}$ that can directly read/write to any part of the memory independently of the CPU. It can induce interrupts to pause any software execution and leak information from its stack, or change its control-flow. We consider Denial-of-Service (DoS) attacks, whereby $\mathcal{A}\mathsf{dv}$ abuses `PfB` functionality in order to render $\mathcal{D}\mathsf{ev}$ unavailable, to be out-of-scope. These are attacks on $\mathcal{D}\mathsf{ev}$ availability and not on sensed data privacy.

Executable Correctness: we stress that `VERSA` aims to guarantee that $\mathcal{S}$, as specified by $\mathcal{C}\mathsf{trl}$, is the only software that can access and process `GPIO` data. Similar to other trusted hardware architectures, `PfB` does not check for lack of implementation bugs within $\mathcal{S}$; thus it is not concerned with run-time (e.g., control-flow and data-only) attacks. As a relatively powerful and trusted entity $\mathcal{C}\mathsf{trl}$ can use various well-known vulnerability detection methods, e.g., fuzzing [47], static analysis [48], and even formal verification, to scrutinize $\mathcal{S}$ before authorizing it.

Physical Attacks: physical and hardware-focused attacks are considered out of scope. We assume that $\mathcal{A}\mathsf{dv}$ cannot modify code in ROM, induce hardware faults, or retrieve $\mathcal{D}\mathsf{ev}$'s secrets via side-channels that require $\mathcal{A}\mathsf{dv}$'s physical presence. Protection against such attacks can be obtained via standard physical security techniques [49]. This assumption is in line with related work on trusted hardware architectures for embedded systems [16], [18], [21], [20].

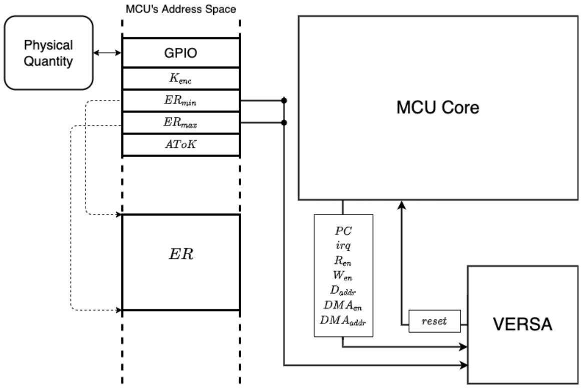


Fig. 4. VERSA Architecture

C. PfB *Game-based Definition*

Definition 4 starts by introducing an auxiliary predicate atomicExec. It defines whether a particular sequence of execution states (produced by the execution of some software $\mathcal{S}$) adheres to all necessary execution properties for *Atomic Sensing Operation Execution* discussed in Section III.

In atomicExec (in Definition 4.1), conditions 1-3 guarantee that a given $\mathcal{S}$ is executed as a whole and no external instruction is executed between its first and last instructions. Condition 4 assures that DMA is inactive during execution, hence protecting intermediate variables in DMEM against DMA tampering. Additionally, malicious interrupts could be leveraged to illegally change the control-flow of $\mathcal{S}$ during its execution. Therefore, condition 4 stipulates that both cases cause atomicExec to return $\perp$.

PfB-Game in Definition 4.2 models $\mathcal{A}$dv's capabilities by allowing it to execute any sequence of (polynomially many) instructions. This models $\mathcal{A}$dv's full control over software executed on the MCU, as well as its ability to use software to modify memory at will. It can also call Verify any (polynomial) number of times in an attempt to gain an advantage (e.g., learn something) from Verify executions.

To win the game, $\mathcal{A}$dv must succeed in executing some software that does not cause an MCU reset, and either: (1) is unauthorized, yet reads from GPIO, or (2) is authorized, yet violates atomicExec predicate conditions during its execution.

VI. VERSA: REALIZING PfB

VERSA runs in parallel with the MCU core and monitors a set of MCU signals: PC, D_{addr}, R_{en}, W_{en}, DMA_{en}, DMA_{addr}, and irq. It also monitors ER_{min} and ER_{max}, the boundary memory addresses of ER where $\mathcal{S}$ is stored; these are collectively referred to as "*METADATA*". VERSA hardware module detects privacy violations in real-time, based on aforementioned signals and *METADATA* values, causing an immediate MCU reset. Figure 4 shows the VERSA architecture. For quick reference, MCU signals and memory regions relevant to VERSA are summarized in Table I. To facilitate specification of VERSA properties, we introduce the following two macros:

$$Read\_Mem(i) \equiv (R_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$$
$$Write\_Mem(i) \equiv (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$$

TABLE I
NOTATION SUMMARY

Notation	Description
PC	Current program counter value
R_{en}	1-bit signal that indicates if MCU is reading from memory
W_{en}	1-bit signal that indicates if MCU is writing to memory
D_{addr}	Memory address of an MCU memory access
DMA_{en}	1-bit signal that indicates if DMA is active
DMA_{addr}	Memory address being accessed by DMA, when active
irq	1-bit signal that indicates if an interrupt is happening
$reset$	Signal that reboots the MCU when set to logic '1'
ER	A configurable memory region where the sensing operation $\mathcal{S}$ is stored, $ER = [ER_{min}, ER_{max}]$
METADATA	Metadata memory region; contains ER_{min} and ER_{max}
ATok	Fixed memory region from which Verify reads the authorization token when called
GPIO	Memory region that is mapped to GPIO port
VR	Memory region storing Verify code which instantiates *VRASED* software and its hardware protection
i_{Auth}	A fixed address in ROM, only be reachable (i.e., $PC = \mathsf{i}_{Auth}$) by a successful Verify call (i.e., Verify returns $\top$)
eKR	(Optional) memory region for the encryption key $\mathcal{K}_{enc}$ necessary to encrypt the $\mathcal{S}$ output (relevant to sensed data)

representing read/write from/to a particular memory address i by either CPU or DMA. For reads/writes from/to some continuous memory region (composed of multiple addresses) $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$, we instead say $D_{addr} \in \mathcal{M}$ to denote that $D_{addr} = i \ \wedge (i \geq \mathcal{M}_{min}) \wedge (i \leq \mathcal{M}_{max})$. The same holds for notation $DMA_{addr} \in \mathcal{M}$.

A. VERSA*: Construction*

Recall the key features of VERSA from Section III. To guarantee *Mandatory Sensing Operation Authorization* and *Atomic Sensing Operation Execution*, VERSA constructs PfB = (Authorize, Verify, XSensing) algorithms as in Construction 1. We describe each algorithm below.

Authorize**:**

To authorize $\mathcal{S}$, $\mathcal{C}$trl picks a monotonically increasing $\mathcal{C}$hal and generates ATok := $HMAC(KDF(\mathcal{C}\text{hal}, \mathcal{K}), \mathcal{S})$ (this follows *VRASED* authentication algorithm – see VERSA Verify specification below). ATok is computed over $\mathcal{S}$ with a one-time key derived from $\mathcal{K}$ and $\mathcal{C}$hal, where $\mathcal{K}$ is the master secret key shared between $\mathcal{C}$trl and $\mathcal{D}$ev.

Verify**:**

To securely verify that an executable $\mathcal{S}$', installed in ER, matches authorized $\mathcal{S}$, $\mathcal{D}$ev invokes *VRASED*[4] to compute σ := $HMAC(KDF(\mathcal{C}\text{hal}, \mathcal{K}), \mathcal{S}')$. Verify outputs $\top$, iff $\sigma =$ ATok. In this case, PC reaches a fixed address, called i_{Auth}. Otherwise, Verify outputs $\perp$.

In the rest of this section, we use "authorized software" to refer to software located in ER, for which Verify(ER, ATok) outputs $\top$. Whereas, "unauthorized software" refers to any software for which Verify(ER, ATok) outputs $\perp$.

XSensing**:**

---

[4] $\mathcal{D}$ev and $\mathcal{C}$trl act as $\mathcal{P}$rv and $\mathcal{V}$rf in *VRASED* respectively.

Construction 1. *VERSA instantiates a* PfB = [Authorize, Verify, XSensing] *scheme as follows:*
– $\mathcal{K}$ *is a symmetric key pre-shared between* $\mathcal{C}$trl *and VRASED secure architecture in* $\mathcal{D}$ev*;*

1) $\mathsf{Authorize}^{\mathcal{C}\mathsf{trl}}(\mathcal{S})$*:* $\mathcal{C}$trl *produces an authorization message* $\mathsf{M} := (\mathcal{S}, \mathcal{C}\mathsf{hal}, \mathsf{ATok})$*, where* $\mathcal{S}$ *is a software, i.e., a sequence of instructions* $\{\mathsf{i}_1, ..., \mathsf{i}_n\}$*, that* $\mathcal{C}$trl *wants to execute on* $\mathcal{D}$ev*;* $\mathcal{C}$hal *is a monotonically increasing challenge; and* ATok *is an authentication token computed as below.* $\mathcal{C}$trl *sends* M *to* $\mathcal{D}$ev*. Upon receiving* M*,* $\mathcal{D}$ev *is expected to parse* M*, find the memory region for* $\mathcal{S}$*, and execute* Verify *(see below).*

$$\mathsf{ATok} := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S}) \tag{7}$$

2) $\mathsf{Verify}^{\mathcal{D}\mathsf{ev}}(ER, \mathsf{ATok}, \mathcal{C}\mathsf{hal})$*: calls VRASED functionality [18] on memory region* $ER := [ER_{min}, ER_{max}]$ *to securely compute:*

$$\sigma := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), ER) \tag{8}$$

If $\sigma = \mathsf{ATok}$*, output* $\top$*; Otherwise, output* $\perp$*.*

3) $\mathsf{XSensing}^{\mathcal{D}\mathsf{ev}}(ER)$*: starts execution of software in* ER *by jumping to* ER_{min} *(i.e., setting* $PC = ER_{min}$*). A benign call to* XSensing *with input* ER *is expected to occur after one successful computation of* Verify *for the same* ER *region and contents therein. Otherwise,* **VERSA** *hardware support (see below) will cause the MCU to reset when* **GPIO** *is read.* XSensing *produces* $\mathsf{E} := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$*, the set of states produced by executing* ER*, and outputs* $\top$ *or* $\perp$ *as follows:*

$$\mathsf{XSensing}(ER) = \begin{cases} (\mathsf{E}, \top), & \text{if } \exists \mathsf{s} \in \mathsf{E} \text{ such that } (\mathsf{s} \in READ_{\mathsf{GPIO}}) \land (\mathsf{s} \notin RESET) \\ (\mathsf{E}, \perp), & \text{otherwise} \end{cases} \tag{9}$$

4) HardwareMonitor*: At all times,* **VERSA** *verified hardware enforces all following LTL properties :*

A – Read-Access Control to GPIO**:**

$$\mathbf{G}: \ \{(Read\_Mem(\mathsf{GPIO}) \land \neg(PC \in ER)) \rightarrow reset\} \tag{10}$$

$$\mathbf{G}: \ \{[(PC = ER_{max}) \lor reset] \rightarrow (\neg Read\_Mem(\mathsf{GPIO}) \lor reset) \ \mathbf{W} \ (PC = \mathsf{i}_{Auth})\} \tag{11}$$

B – Ephemeral Immutability of ER **and** ***METADATA***

$$\mathbf{G}: \ \{(PC = \mathsf{i}_{Auth}) \land (Write\_Mem(ER) \lor Write\_Mem(METADATA)) \rightarrow reset\} \tag{12}$$

$$\mathbf{G}: \{((Write\_Mem(ER) \lor Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(\mathsf{GPIO}) \lor reset) \ \mathbf{W} \ (PC = \mathsf{i}_{Auth}))\} \tag{13}$$

$$\text{[Optional] } \mathbf{G}: \{((Write\_Mem(ER) \lor Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(eKR) \lor reset) \ \mathbf{W} \ (PC = \mathsf{i}_{Auth}))\} \tag{14}$$

C – Atomicity and Controlled Invocation of ER**:**

$$\mathbf{G}: \ \{\neg reset \land (PC \in ER) \land \neg \mathbf{X}(PC \in ER) \rightarrow (PC = ER_{max}) \lor \mathbf{X}(reset)\} \tag{15}$$

$$\mathbf{G}: \ \{\neg reset \land \neg(PC \in ER) \land \mathbf{X}(PC \in ER) \rightarrow \mathbf{X}(PC = ER_{min}) \lor \mathbf{X}(reset)\} \tag{16}$$

$$\mathbf{G}: \ \{(PC \in ER) \land (irq \lor DMA_{en}) \rightarrow reset\} \tag{17}$$

[Optional] Read/Write-Access Control to Encryption Key ($\mathcal{K}_{enc}$**) in** eKR**:**

$$\mathbf{G}: \ \{(Read\_Mem(eKR) \land \neg(PC \in ER)) \rightarrow reset\} \tag{18}$$

$$\mathbf{G}: \ \{[(PC = ER_{max}) \lor reset] \rightarrow (\neg Read\_Mem(eKR) \lor reset) \ \mathbf{W} \ (PC = \mathsf{i}_{Auth})\} \tag{19}$$

$$\mathbf{G}: \ \{[Write\_Mem(eKR) \land \neg(PC \in VR)] \rightarrow reset\} \tag{20}$$

Remark: *[Optional] properties are needed only if support for encryption of outputs is desired.*

Fig. 5. Verified Remote Sensing Authorization (VERSA) Scheme

When XSensing (ER) is invoked, PC jumps to ER_{min}, and starts executing the code in ER. It produces a set E of states by executing ER, and outputs $\top$, if there is at least one state that reads GPIO without triggering an MCU reset. Otherwise, it outputs $\perp$.

HardwareMonitor:

VERSA HardwareMonitor is verified to enforce LTL specifications (10)–(20) in Construction 1.

A – Read-Access Control to GPIO is jointly specified by LTLs (10) and (11). LTL (10) states that GPIO can only be read during execution of ER ($PC \in ER$), requiring an MCU reset otherwise. LTL (11) forbids all GPIO reads (even those within ER execution) before successful computation of Verify on ER binary using a valid ATok. Successful Verify computation is captured by condition $PC = \mathsf{i}_{Auth}$. A new successful computation of Verify(ER, ATok) is necessary whenever ER execution completes ($PC = ER_{max}$) or after reset/boot. Hence, each legitimate ATok can be used to authorize ER execution once.

B – Ephemeral Immutability of ER and *METADATA* is specified by LTLs (12)-(14). From the time when ER binary is authorized until it starts executing, no modifications to ER or *METADATA* are allowed. LTL (12) specifies that no such modification is allowed at the moment when verification succeeds ($PC = \mathsf{i}_{Auth}$); LTL (13) requires ER to be re-authorized from scratch if ER or *METADATA* are ever modified. Whenever these modifications are detected ($Write\_Mem(ER) \lor Write\_Mem(METADATA)$) further reads to GPIO are immediately blocked ($\neg Read\_Mem(\mathsf{GPIO}) \lor reset$) until subsequent re-authorization of ER is completed (... $\mathbf{W}$ ($PC = \mathsf{i}_{Auth}$)). LTL (14) specifies the same requirement in order to read VERSA-provided encryption key ($\mathcal{K}_{enc}$) which is stored in memory region eKR. This property is only required when support for encryption of outputs is desired.

C – Atomicity & Controlled Invocation of ER are enforced by LTLs (15), (16), and (17). They specify that ER execution must start at ER_{min} and end at ER_{max}. Specifically, they use the relation between *current* and *next* PC values. The only legal PC transition from currently outside of ER to next inside ER is via $PC = ER_{min}$. Similarly, the only legal PC transition from currently inside ER to next outside ER is via $PC = ER_{max}$. All other cases trigger an MCU reset. In addition, LTL (17) requires an MCU reset whenever interrupts or DMA activity is detected during ER execution. This is done by simply checking irq and DMA_{en} signals.

We note that XSensing relies on the HardwareMonitor to reset the MCU when violations to ER atomic execution are detected. Upon reset all data is erased from memory. However, when execution of $\mathcal{S}$ completes successfully VERSA does not trigger resets. In this case, $\mathcal{S}$ is responsible for erasing its own stack before completion (reaching of ER its last instruction). We discuss how this self-clean-up routine can be implemented as a part of $\mathcal{S}$ behavior in Appendix A.

B. Encryption & Integrity of ER Output

As mentioned in Section III, after reading and processing `GPIO` inputs, $\mathcal{S}$ might need to encrypt and send the result to $\mathcal{C}$trl. VERSA supports encryption of this output, regardless of the underlying encryption scheme. For that purpose, Verify implementation derives a fresh one-time encryption key ($\mathcal{K}_{enc}$) from $\mathcal{K}$ and $\mathcal{C}$hal. To assure confidentiality of $\mathcal{K}_{enc}$, the following properties are required for the memory region (eKR) reserved to store $\mathcal{K}_{enc}$:

1) eKR is writable only by Verify (i.e., $PC \in VR$); and
2) eKR is readable only by ER after authorization.

LTLs (18)-(20) and (14) specify the confidentiality requirements of $\mathcal{K}_{enc}$. In sum, these properties establish the same read access-control policy for eKR and `GPIO` regions. Therefore, only authorized $\mathcal{S}$ is able to retrieve $\mathcal{K}_{enc}$.

VII. VERIFIED IMPLEMENTATION & SECURITY ANALYSIS

A. Sub-module Implementation & Verification

VERSA sub-modules are represented as FSMs and individually verified to hold for LTL properties from Construction 1. They are implemented in Verilog HDL as Mealy machines, i.e., their output is determined by both their current state and current inputs. Each FSM has a single output: a local $reset$. VERSA global output $reset$ is given by the disjunction (logic OR) of all local $reset$-s. For simplicity, instead of explicitly representing the output $reset$ value for each state, we use the following convention:

1) $reset$ is 1 whenever an FSM transitions to $RESET$ state;
2) $reset$ remains 1 while on $RESET$ state;
3) $reset$ is 0 otherwise.

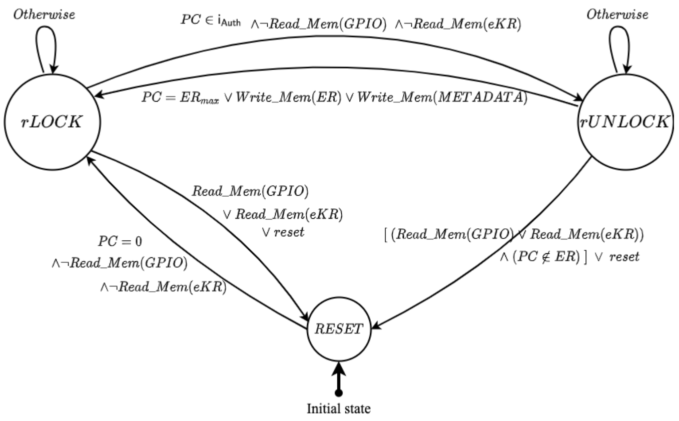

Otherwise
$PC \in i_{Auth}$ $\wedge\neg Read\_Mem(GPIO)$ $\wedge\neg Read\_Mem(eKR)$
Otherwise
$PC = ER_{max} \vee Write\_Mem(ER) \vee Write\_Mem(METADATA)$
$rLOCK$
$rUNLOCK$
$Read\_Mem(GPIO)$
$\vee Read\_Mem(eKR)$
$\vee reset$
$PC = 0$
$\wedge\neg Read\_Mem(GPIO)$
$\wedge\neg Read\_Mem(eKR)$
$[(Read\_Mem(GPIO) \vee Read\_Mem(eKR))$
$\wedge (PC \notin ER)] \vee reset$
$RESET$
Initial state

Fig. 6. Verified FSM for `GPIO` and eKR Read-Access Control (LTL (10)-(14) & LTL (18)-(19))

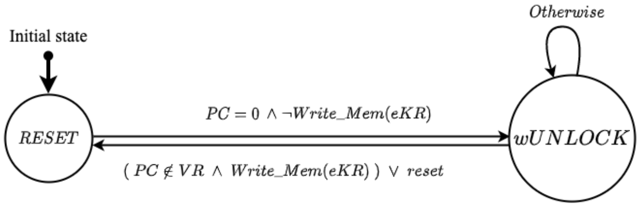

Initial state
Otherwise
$RESET$
$PC = 0 \wedge \neg Write\_Mem(eKR)$
$wUNLOCK$
$(PC \notin VR \wedge Write\_Mem(eKR)) \vee reset$

Fig. 7. Verified FSM for eKR Write-Access Control (LTL (20))

Note that all FSMs remain in $RESET$ state until $PC = 0$ which indicates that the MCU reset routine finished.

Fig. 6 illustrates the VERSA sub-module that implements read-access control to `GPIO` and eKR (when applicable). It guarantees that such reads are only possible when they emanate from execution of authorized software $\mathcal{S}$ contained in ER. It also assures that no modifications to ER or *METADATA* occur between authorization of $\mathcal{S}$ and its subsequent execution. The Verilog implementation of this FSM is formally verified to adhere to LTLs (10)-(14) and (18)-(19). It has 3 states: (1) $rLOCK$, when reads to `GPIO` (and possibly eKR) are disallowed; (2) $rUNLOCK$, when such reads are allowed to ER; and (3) $RESET$. The initial state (after reset or boot) is $RESET$, and it switches to $rLOCK$ state when $PC = 0$. It switches to $rUNLOCK$ when $PC = \mathsf{i}_{Auth}$ (with no reads to `GPIO` and eKR), indicating that Verify was successful. Note that $rUNLOCK$ transitions to $RESET$ when reads are attempted from outside ER, thus preventing reads by any unauthorized software. Once PC reaches ER_{max}, indicating that ER execution has finished, the FSM transitions back to $rLOCK$. Also, any attempted modifications to *METADATA* or ER in $rUNLOCK$ state bring the FSM back to $rLOCK$. Note that $rUNLOCK$ is only reachable after authorization of ER, i.e., $PC = \mathsf{i}_{Auth}$.

The FSM in Figure 7 enforces LTL (20) to protect eKR from external writes. It has two states: (1) $wUNLOCK$, when writes to eKR are allowed; and (2) $RESET$. At boot/after reset ($PC = 0$), this FSM transitions from $RESET$ to $wUNLOCK$. It transitions back to $RESET$ state whenever

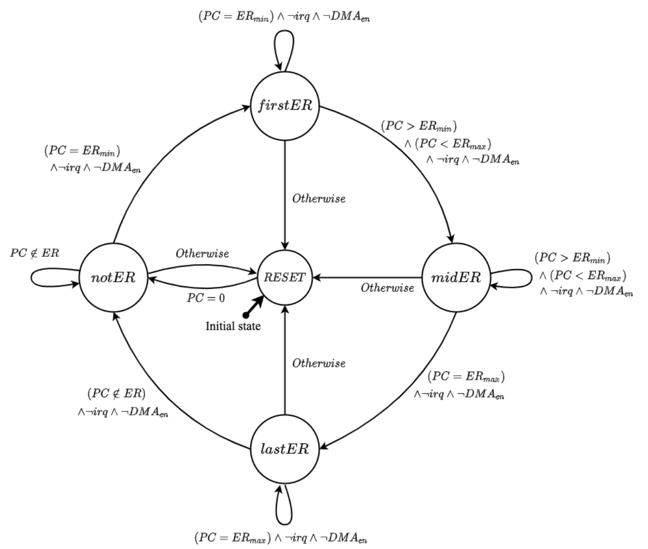

$(PC = ER_{min}) \land \neg irq \land \neg DMA_{en}$
$firstER$
$(PC > ER_{min}) \land (PC < ER_{max}) \land \neg irq \land \neg DMA_{en}$
$(PC = ER_{min}) \land \neg irq \land \neg DMA_{en}$
Otherwise
$PC \notin ER$
$notER$
Otherwise
$RESET$
$midER$
Otherwise
$(PC > ER_{min}) \land (PC < ER_{max}) \land \neg irq \land \neg DMA_{en}$
$PC = 0$
Initial state
Otherwise
$(PC = ER_{max}) \land \neg irq \land \neg DMA_{en}$
$(PC \notin ER) \land \neg irq \land \neg DMA_{en}$
$lastER$
$(PC = ER_{max}) \land \neg irq \land \neg DMA_{en}$

Fig. 8. ER Atomicity and Controlled Invocation FSM (LTL (15)-(17))

writes to eKR are attempted, unless these writes come from Verify execution ($PC \in VR$).

Figure 8 shows the FSM verified to enforce ER atomicity and controlled invocation: LTLs (15)-(17). It has five states; $notER$ and $midER$ correspond to PC being outside and within ER (not including ER_{min} and ER_{max}), respectively. $firstER$ and $lastER$ are states in which PC points to ER_{min} and ER_{max}, respectively. The only path from $notER$ to $midER$ is via $firstER$. Likewise, the only path from $midER$ to $notER$ is via $lastER$. The FSM transitions to $RESET$ whenever PC transitions do not follow aforementioned paths. It also transitions to $RESET$ (from any state other than $notER$) if irq or DMA_{en} signals are set.

B. Sub-module Composition and VERSA End-To-End Security

To demonstrate security of VERSA according to Definition 4, our strategy is two-pronged:

- **A)** We show that LTL properties from Construction 1 are sufficient to imply that GPIO (and eKR) is only readable by $\mathcal{S}$ and any XSensing operation that returns $\top$ (i.e., performs sensing) is executed atomically. The former is formally specified in Definition 6, and the latter in Definition 5. For this part, we write an LTL computer proof using SPOT LTL proof assistant [46].
- **B)** We use a cryptographic reduction to show that, as long as item **A** holds, *VRASED* security can be reduced to VERSA security according to Definition 4.

The intuition for this strategy is that, to win PfB-game in Definition 4, $\mathcal{A}$dv must either break the atomicity of XSensing (which is in direct conflict with Definition 5) or execute XSensing with unauthorized software and read GPIO without causing an MCU $reset$. Definition 6 guarantees that the latter is not possible without a prior successful call to Verify. On the other hand, Verify is implemented using *VRASED* verified architecture, which guarantees the unforgeability of ATok. Hence, breaking VERSA requires either violating VERSA verified guarantees or breaking *VRASED* verified guarantees, which should be infeasible to any PPT $\mathcal{A}$dv.

Appendix C includes proofs for Theorems 1, 2, and 3, in accordance to this proof strategy. The rest of this section focuses on VERSA end-to-end implementation goals captured by LTLs in Definitions 5 and 6 as well as their relation to VERSA high-level features discussed in Section III.

Theorem 1. *Definition 2 $\land$ LTL 15, 16, 17 $\rightarrow$ Definition 5*
Theorem 2. *Definition 2 $\land$ LTL 10, 11, 12, 13, 16, 20 $\rightarrow$ Definition 6*

Theorem 3. *VERSA is secure according to the PfB-game in Definition 4, as long as VRASED is a secure $\mathcal{R}$A architecture according to VRASED security game from [18].*

[Definition 5] states that it **globally** (always) holds that ER is atomically executed with controlled invocation. That is, whenever an instruction in ER executes ($PC \in ER$), it keeps executing instructions within ER ($PC \in ER$), with no interrupts and no DMA enabled, **until** PC reaches the last instruction in ER_{max} or an MCU reset occurs. Also, if an instruction in ER starts to execute, it always begins with the first instruction in ER_{min}. This formally specifies the *Atomic Sensing Operation Execution* feature discussed in Section III.

[Definition 6] **globally** requires that whenever GPIO is successfully read (i.e., without a $reset$), this read must come from the CPU while ER is being executed. In addition, **before** this read operation, the following must have happened at least once:

(1) Verify succeeded (i.e., $PC = i_{Auth}$);
(2) From the time when $PC = i_{Auth}$ **until** ER starts executing (i.e., $PC = ER_{min}$), no modification to ER and *METADATA* occurred; and
(3) If there was any write to eKR from the time when $PC = i_{Auth}$, **until** $PC = ER_{min}$, it must have been from Verify, i.e., while $PC \in VR$.

This formally specifies the intended behavior of the *Mandatory Sensing Operation Authorization* feature, discussed in Section III.

VIII. EVALUATION & DISCUSSION

In this section, we discuss VERSA implementation details and evaluation. VERSA source code and verification/proofs are publicly available at [33]. Evaluation of verification costs and discussion of VERSA limitations are deferred to Appendix D.

A. Toolchain & Prototype Details

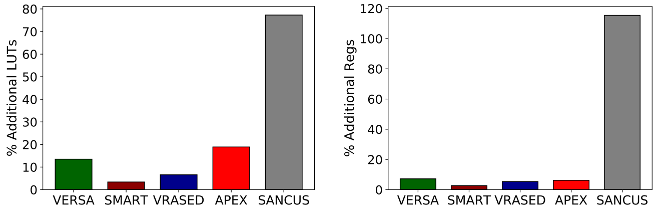
VERSA is built atop OpenMSP430 [36]: an open source implementation of TI-MSP430 [50]. We use Xilinx Vivado to synthesize an RTL description of HardwareMonitor and deploy it on Diligent Basys3 prototyping board for Artix7 FPGA. For the software part (mostly to implement Verify), VERSA extends *VRASED* software (which computes $HMAC$ over $\mathcal{D}$ev memory) to include a comparison with the received ATok (See Section VIII-C for extension details). We use the NuSMV

Definition 5. *Atomic Sensing Operation Execution:*

$$\begin{aligned}&\boldsymbol{G}\{\ (PC \in ER) \rightarrow [(PC \in ER) \land \neg irq \land \neg DMA_{en}]\ \boldsymbol{W}\ [(PC = ER_{max}) \lor reset]\ \}\\ \land\quad &\boldsymbol{G}\{\ \neg reset \land \neg(PC \in ER) \land \boldsymbol{X}(PC \in ER) \rightarrow \boldsymbol{X}(PC = ER_{min}) \lor \boldsymbol{X}(reset)\ \}\end{aligned}$$

Definition 6. *Mandatory Sensing Operation Authorization:*

$$\begin{aligned}&\boldsymbol{G}\{\ (Read\_Mem(\texttt{GPIO}) \land \neg reset) \rightarrow (PC \in ER)\ \} \land\\ &\Big\{(PC = \mathsf{i}_{Auth}) \land \{(PC = \mathsf{i}_{Auth}) \rightarrow [\neg Write\_Mem(ER) \land \neg Write\_Mem(\mathit{METADATA}) \land (Write\_Mem(eKR) \rightarrow (PC \in VR))]\ \boldsymbol{U}\ (PC = ER_{min})\Big\}\\ &\Big\}\ \boldsymbol{B}\ \big\{Read\_Mem(\texttt{GPIO}) \land \neg reset\big\}\end{aligned}$$



(a) Additional HW overhead (%) in Number of Look-Up Tables

(b) Additional HW overhead (%) in Number of Registers

Fig. 9. Hardware overhead comparisons with other low-end security architectures.

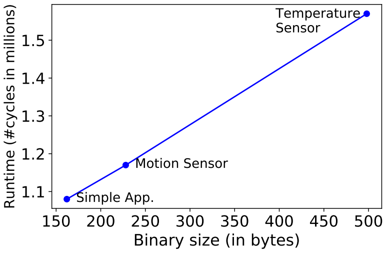


Fig. 10. Runtime overhead of VERSA due to Verify

model checker to formally verify that HardwareMonitor implementation adheres to LTL specifications (10)-(20). See Appendix D for details on the verification setup and costs.

B. Hardware Overhead

Table II reports on VERSA hardware overhead, as compared to unmodified OpenMSP430 and *VRASED*. Similar to other schemes [18], [22], [17], [16], we consider hardware overhead in terms of additional Look-Up Tables (LUTs) and registers. Extra hardware in terms of LUTs gives an estimate of additional chip cost and size required for combinatorial logic, while extra hardware in terms of registers gives an estimate of memory overhead required by sequential logic in VERSA FSMs. Compared to *VRASED*, VERSA requires 10% additional LUTs and 2% additional registers. In actual numbers, it adds 255 LUTs and 50 registers to the underlying MCU as shown in Table II.

TABLE II

HARDWARE OVERHEAD AND VERIFICATION COST

Architecture	Hardware		Reserved	Verification			
	LUTs	Regs	RAM (bytes)	LoC	#(LTLs)	Time (s)	RAM (MB)
OpenMSP430	1854	692	0	-	-	-	-
VRASED	1891	724	2332	481	10	0.4	13.6
VERSA + *VRASED*	2109	742	2336	1118	21	13956.4	1059.1

C. Runtime Overhead

VERSA requires any software piece seeking to access `GPIO` (and $\mathcal{K}_{enc}$) to be verified. Consequently, runtime overhead is due to Verify computation which instantiates *VRASED*. This runtime includes: (**1**) time to compute σ from equation 8; (**2**) time to check if $\sigma = \mathsf{ATok}$; and (**3**) time to write $\mathcal{K}_{enc}$ to eKR, when applicable. Naturally the runtime overhead is dominated by the computation in **(1)** which is proportional to the size of ER.

We measure Verify cost on three sample applications: (1) Simple Application, which reads 32-bytes of `GPIO` input and encrypts it using One-Time-Pad (OTP) with $\mathcal{K}_{enc}$; (2) Motion Sensor – available at [51] – which continuously reads `GPIO` input to detect movements and actuates a light source when movement is detected; and (3) Temperature Sensor – adapted code from [52] to support encryption of its outputs – which reads ambient temperature via `GPIO` and encrypts this reading using OTP. We prototype using OTP for encryption for the sake of simplicity noting that VERSA does not mandate a particular encryption scheme. All of these sample applications also include a self-clean-up code executed immediately before reaching their exit point to erase their stack traces once their execution is over.

Figure 10 shows Verify runtimes on these applications. Assuming a clock frequency of 10MHz (a common frequency for low-end MCUs), Verify runtime ranges from $100 - 200$ milliseconds for these applications. The overhead is linear on the binary size.

D. Comparison with Other Low-End Architectures:

To the best of our knowledge, VERSA is the first architecture related to PfB. However, to provide a point of reference in terms of performance and overhead, we compare VERSA with other low-end trusted hardware architectures, such as SMART [16], VRASED [18], APEX [22], and SANCUS [17]. All these architectures provide $\mathcal{R}$A-related services to attest integrity of software on $\mathcal{D}$ev either statically or at runtime. Since PfB also checks software integrity before granting access to `GPIO`, we consider these architectures to be related to VERSA.

Figure 9 compares VERSA hardware overhead with the aforementioned architectures in terms of additional LUTs and registers. Percentages are relative to the plain MSP430 core total cost.

VERSA builds on top of *VRASED*. As such, it is naturally more expensive than hybrid $\mathcal{R}$A architectures such as SMART and *VRASED*. Similar to VERSA, APEX also monitors execution properties and also builds on top of $\mathcal{R}$A (in APEX case, with the goal of producing proofs of remote software execution). Therefore, VERSA and APEX exhibit similar overheads. SANCUS presents a higher cost because it implements $\mathcal{R}$A and isolation features in hardware.

IX. RELATED WORK

There is a considerable body of work (overviewed in Section I) on IoT/CPS privacy. However, to the best of our knowledge, this paper is the first effort specifically targeting PfB, i.e., sensor data privacy on potentially compromised MCUs. Nonetheless, prior work has proposed trusted hardware/software co-designs – such as VERSA – offering other security services. We overview them in this section.

Trusted components, commonly referred to as Roots of Trust (RoTs), are categorized as software-based, hardware-based, or hybrid (i.e., based on hardware/software co-designs). Their usual purpose is to verify software integrity on a given device. Software-based RoTs [53], [54], [55], [56], [57], [58], [59] usually do not rely on any hardware modifications. However, they are insecure against compromises to the entire software state of a device (e.g., in cases where $\mathcal{A}$dv can physically re-program $\mathcal{D}$ev). In addition, their inability to securely store cryptographic secrets imposes reliance on strong assumptions about precise timing and constant communication delays to enable device authentication. These assumptions can be unrealistic in the IoT ecosystem. Nonetheless, software-based RoTs are the only viable choice for legacy devices that have no security-relevant hardware support. Hardware-based methods [60], [61], [62], [63], [64], [65], [17] rely on security provided by dedicated hardware components (e.g., TPM [61] or ARM TrustZone [66]). However, the cost of such hardware is normally prohibitive for low-end MCUs. Hybrid RoTs [16], [22], [18], [20], [21] aim to achieve security equivalent to hardware-based mechanisms, yet with lower hardware costs. They leverage minimal hardware support while relying on software to reduce additional hardware complexity.

Other architectures, such as SANCTUM [35] and Notary [67], provide strong memory isolation and peripheral isolation guarantees, respectively. These guarantees are achieved via hardware support or external hardware agents. They are also hybrid architectures where trusted hardware works in tandem with trusted software. However, we note that such schemes are designed for application computers that support MMUs and are therefore unsuitable for simple MCUs.

In terms of functionality, such embedded RoTs focus on integrity. Upon receiving a request from an external trusted *Verifier*, they can generate unforgeable proofs for the state of the MCU or that certain actions were performed by the MCU. Security services implemented by them include: (1) memory integrity verification, i.e., remote attestation [16], [17], [18], [19], [20], [21]; (2) verification of runtime properties, including control-flow and data-flow attestation [22], [23], [24], [25], [26], [27], [28], [29], [68]; and (3) proofs of remote software updates, memory erasure, and system-wide resets [30], [31], [32]. As briefly mentioned in Section I, due to their reactive nature, they can be used to *detect* whether $\mathcal{D}$ev has been compromised *after the fact*, but *cannot prevent* the compromised entity from exfiltrating private sensor data. VERSA, on the other hand, enforces mandatory authorization before any sensor data access and thus prevents leakage even when a compromise has already happened.

Formalization and formal verification of RoTs for MCUs have gained attention due to the benefits discussed in Sections I and II. VRASED [18] implemented a verified hybrid $\mathcal{R}$A scheme. APEX [22] built atop VRASED to implement and formally verify an architecture for proofs of remote execution of attested software. PURE [30] implemented provably secure services for software update, memory erasure, and system-wide reset. Another recent result [69] formalized and proved the security of a hardware-assisted mechanism to prevent leakage of secrets through timing side-channels due to MCU interrupts. Inline with the aforementioned work, VERSA also formalizes its assumptions along with its goals and implements the first formally verified design assuring PfB.

X. CONCLUSIONS

We formulated the notion of Privacy-from-Birth (PfB) and proposed VERSA: a formally verified architecture realizing PfB. VERSA ensures that only duly authorized software can access sensed data even if the entire software state of the sensor is compromised. To attain this, VERSA enhances the underlying MCU with a small hardware monitor, which is shown sufficient to achieve PfB. The experimental evaluation of VERSA publicly available prototype [33] demonstrates its affordability on a typical low-end IoT MCU: TI MSP430.

ACKNOWLEDGMENTS

The authors sincerely thank S&P'22 reviewers. This work was supported by funding from NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs. Part of this work was conducted while the first author was at the University of California, Irvine.

REFERENCES

[1] R. H. Weber, "Internet of things–new security and privacy challenges," *Computer law & security review*, vol. 26, no. 1, pp. 23–30, 2010.

[2] A. Ukil, S. Bandyopadhyay, and A. Pal, "Iot-privacy: To be private or not to be private," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 123–124.

[3] H. Lin and N. W. Bergmann, "Iot privacy and security challenges for smart home environments," *Information*, vol. 7, no. 3, p. 44, 2016.

[4] S. Zheng, N. Apthorpe, M. Chetty, and N. Feamster, "User perceptions of smart home iot privacy," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–20, 2018.

[5] P. Porambage, M. Ylianttila, C. Schmitt, P. Kumar, A. Gurtov, and A. V. Vasilakos, "The quest for privacy in the internet of things," *IEEE Cloud Computing*, vol. 3, no. 2, pp. 36–45, 2016.

[6] A. L. M. Neto, A. L. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. Loureiro, D. F. Aranha, H. K. Patil *et al.*, "Aot: Authentication and access control for the entire iot device life-cycle," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, 2016, pp. 1–15.

[7] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler, "$JEDI$: Many-to-many end-to-end encryption and key delegation for iot," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1519–1536.

[8] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home devices," in *Network and Distributed Systems Security (NDSS) Symposium*, vol. 2020, 2020.

[9] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart (er) iot traffic shaping," *arXiv preprint arXiv:1812.00955*, 2018.

[10] N. Apthorpe, D. Reisman, and N. Feamster, "Closing the blinds: Four strategies for protecting smart home privacy from network observers," *arXiv preprint arXiv:1705.06809*, 2017.

[11] N. J. Apthorpe, D. Reisman, and N. Feamster, "A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic," *CoRR*, vol. abs/1705.06805, 2017. [Online]. Available: http://arxiv.org/abs/1705.06805

[12] Y. Cheng, X. Ji, X. Zhou, and W. Xu, "Homespy: Inferring user presence via encrypted traffic of home surveillance camera." in *ICPADS*, 2017, pp. 779–782.

[13] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir, "Inferring user routes and locations using zero-permission mobile sensors," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 397–413.

[14] A. S. A. Sukor, A. Zakaria, N. A. Rahim, L. M. Kamarudin, R. Setchi, and H. Nishizaki, "A hybrid approach of knowledge-driven and data-driven reasoning for activity recognition in smart homes," *Journal of Intelligent & Fuzzy Systems*, vol. 36, no. 5, pp. 4177–4188, 2019.

[15] M. Jin, R. Jia, and C. J. Spanos, "Virtual occupancy sensing: Using smart meters to indicate your presence," *IEEE Transactions on Mobile Computing*, vol. 16, no. 11, pp. 3264–3277, 2017.

[16] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.

[17] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, pp. 7:1–7:33, 2017. [Online]. Available: https://doi.org/10.1145/3079763

[18] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *USENIX Security*, 2019.

[19] M. Ammar, B. Crispo, and G. Tsudik, "Simple: A remote attestation approach for resource-constrained iot devices," in *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2020, pp. 247–258.

[20] F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, pp. 34:1–34:6. [Online]. Available: https://doi.org/10.1145/2744769.2744922

[21] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[22] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/nunes

[23] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[24] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 743–754. [Online]. Available: https://doi.org/10.1145/2976749.2978358

[25] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 24.

[26] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 384–391.

[27] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1433–1449.

[28] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," in *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[29] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices," 2021.

[30] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," 2019.

[31] M. Ammar and B. Crispo, "Verify&revive: Secure detection and recovery of compromised low-end embedded devices," in *Annual Computer Security Applications Conference*, 2020, pp. 717–732.

[32] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, 2018.

[33] "VERSA source code," https://github.com/sprout-uci/pfb, 2022.

[34] Intel, "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx.

[35] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[36] O. Girard, "openMSP430," 2009. [Online]. Available: https://opencores.org/projects/openmsp430

[37] John Wiley & Sons, Ltd, 2010, pp. 295–300. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470570517

[38] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS*, 2017.

[39] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," *CCS*, 2021.

[40] E. Aliaj, I. D. O. Nunes, and G. Tsudik, "GAROTA: generalized active root-of-trust architecture," *CoRR*, vol. abs/2102.07014, 2021. [Online]. Available: https://arxiv.org/abs/2102.07014

[41] D. W. Loveland, *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[42] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[43] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, 2002.

[44] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A tool for word-level verification," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[45] K. L. McMillan, "The smv system," in *Symbolic Model Checking*. Springer, 1993, pp. 61–85.

[46] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0—a framework for ltl and ω-automata manipulation," in *International Symposium on Automated Technology for Verification and Analysis*, 2016.

[47] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[48] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 95–110.

[49] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSI Design*, 2004.

[50] T. Instruments. Msp430 ultra-low-power sensing & measurement mcus. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.

[51] "Motion sensor code," https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/pir_motion_sensor.

[52] "Temperature sensor code," https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor.

[53] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*, 2003.

[54] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Research in Security and Privacy (S&P)*. Oakland, California, USA: IEEE, 2004, pp. 272–282.

[55] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems,” in *ACM SOSP*, 2005.

[56] A. Seshadri, M. Luk, and A. Perrig, “SAKE: Software attestation for key establishment in sensor networks,” in *DCOSS*, 2008.

[57] R. W. Gardner, S. Garera, and A. D. Rubin, “Detecting code alteration by creating a temporary memory bottleneck,” *IEEE TIFS*, 2009.

[58] Y. Li, J. M. McCune, and A. Perrig, “VIPER: Verifying the integrity of peripherals’ firmware,” in *ACM CCS*, 2011.

[59] V. D. Gligor and S. L. M. Woo, “Establishing software root of trust unconditionally.” in *NDSS*, 2019.

[60] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot — A coprocessor-based kernel runtime integrity monitor,” in *USENIX Security Symposium*, 2004.

[61] Trusted Computing Group., “Trusted platform module (tpm),” 2017. [Online]. Available: http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[62] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, “New results for timing-based attestation,” in *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 2012.

[63] D. Schellekens, B. Wyseur, and B. Preneel, “Remote attestation on legacy operating systems with trusted platform modules,” *Science of Computer Programming*, vol. 74, no. 1, pp. 13 – 22, 2008.

[64] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for tcb minimization,” in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 315–328.

[65] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient TCB reduction and attestation,” in *IEEE S&P ’10*, 2010.

[66] Arm Ltd., “Arm TrustZone,” https://www.arm.com/products/security-on-arm/trustzone, 2018.

[67] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, “Notary: A device for secure transaction approval,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 97–113. [Online]. Available: https://doi.org/10.1145/3341301.3359661

[68] M. Geden and K. Rasmussen, “Hardware-assisted remote runtime attestation for critical embedded systems,” in *2019 17th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2019, pp. 1–10.

[69] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, “Provably secure isolation for interruptible enclaved execution on small microprocessors,” in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020, pp. 262–276.

[70] “Msp430 flash memory characteristics,” https://www.ti.com/lit/an/slaa334b/slaa334b.pdf?ts=1638460551489, 2018.

[71] “Avr atmega 1284p 8-bit microcontroller,” http://ww1.microchip.com/downloads/en/DeviceDoc/doc8059.pdf, 2009.

[72] E. Aras, M. Ammar, F. Yang, W. Joosen, and D. Hughes, “Microvault: Reliable storage unit for iot devices,” in *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2020, pp. 132–140.

APPENDIX

A. Clean-up after Program Termination

While `VERSA` guarantees the confidentiality of sensing operations, it requires the authorized executable $\mathcal{S}$ to erase its own stack/heap before its termination. This ensures that unauthorized software can not extract and leak sensitive information from $\mathcal{S}$ execution and allocated data. Erasure in this case can be achieved via a single call to libc’s `memset` function with start address matching the base of $\mathcal{S}$ stack and size equal to the maximum size reached by $\mathcal{S}$ execution.

The maximum stack size can be determined manually by counting the allocated local variables in small and simple $\mathcal{S}$ implementations. To automatically determine this size in more complex $\mathcal{S}$ implementations, all functions called within $\mathcal{S}$ must update the highest point reached by their respective stacks. Figure 11 shows a sample application that reads 32 bytes of sensor data, encrypts this data using `VERSA` one-time key $\mathcal{K}_{enc}$, and cleans-up the stack thereafter. Line 12 is $\mathcal{S}$ entry point (ER_{min}). $\mathcal{S}$ first saves the stack pointer to `STACK_MIN` address. Then, the `application` function is called, in line 17. `application` implements $\mathcal{S}$ intended behavior. After the `application` is done, the clean up code (lines 51-53) is called with `STACK_MIN` as the start pointer and size of 32 + 4 bytes (32 bytes for `data` variable (in line 39) and 4 bytes for stack metadata).

```
1    #include <string.h>
2    // Sensor is at P3IN
3    #define P3IN          (*(volatile unsigned char *) 0x0018)
4    #define BIT4          0x10
5    #define HIGH          0x1
6    #define LOW           0x0
7    #define eKR           0x360
8    #define STACK_MIN     0x1010 // App stack start pointer is stored here
9    #define RESULT        0x1030 // App result is written here
10
11   // ER_min
12   __attribute__ ((section (".er.entry"), naked)) void applicationEntry() {
13       // Save the stack pointer to STACK_MIN at entry.
14       __asm__ volatile("mov    r1,   &0x1010" "\n\t");
15
16       // Call the application
17       application();
18
19       // Call the clean up code
20       cleanUp((uint8_t*)STACK_MIN, 32 + 4);
21
22       // Jump to applicationExit()
23       __asm__ volatile( "br #__er_leave" "\n\t");
24   }
25
26   __attribute__ ((section (".er.body"))) int digitalRead() {
27       if(P3IN & BIT4) return HIGH;
28       else return LOW;
29   }
30
31   __attribute__ ((section (".er.body"))) void encrypt(uint8_t* data, int
             data_size, uint8_t* key, int key_size) {
32       for (int i = 0; i < data_size; i++) {
33           data[i] = data[i] ^ key[i];
34       }
35   }
36
37   __attribute__ ((section (".er.body"))) void application() {
38       // Read sensor data
39       uint8_t data[32];
40       for (int i = 0; i < 32; i++) {
41           data[i] = digitalRead();
42       }
43
44       // One-Time-Pad encryption of data using the enc_key in eKR.
45       encrypt(data, 32, (uint8_t*)eKR, 32);
46
47       // Copy encrypted result to RESULT address.
48       memcpy(RESULT, data, 32);
49   }
50
51   __attribute__ ((section (".er.body"))) void cleanUp(uint8_t* start, int size){
52       memset((uint8_t*)start, 0, size);
53   }
54
55   //ER_max
56   __attribute__ ((section (".er.exit"), naked)) void applicationExit() {
57       __asm__ volatile("ret" "\n\t");
58   }
```

Fig. 11. Sample sensing operation that reads GPIO input, encrypts it, and cleans up its stack after execution.

B. Data Erasure on Reset/Boot

Violations to `VERSA` properties trigger an MCU reset. A reset immediately stops execution and prepares the MCU core to reboot by clearing all registers and pointing the program counter (PC) to the first instruction of PMEM. However, some MCUs may not guarantee erasure of DMEM as a part of this process. Therefore, traces of data allocated by $\mathcal{S}$ (including sensor data) could persist across resets.

In MCUs that do not offer DMEM erasure on reset, a software-base Data Erasure (DE) can be implemented and invoked it as soon as the MCU starts, i.e., as a part of the bootloader code. In particular, DE can be implemented using `memset` (similar to lines 51-53) with constant arguments matching the entirety of the MCU’s DMEM.DE should be immutable (e.g., stored in ROM) which is often the case for bootloader binaries. Upon reset, PC must always point to the first instruction of DE. The normal MCU start-up proceeds normally after DE execution is completed.

C. VERSA Composition Proof

In this section, we show that VERSA is a secure PfB architecture according to Definition 4, as long as **A)** the sub-properties in Construction 1 hold (Theorem 1, 2) and **B)** *VRASED* is a secure remote attestation ($\mathcal{R}$A) architecture according to the *VRASED* security definition in [18] (Theorem 3). Informally, part **A)** shows that if the machine model and all LTLs in Construction 1 hold, then the end-to-end goals for secure PfB architecture are met, while this does not include the goal of prevention of forging authorization tokens. Part **B)** handles the latter using a cryptographic reduction, i.e., it shows that an adversary able to forge the authorization token (with more than negligible probability) can also break *VRASED* according to the $\mathcal{R}$A-game, which is a contradiction assuming the security of *VRASED*. Therefore, Theorems 1-3 prove that VERSA is a secure PfB architecture as long as *VRASED* is a secure $\mathcal{R}$A architecture.

For part **A)**, computer-checked LTL proofs are performed using SPOT LTL proof assistant [46]. These proofs are available at [33]. We present the intuition behind them below.

Proof of Theorem 1 (Intuition). LTL (16) states the legal entry instruction requirement, while LTL (15) states the legal exit instruction requirement in atomicExec. Also, since LTL (15) states that ER_{max} is the only possible exit of the ER without a reset, it implies self-contained execution of ER. Lastly, LTL (17) enforces MCU reset if any interrupt or DMA occurs, which naturally prevents interrupts and DMA actions, as required by atomicExec. These imply the LTL in Definition 5 which stipulates that execution of ER must start with ER_{min} and stays within ER with no interrupts nor DMA actions, until PC reaches ER_{max} (causing a reset otherwise). □

Proof of Theorem 2 (Intuition). Definition 6 **(i)** requires at least one successful verification of ER before GPIO can be read successfully (without triggering a reset); and **(ii)** disallows modifications to ER, *METADATA*, and $\mathcal{K}_{enc}$ (other than by VR) in between ER verification subsequent ER execution. LTLs (10) and (11) state that PC must be within ER to read GPIO and disallow GPIO reads by default (including when MCU reset occurs) and after the execution of ER is over ($PC = ER_{max}$). Also, LTL (11) requires (re-)authorization ($PC = \mathsf{i}_{Auth}$) of ER after the execution of ER is over ($PC = ER_{max}$). LTL (13) disallows GPIO reads until the (re-)verification whenever ER or *METADATA* are written. LTL (12) disallows changes to ER and *METADATA* at the exact time when verification succeeds. LTL (16) guarantees that the execution of ER starts with ER_{min} and LTL (20) guarantees that only the VR code can modify the value in eKR. Thus, these are sufficient imply Definition 6. □

For part **B)**, we construct a reduction from the security game of *VRASED* in [18] to the security game of VERSA according to the Definition 4. i.e., The ability to break the PfB-game of VERSA allows to break the $\mathcal{R}$A-game of *VRASED*, and therefore, as long as *VRASED* is a secure $\mathcal{R}$A architecture according to the $\mathcal{R}$A-game, VERSA is secure according to the PfB-game.

Proof of Theorem 3. Denote by $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$, an adversary who can win the security game in Definition 4 against VERSA with more than negligible probability. We show that if such $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ exists, then it can be used to construct $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ that wins the $\mathcal{R}$A-game with more than negligible probability.

Recall that, to win the PfB-game, $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ must trigger $\top$ as a result of XSensing, which means it reads the sensed data without MCU reset. From the PfB-game step 4 in Definition 4, it can be done in either of the following two ways:

`Case1.` $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ executes a new, unauthorized software $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ which causes $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}) \to \top$; or

`Case2.` $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ breaks the atomic execution of an authorized, but not yet executed software, $\mathcal{S}_j$, so that it causes $\mathsf{XSensing}(\mathcal{S}_j) \to (E, \top)$ such that $\mathsf{atomicExec}(E, \mathcal{S}_j) \equiv \perp$.

Recall that for the instruction set I_j of $\mathcal{S}_j$ and a set E_j of execution states , to have $\mathsf{atomicExec}(I_j, E_j) \equiv \perp$, at least one of four requirements in Definition 4.1 must be false. Note that the atomic sensing operation execution goal in Definition 5 rules out the probability of `Case2`. Specifically, LTL (16) enforces 1), while 2) and 3) are guaranteed by LTL (15). Lastly, 4) is covered by LTL (17).

For `Case1`, i.e., to trigger $\top$ by running XSensing, $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ needs to read GPIO without causing an MCU reset. Recall that the *Mandatory Sensing Operation Authorization* in Definition 6 requires Verify (with input executable in ER) to succeed at least once before reading GPIO. According to VERSA construction, $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ causes $\mathsf{Verify}(ER, \mathsf{ATok}^*, \mathcal{C}\mathsf{hal}^*)$ to output $\top$, where ER contains $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ which is an unauthorized software, ATok^* is a valid issued (but never used) token, and $\mathcal{C}\mathsf{hal}^*$ is its corresponding challenge. Since Verify function is implemented using *VRASED* to compute HMAC of $\mathcal{C}\mathsf{hal}$ and ER, such $\mathcal{A}\mathsf{dv}_{\mathsf{PfB}}$ can be directly used as $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ to win the $\mathcal{R}$A-game of *VRASED*. Thus, assuming secure $\mathcal{R}$A architecture *VRASED*, this is a contradiction, which implies the security of VERSA according to the PfB-game. □

D. Extended Evaluation & Discussion

Verification Costs: Formal verification costs are reported in Table II. We use a Ubuntu 18.04 desktop machine running at 3.4GHz with 32GB of RAM for formal verification. Our verification pipeline converts Verilog HDL to SMV specification language and then verifies it against the LTL properties listed in Construction 1 using the NuSMV model checker (per Section II). VERSA verification requires checking 11 extra invariants – LTLs (10) to (20) – in addition to *VRASED* LTL invariants. It also incurs higher run-time and memory usage than *VRASED* verification. This is due to two additional 16-bit hardware signals (ER_{min}, ER_{max}) which increase the space of possible input combinations and thus the complexity of model checking process. However, verification is still manageable in a commodity desktop – it takes around 5 minutes and consumes 340MB of memory.

VERSA Limitations:

1) Shared Libraries: to verify $\mathcal{S}$, $\mathcal{C}$trl must ensure that $\mathcal{S}$ spans one contiguous memory region (ER) on $\mathcal{D}$ev. If any code dependencies exist outside of ER, VERSA will reset the MCU according to LTL (17). To preclude this situation, $\mathcal{S}$ must be made self-contained by statically linking all of its dependencies within ER.

2) Atomic Execution & Interrupts: per Definition 4, VERSA forbids interrupts during execution of XSensing. This can be problematic, especially on a $\mathcal{D}$ev with strict real-time constraints. In this case, $\mathcal{D}$ev must be reset in order to allow servicing the interrupt after DMEM erasure. This can cause a delay that could be harmful to real-time settings. Trade-offs between privacy and real-time constraints should be carefully considered when using VERSA. One possibility to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) themselves immutable and uninterruptible from the start of XSensing until its end; and (2) included in ER memory range and are thus checked by Verify.

3) Possible Side-channel Attacks: MSP430 and similar MCU-s allow configuring some GPIO ports to trigger interrupts. If one of such ports is used for triggering an interrupt, $\mathcal{A}$dv could possibly look at the state of $\mathcal{D}$ev and learn information about `GPIO` data. For example, suppose that a button press mapped to a GPIO port triggers execution of a program that sends some fixed number of packets over the network. Then, $\mathcal{A}$dv can learn that the GPIO port was activated by observing network traffic. To prevent such attacks, privacy-sensitive quantities should always be physically connected to GPIO ports that are not interrupt sources (these are usually the vast majority of available GPIO ports). Other popular timing attacks related to cache side-channels and speculative execution, are not applicable to this class of devices, as these features are not present in low-end MCUs.

4) Flash Wear-Out: VERSA implements Verify using *VRASED*. As discussed in Section II-C, the authentication protocol suggested by *VRASED* requires persistent storage of the highest value of a monotonically increasing challenge/counter in flash. We note that flash memory has a limited number of write cycles (typically at least 10,000 cycles [70], [71]). Hence, a large number of successive counter updates may wear-out flash causing malfunction. In *VRASED* authentication, the persistent counter stored in flash is only updated following successful authentication of $\mathcal{C}$trl. Therefore, only legitimate requests from $\mathcal{C}$trl cause these flash writes, limiting the capability of an attacker to exploit this issue. Nonetheless, if the number of expected legitimate calls to Verify is high, one must select the persistent storage type or (alternatively) use different flash blocks once a given flash block storing the counter reaches its write cycles' limit. For a more comprehensive discussion of this matter, see [72].

VERSA Alternative Use-Case:

VERSA can be viewed as a general technique to control access to memory regions based on software authorization tokens. We apply this framework to `GPIO` in low-end MCUs. Other use-cases are possible. For example, a VERSA-like architecture could be used to mark a secure storage region and grant access only to explicitly authorized software. This could be useful if $\mathcal{D}$ev runs multiple (mutually distrusted) applications and data must be securely shared between subsets thereof.