PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in Low-End Embedded Systems

Ivan De Oliveira Nunes
University of California, Irvine
ivanoliv@uci.edu

Karim Eldefrawy
SRI International
karim.eldefrawy@sri.com

Norrathep Rattanavipanon
University of California, Irvine
nrattana@uci.edu

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

ABSTRACT

Remote Attestation ($\mathcal{R}$A) is a security service that enables a trusted verifier ($\mathcal{V}$rf) to measure current memory state of an untrusted remote prover ($\mathcal{P}$rv). If correctly implemented, $\mathcal{R}$A allows $\mathcal{V}$rf to remotely detect if $\mathcal{P}$rv's memory reflects a compromised state. However, $\mathcal{R}$A by itself offers no means of remedying the situation once $\mathcal{P}$rv is determined to be compromised. In this work we show how a secure $\mathcal{R}$A architecture can be extended to enable important and useful security services for low-end embedded devices. In particular, we extend the formally verified $\mathcal{R}$A architecture, *VRASED*, to implement provably secure software update, erasure, and system-wide resets. When (serially) composed, these features guarantee to $\mathcal{V}$rf that a remote $\mathcal{P}$rv has been updated to a functional and malware-free state, and was properly initialized after such process. These services are provably secure against an adversary (represented by malware) that compromises $\mathcal{P}$rv and exerts full control of its software state. Our results demonstrate that such services incur minimal additional overhead (0.4% extra hardware footprint, and 100-s milliseconds to generate combined proofs of update, erasure, and reset), making them practical even for the lowest-end embedded devices, e.g., those based on MSP430 or AVR ATMega micro-controller units (MCUs). All changes introduced by our new services to *VRASED* trusted components are also formally verified.

1 INTRODUCTION

In recent years, the number and variety of special-purpose computing devices increased dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS), and Internet-of-Things (IoT) gadgets, that are increasingly utilized in various "smart" settings, such as homes, offices, factories, automotive systems, and public venues. Despite their many benefits, such devices unfortunately also represent natural and attractive targets for attacks, especially, remote exploits and malware infestations. As society becomes accustomed to being surrounded by, and dependent on, such devices, their security becomes extremely important.

At the low-end of the spectrum, embedded devices are designed with strict constraints on cost, physical size, and energy consumption, i.e., ultra-low-power MCUs, such as TI MSP430 [1]. It is thus unrealistic to expect such devices to have sophisticated means (akin to those available for laptops or smartphones) to prevent malware injection. **Remote Attestation ($\mathcal{R}$A)** has recently emerged as an alternative and inexpensive mean to detect malware presence on remote low-end devices. $\mathcal{R}$A allows a trusted and more powerful verifier ($\mathcal{V}$rf) to remotely measure the software state of an untrusted remote prover ($\mathcal{P}$rv). As shown in Figure 1, $\mathcal{R}$A is typically realized as a simple challenge-response protocol:

(1) $\mathcal{V}$rf sends an attestation request with a (random) challenge (Chal) to $\mathcal{P}$rv. This request might also contain a token derived from a secret that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.
(2) $\mathcal{P}$rv receives the attestation request and computes an *authenticated integrity check* over its memory and Chal. The memory region to be attested might be either pre-defined, or explicitly specified in the request. In the latter case, authentication of $\mathcal{V}$rf in step (1) is important for overall security/privacy of $\mathcal{P}$rv, since the request can specify arbitrary memory regions.
(3) $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
(4) $\mathcal{V}$rf receives the result from $\mathcal{P}$rv, and checks whether it corresponds to a valid memory state.

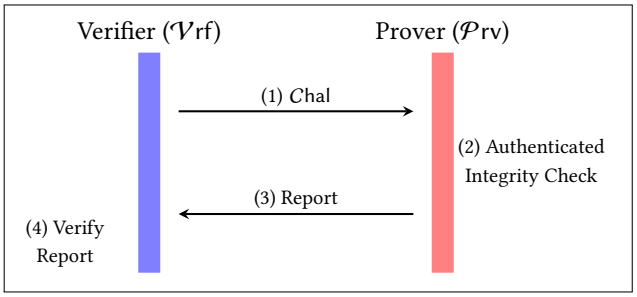


Figure 1: Typical $\mathcal{R}$A Protocol

The *authenticated integrity check* can be implemented as a Message Authentication Code (MAC) over $\mathcal{P}$rv's entire memory, or a specific range thereof. However, computing a MAC requires $\mathcal{P}$rv to have a unique secret key (denoted by $\mathcal{K}$) shared with $\mathcal{V}$rf. $\mathcal{K}$ must reside in secure storage, where it is **inaccessible** to any software running on $\mathcal{P}$rv, except for privileged attestation code. Since the usual $\mathcal{R}$A threat model assumes a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support. Hybrid RA (based on hardware/software co-design) is a particularly promising approach for low-end embedded devices. It aims to provide the same security guarantees as (more expensive) hardware-based $\mathcal{R}$A approaches (e.g., those based on TPMs [1], SGX [2] or

[1] http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/applications.html

TrustZone [3]), while minimizing modifications to underlying hardware.

Despite major progress and many proposed $\mathcal{R}$A architectures with different assumptions and guarantees [4–9], current RA approaches are limited in two major aspects: **First,** they assume that, if an invalid state is detected (either because of malware or errors), physical out-of-band measures must be taken to re-program and/or reset $\mathcal{P}$rv. This is a cumbersome requirement, and we show how it can be avoided by extending the $\mathcal{R}$A architecture to provide secure software update, memory erasure, and proofs of reset. **Second,** they typically do not offer formal guarantees or any kind of provable security. We argue that, before such architectures can be adopted on a large scale, all their security services and properties should stand on solid ground. To do so, we build upon a recently proposed formally verified *VRASED* $\mathcal{R}$A architecture. This allows us to prove security of every protocol and implementation proposed in this work. Whenever modifications to the Trusted Code Base (TCB) of the $\mathcal{R}$A architecture are required, they are formally verified. This way, security is attained for both the protocol and its implementation.

Our goal is to develop an architecture that assures $\mathcal{V}$rf that a remote $\mathcal{P}$rv has been brought to a functional malware-free state and has been properly initialized. Achieving this goal requires composing three services: (i) proof of software update to ensure $\mathcal{P}$rv's program memory re-configuration; (ii) proof of erasure of data memory, to ensure that no data from prior executions remain, including malware that can hide itself; and (iii) proof of system reset to ensure that – after program memory update and data memory erasure – $\mathcal{P}$rv's software state is reset and execution control flow is re-initialized. With this end-goal in mind, we propose a provably secure and formally verified architecture, called PURE: Proofs of Update, Reset, and Erasure. This work makes the following contributions:

– **New security services for low-end embedded systems:** we construct security services for proofs of remote system-wide reset, secure update and erasure by extending *VRASED* [9] – a formally verified hybrid $\mathcal{R}$A architecture. The overhead of our design is minimal. To the best of our knowledge, PURE is the first architecture that provides proofs of system-wide reset. Secure update based on $\mathcal{R}$A have been proposed for medium-end devices (capable of storing and running a secure micro-kernel) [10]. However, PURE is the first $\mathcal{R}$A-based secure update service for low-end devices and also the first that is provably secure.

– **Formal verification and provable security:** We implement reset, update and erasure services by extending *VRASED* formally verified $\mathcal{R}$A TCB. All changes to *VRASED* TCB are also formally verified, in order to yield end-to-end provable security. Resulting services are provably secure under the assumption of full compromise of $\mathcal{P}$rv's software state.

– **Public-domain implementation and evaluation:** All proposed services are implemented on a real-world low-end MCU platform (TI MSP430) and deployed using FPGAs. The entire design, along with its verification, is publicly available and is incorporated into *VRASED* open-source project [9]. Our evaluation demonstrates a low hardware footprint (0.4%), which we consider to be affordable even for low-end devices.

2 RELATED WORK

Remote Attestation ($\mathcal{R}$A): prior techniques fall into three categories: software-based, hardware-based, or hybrid. Security of software-based attestation [11–13] relies on strong assumptions about precise timing and/or constant communication delays, which are mostly unrealistic in the IoT ecosystem. Hardware-based methods [14–16] rely on security provided by dedicated hardware components, e.g., TPMs [1]. However, the cost of such hardware is prohibitive for low-end devices. Hybrid $\mathcal{R}$A [6, 7, 9] aims to achieve security equivalent to hardware-based mechanisms, yet with lower hardware cost. It imposes minimal hardware requirements, while relying on software to minimize the complexity of the additional hardware.

Formally Verified Security Services: In recent years, several efforts focused on formal verification of security services. In terms of cryptographic primitives, Hawblitzel et al. [17] verified implementations of SHA, HMAC, and RSA. Beringer et al.[18] verified the Open-SSL SHA-256 implementation. Bond et al. [19] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. More recently, Zinzindohoué, et al. [20] presented HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [21]. Also, larger security-critical systems have been successfully verified, e.g., [22]. CompCert[23] is a C compiler formally verified to preserve C code semantics in generated assembly code. Klein et al. [24] designed and proved functional correctness of seL4 – the first verified general-purpose microkernel. Cabodi et al. [25, 26] took the first steps towards formalizing hybrid $\mathcal{R}$A properties. Later, *VRASED* [9] realized a formally verified hybrid $\mathcal{R}$A architecture.

3 BACKGROUND

We now overview relevant formal verification methods, and *VRASED* which serves as our main building block.

3.1 Formal Verification, Model Checking & LTL

Formal verification typically involves three basic 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 must satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system satisfies them. This checking can be achieved via either Theorem Proving or Model Checking. We use the latter to verify the implementation of system modules, and the former to derive new properties from sub-properties that are proved for the modules' implementation.

One approach to perform model checking is to specify properties as *formulae* using Linear Temporal Logic (LTL) and system models as FSMs. Hence, a system is represented by a triple: (S, S_0, T), where S is a finite set of states, $S_0 \subseteq S$ is the set of possible initial states, and $T \subseteq S \times S$ is the transition relation set, which describes the set of states reachable in a single step from each state. The use of LTL to specify properties allows representation of expected system behavior over time.

We apply the model checker NuSMV [27], which can be used to verify generic HW or SW 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 model specification is simple. Furthermore, it can be automated [28], since the standard RTL design already relies on describing hardware as an FSM.

In NuSMV, properties are specified in Linear Temporal Logic (LTL), which is particularly useful for verifying sequential systems, because it extends common logic statements with temporal clauses. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL includes temporal connectives, thus enabling sequential reasoning. We are interested in the following temporal connectives:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – Future ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
- $\phi\ \mathbf{U}\ \psi$ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- $\phi\ \mathbf{W}\ \psi$ – ϕ Weak Until ψ: holds if ϕ holds for all states prior to the state when ψ holds. However, the state when ψ holds is not guaranteed to happen. In that case ϕ will remain true forever.

This set of temporal connectives combined with propositional connectives (with their usual meanings) allows us to specify powerful rules. NuSMV works by checking LTL specifications against the system FSM for all reachable states.

3.2 *VRASED* Overview

VRASED [9] is a formally verified hardware/software co-design for $\mathcal{R}$A targeting low-end devices. It is designed as a set of sub-modules, each guaranteeing a specific set of sub-properties. All sub-modules, both hardware and software, are individually verified. Finally, composition of all sub-modules is proved to achieve formal definitions of $\mathcal{R}$A soundness and security. $\mathcal{R}$A soundness guarantees that an integrity-ensuring function (HMAC in *VRASED* case) is correctly computed over attested memory (AR). Moreover, it guarantees that the content of AR can not be modified once $\mathcal{R}$A computation starts, thus protecting against "hide-and-seek" attacks caused by roving malware [29]. $\mathcal{R}$A security ensures that $\mathcal{R}$A execution generates an unforgeable authenticated memory measurement and that the secret key $\mathcal{K}$ is not leaked before, during, or after attestation.

Notation	Description
$fst(M)$ & $lst(M)$	First and last instructions in memory range M, respectively
TCB	ROM region storing SW-Att: $TCB = [fst(TCB), lst(TCB)]$
MR	(MAC RAM) RAM region where SW-Att computation result is written: $MR = [fst(MR), lst(MR)]$. The same region is used to pass the attestation challenge as input to SW-Att
AR	Attested Region, i.e., memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}$rf: $AR = [fst(AR), lst(AR)]$

Table 1: Notation for memory addresses

Figure 2 shows *VRASED* architecture and Table 1 summarizes the notation used throughout the rest of this paper. To achieve aforementioned goals, *VRASED* software (SW-Att in Figure 2) is stored in Read-Only Memory (ROM) and relies on a formally verified HMAC implementation from the HACL* cryptographic library [20]. A typical SW-Att execution is as follows:

(1) Read a challenge $\mathcal{C}$hal that is assumed to be stored in the memory region MR.
(2) Derive a one-time key from $\mathcal{C}$hal and the attestation master key $\mathcal{K}$.
(3) Generate an attestation token H by computing an HMAC over an attested memory region AR using the derived key. More formally, $\mathsf{H} = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), AR)$.
(4) Write H into MR and return the execution to an unprivileged Software, i.e, normal applications.

VRASED hardware, as shown in Figure 2, monitors 7 MCU signals:

- PC: Current Program Counter value;
- R_{en}: Indicates if the MCU is reading from memory (1-bit);
- W_{en}: Indicates if the MCU is writing to memory (1-bit);
- D_{addr}: Address for an MCU memory access;
- DMA_{en}: Indicates if Direct Memory Access (DMA) is currently enabled (1-bit);
- DMA_{addr}: Memory address being accessed by DMA.
- irq: Signal that indicates if an interrupt is happening (1-bit);

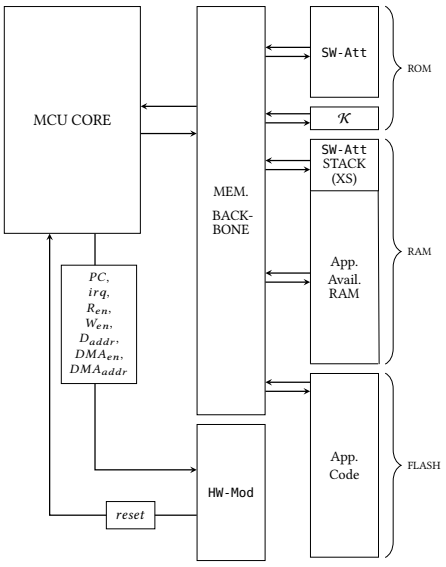


Figure 2: VRASED architecture (from [9])

These signals are used to determine a one-bit *reset* signal output. When set to 1, it triggers an immediate system-wide MCU reset, prior to the execution of the next instruction. *reset* is triggered when *VRASED* hardware detects any violation of security properties. *VRASED* hardware is described in Register Transfer Level (RTL) using Finite State Machines (FSMs). Then, NuSMV Model Checker [30] is used to automatically prove that FSMs achieve claimed security sub-properties. Finally, the proof that conjunction of hardware and software sub-properties implies end-to-end soundness and security is performed using an LTL theorem prover. More formally, *VRASED* end-to-end security proof guarantees that no Probabilistic Polynomial-Time (PPT) adversary can win the security game in Definition 1 with more than negligible probability in the security parameter. In the rest of this paper, we show (via reductions) how PURE proofs of update and erasure can be constructed in a provably secure manner entirely based on *VRASED* end-to-end security guarantees; whereas we leverage specific LTL sub-properties guaranteed by *VRASED* hardware to design a minimal formally verified architecture for proofs of reset.

Definition 1.
1.1 $\mathcal{RA}$ Security Game ($\mathcal{RA}$-game):
Notation:
- l is the security parameter and $|\mathcal{K}| = |Chal| = |MR| = l$
- $AR(t)$ denotes the content of AR at time t
$\mathcal{RA}$-game:

(1) `Setup:` *$\mathcal{A}$dv is given oracle access to* `SW-Att` *calls.*
(2) `Challenge:` *A random challenge $Chal \leftarrow \$\{0,1\}^l$ is generated and given to $\mathcal{A}$dv.*
(3) `Response:` *$\mathcal{A}$dv responds with a pair (M, σ), where σ is either forged by $\mathcal{A}$dv or the result of calling* `SW-Att` *at some arbitrary time t.*
(4) $\mathcal{A}$dv wins if and only if $M \neq AR(t)$ and $\sigma = HMAC(KDF(\mathcal{K}, Chal), M)$.

4 ADVERSARIAL MODEL & ASSUMPTIONS

We consider an adversary, $\mathcal{A}$dv, that controls entire software state of $\mathcal{P}$rv, including code and data. $\mathcal{A}$dv can thus modify any writable memory and read any memory (including secrets) that is not explicitly protected by *VRASED* access control rules. Also, $\mathcal{A}$dv has full access to all Direct Memory Access (DMA) controllers on $\mathcal{P}$rv.[2] Physical hardware attacks are out of scope of this paper: $\mathcal{A}$dv can not alter any hardware components to violate *VRASED* rules, e.g., by modifying code in ROM, inducing hardware faults, or retrieving $\mathcal{P}$rv secrets via physical side-channels. Protection against physical attacks is orthogonal and attainable via standard tamper-resistance techniques [31].

We assume that the MCU architecture strictly adheres to, and correctly implements, all of its specifications. In particular, we assume that program counter (*PC*) always contains the address of the instruction being executed in a given cycle. Whenever a DMA controller attempts to access main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (i.e., logical zero). The *reset* signal triggers an immediate (before executing the next instruction) system-wide reset. At the end of a successful *reset*, all registers (including *PC*) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, reset handling routine can not be modified.

5 PROOFS OF REMOTE SYSTEM-WIDE RESET (POR)

A "Proofs of (remote system-wide) Reset" (PoR) is needed in several situations. First, PoR can be used to restart software execution flow, if $\mathcal{P}$rv reaches an unexpected state. Second, if $\mathcal{P}$rv has a secure boot module, resetting is a way to trigger secure boot code (e.g., to wipe-out privacy sensitive data memory after completion of a task). Third, after a software update, PoR is needed to make sure that execution of newly installed software is initiated properly by restarting the MCU control flow execution. Without a reset, the MCU might start execution of the new code using state kept from software that was previously executing. Execution of new code might thus start at an inappropriate point, leading to an invalid control flow, unpredictable behavior, and even return-oriented programming (ROP) attacks [32]. Forcing a reset ensures that execution of the new code is properly initiated. PoR syntax and security goals are formalized in Definitions 2 and 3, respectively. PURE PoR scheme is presented in Construction 1. We prove that it is secure in Section 5.1 and formally verify its implementation in Section 5.2.

[2] DMA allows a hardware controller to directly access main memory (e.g., RAM, flash or ROM) without going through the CPU.

Definition 2 (syntax). *PoR is a tuple (**Request**, **Reset**, **Verify**) of algorithms:*

- **Request**$^{\mathcal{V}rf \rightarrow \mathcal{P}rv}()$ *: algorithm initiated by $\mathcal{V}$rf at time t to request a proof that $\mathcal{P}$rv has performed a reset at some point t' in time, where it must hold that $t' > t$. As a part of* **Request**, *$\mathcal{V}$rf sends a challenge to $\mathcal{P}$rv.*
- **Reset**$^{\mathcal{P}rv \rightarrow \mathcal{V}rf}(Chal)$ *: algorithm executed by $\mathcal{P}$rv to perform a reset and use Chal to provide an unforgeable proof H that a reset has happened.*
- **Verify**$^{\mathcal{V}rf}(H, Chal)$ *: algorithm executed by $\mathcal{V}$rf upon receiving H. It outputs 1 if H is a valid proof in response to* **Request**. *Otherwise, it outputs 0.*

Definition 3.
3.1 PoR Security Game (PoR-game): *Challenger plays the following game with $\mathcal{A}$dv:*

(1) $\mathcal{A}$dv is given full control over $\mathcal{P}$rv software state and oracle access to **Reset** *calls.*
(2) At time t, $\mathcal{A}$dv is presented with Chal.
(3) $\mathcal{A}$dv wins iff, after t, $\mathcal{A}$dv can produce $H_{\mathcal{A}dv}$, such that **Verify** *$(H_{\mathcal{A}dv}, Chal) = 1$, without causing $\mathcal{P}$rv to reset.*

3.2 PoR Security Definition:
A PoR scheme is considered secure if for all PPT adversaries $\mathcal{A}$dv, there exists a negligible function negl *such that:*

$$Pr[\mathcal{A}dv, PoR\text{-}game] \leq negl(l)$$

5.1 Security Proof for Construction 1

Theorem 1. *Construction 1 is a secure PoR implementation according to Definition 3 provided that HMAC is a secure MAC.*

Construction 1 (PURE PoR). *Suppose PoR.C is the program memory region inside the TCB storing PoR software binary: PoR.C ∈ TCB. Construction of PURE PoR is defined as follows:*

- **Request**$^{\mathcal{V}rf \rightarrow \mathcal{P}rv}()$*: $\mathcal{V}$rf generates a random challenge $Chal \leftarrow \$\{0,1\}^l$ and sends it to $\mathcal{P}$rv.*
- **Reset**$^{\mathcal{P}rv \rightarrow \mathcal{V}rf}(Chal)$*:*
  (1) Use VRASED's HMAC to compute $HMAC(\mathcal{K}, Chal)$, and write the result at RSP_RESULT_ADDR, where RSP_RESULT_ADDR is in a persistent storage (flash) memory region.
  (2) Enforce the following LTL invariant:

$$\mathbf{G}:\{PC = fst(PoR.C) \rightarrow [(PC \in TCB \wedge \neg DMA_{en}) \ \ \mathbf{W} \ \ reset]\} \quad (1)$$

- **Verify**$^{\mathcal{V}rf}(H, Chal)$*: $\mathcal{V}$rf returns 1 if and only if $H = HMAC(\mathcal{K}, Chal)$.*

Lemma 1.

$$
\begin{aligned}
&VRASED \ \wedge \\
&\mathbf{G}\text{:}\{ \\
&\quad [PC = fst(PoR.C) \wedge \mathbf{F}\text{:}(PC = lst(TCB))] \rightarrow [\neg(PC = lst(TCB)) \ \ \mathbf{U} \ \ reset] \\
&\quad \} \rightarrow \\
&\mathbf{G} : \{PC = fst(PoR.C) \rightarrow \ \ [(PC \in TCB \wedge \neg DMA_{en}) \ \ \mathbf{W} \ \ reset]\}
\end{aligned} \tag{2}
$$

where VRASED denotes the set of LTL properties already guaranteed by VRASED.

Proof. Recall that *VRASED* HMAC is already verified for functional correctness and termination. Thus, it adheres to cryptographic properties of HMAC's specification. In addition, *VRASED* architecture guarantees that $\mathcal{K}$ is not leaked to $\mathcal{A}$dv. It then follows, that a reduction can be constructed by using $\mathcal{A}$dv that wins the game in Definition 3, without calling **Reset**, to construct $\mathcal{A}$dv* which breaks the existential unforgeability of HMAC [33].

Since $\mathcal{A}$dv can not succeed without calling **Reset**, it remains to be shown that, when **Reset** is called to produce H, $\mathcal{P}$rv always resets before untrusted software (i.e., outside the TCB) resumes execution. However, this is exactly what is stated in LTL Specification 1 in Construction 1. LTL specification 1 states that, whenever **Reset** is called – i.e., when *PC* points to $fst(PoR.C)$, the first instruction in **Reset** code – *PC* remains inside *VRASED* TCB until the MCU resets[3]. Therefore, an unprivileged application can not execute until then and, a reset must occur before application execution can resume, in order to send H back to $\mathcal{V}$rf. Moreover, LTL specification 1 enforces that the DMA controller is disabled in the interim, also preventing DMA access to intermediate results that are written into *RSP_RESULT_ADDR* before *reset*. In other words, adding a formally verified implementation of LTL 1 to *VRASED* implies that Construction 1 is a formally verified implementation of a secure PoR. □

5.2 PoR Verified Implementation

In Section 5.1 we show that Construction 1 is a secure PoR scheme as long as we can formally verify that the underlying implementation adheres to LTL 1 defined in Construction 1. To guarantee LTL 1 with minimal modification to *VRASED*, we rely on Lemma 1.

Proof. The intuition behind Lemma 1 is that *VRASED* already guarantees that TCB code can not be interrupted before reaching its last instruction. This property is referred to as "atomicity" and specified in LTL as:

$$\mathbf{G} : \{\neg reset \wedge (PC \in TCB) \wedge \neg(\mathbf{X}(PC) \in TCB) \rightarrow PC = lst(TCB) \vee \mathbf{X}(reset)\} \tag{3}$$

$$\mathbf{G} : \{\neg reset \wedge \neg(PC \in TCB) \wedge (\mathbf{X}(PC) \in TCB) \rightarrow \mathbf{X}(PC) = fst(TCB) \vee \mathbf{X}(reset)\} \tag{4}$$

LTLs 3 and 4 express that the only valid entry point for *VRASED* software is the first instruction in the TCB, the only legal exit is via the last instruction in the TCB. Since $fst(PoR.C) \in TCB$ and LTL 4 is guaranteed, it suffices to enforce that:

$$\mathbf{G}\text{:}\{[PC = fst(PoR.C) \wedge \mathbf{F}\text{:}(PC = lst(TCB))] \rightarrow [\neg(PC = lst(TCB)) \ \ \mathbf{U} \ \ reset]\} \tag{5}$$

LTL 5 states that, after **Reset** functionality is invoked (when $PC = fst(PoR.C)$), *PC* does not reach the last instruction in the TCB before a system *reset* is triggered. On the other hand, *VRASED* triggers a *reset* if *PC* leaves the TCB from any instruction other than $lst(TCB)$. In addition to atomicity property, *VRASED* also enforces, through LTL Specification 6, that the DMA controller must be disabled during TCB code execution:

$$\mathbf{G} : \{(PC \in TCB) \wedge DMA_{en} \rightarrow reset\} \tag{6}$$

Therefore, after a call to **Reset** functionality, there is no way to exit the TCB or to enable unprivileged DMA access without triggering a reset.

Remark: *The above discussion provides some intuition for the proof of Lemma 1. The actual proof was automated and machine-checked using Spot [34], an LTL theorem prover. The formal computer proof is available at [9].* □

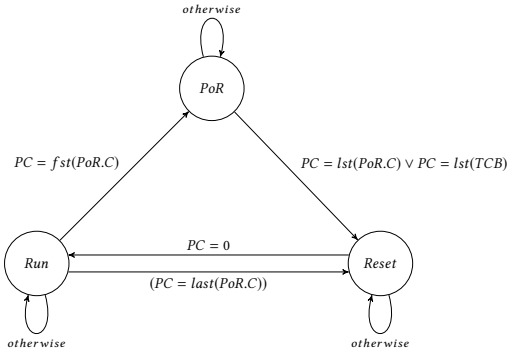


Figure 3: Verified hardware module for PoR

Figure 3 shows a formally verified FSM corresponding to the Hardware module enforcing LTL 5. This FSM works by monitoring the PC value. Normally $PC = lst(TCB)$ should not trigger a *reset*. In the PoR case, however, once $PC = fst(PoR.C)$, this state machine transitions into the *PoR* state. Once there, a future state with $PC = lst(TCB)$ always triggers a *reset*. Formal verification of such hardware module adopts the same methodology as used in *VRASED*. We design the FSM in Verilog HDL and automatically translate into SMV using Verilog2SMV [28]. Finally, we use the NuSMV model checker [30] to automatically prove that the state machine complies with LTL 5.

5.3 PoR Correctness

We design the PoR scheme to co-exist with *VRASED* remote attestation functionality. In Figure 3, a *reset* is only triggered at $PC = lst(TCB)$ after the **Reset** functionality is invoked. This way, the original *VRASED* remote attestation is unaffected by the added rules.

PoR security implies that it is impossible for untrusted code to produce H without resetting, and any attempt to do so triggers a reset. We now consider PoR correctness. Figure 4 shows the C implementation of the software TCB, including both *VRASED* $\mathcal{R}$A and PoR. Line 7 corresponds to $lst(PoR)$. If nothing goes wrong,

[3]Note that **Reset**'s implementation –PoR.C– resides inside the TCB (i.e., inside SW-Att using *VRASED* nomenclature). In other words, $PoR.C \in TCB$.

```
1  void Hacl_HMAC_SHA2_256_hmac_entry(uint8_t operation) {
2      uint8_t key[64] = {0};
3      memcpy(key, (uint8_t*) KEY_ADDR, 64);
4      if (operation == RESET){
5          // fst(PoR.C) instruction:
6          hacl_hmac((uint8_t*) RST_RESULT_ADDR, (uint8_t*) key, (uint32_t) 64,
                   (uint8_t*) CHALL_ADDR, (uint32_t) 32);
7          return;
8          // lst(PoR.C) instruction.
9      }else{
10         //Unmodified VRASED code for regular attestation
11         hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
                   CHALL_ADDR, (uint32_t) 32);
12         hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32, (
                   uint8_t*) ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
13     }
14     return();
15 }
```

Figure 4: *VRASED* **software TCB extended to incorporate PoR functionality**

HMAC executes and terminates at $lst(PoR)$. In PoR's FSM of Figure 3, $PC = lst(PoR)$ always triggers a *reset*, which is the expected behavior after completion of HMAC. After a reset, H is accessible at *RST_RESULT_ADDR* persistent memory address (Line 6 in Figure 4), and $\mathcal{P}$rv can forward it to $\mathcal{V}$rf.

5.4 Practical Considerations

After a reset, the (unprivileged) MCU application always begins execution anew. It must start by checking whether something (H) is stored at *RST_RESULT_ADDR*. If so, the first action is to send H to $\mathcal{V}$rf and zero-out *RST_RESULT_ADDR*, thus completing the PoR protocol. We note that this code is malleable and is not part of the TCB. In particular, it can always decide not to reply to $\mathcal{V}$rf. However, lack of response would cause $\mathcal{V}$rf to realize that something is wrong with $\mathcal{P}$rv.

6 PROOFS OF (REMOTE SOFTWARE) UPDATE (PoU)

We consider a Proof of (Remote Software) Update (PoU) scheme involving two parties: $\mathcal{P}$rv and $\mathcal{V}$rf. The goal is to allow $\mathcal{V}$rf to request a remote software update for $\mathcal{P}$rv, and to obtain an unforgeable proof upon a successful update. The generic description of PoU scheme is presented in Definition 4.

Definition 4 (syntax). *A PoU scheme consists of a tuple* (**Request**, **Install**, **Verify**) *fulfilling the following:*

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S})$*: algorithm initiated by* $\mathcal{V}$rf *to request software* $\mathcal{S}$ *to be installed on* $\mathcal{P}$rv *at a memory range* UR*, where* $|UR| = |\mathcal{S}|$*.* $\mathcal{P}$rv *also receives a challenge* Chal *as a part of the request.*
- **Install**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$(Chal, $\mathcal{S}$)*: algorithm executed by* $\mathcal{P}$rv *to update software such that* $UR = \mathcal{S}$*. Upon successful update, it outputs a proof* H *that the update has completed.*
- **Verify**$^{\mathcal{V}\text{rf}}$(H, Chal, $\mathcal{S}$)*: algorithm executed by* $\mathcal{V}$rf *upon receiving* H*. It outputs* 1 *if* H *is a valid proof in response to* **Request**. *Otherwise, it outputs* 0.

Based on this syntax, we define a security game in Definition 5 that captures our $\mathcal{A}$dv model (discussed in Section 4). Intuitively, security of PoU means that $\mathcal{A}$dv cannot compute a valid proof of successful update for software that is not installed on $\mathcal{P}$rv. In other words, $\mathcal{A}$dv wins the game if it can generate a valid proof of update without actually updating UR memory on $\mathcal{P}$rv with $\mathcal{S}$. PURE secure PoU scheme is presented in Construction 2.

Definition 5.
5.1 PoU Security Game (PoU-game):
Notation:
- UR is the memory region on $\mathcal{P}$rv that should be updated, and $UR(t)$ denotes the content of UR at time t.

Challenger plays the following game with $\mathcal{A}$dv:

1. *$\mathcal{A}$dv is given full control over $\mathcal{P}$rv software state and oracle access to* **Install**.
2. *$\mathcal{A}$dv is presented with* Chal *and* $\mathcal{S}$*, and continues to have oracle access to* **Install**.
3. *Eventually at time t, $\mathcal{A}$dv queries* **Install** *and outputs* H.
4. *$\mathcal{A}$dv wins if and only if* **Verify**(H, Chal, $\mathcal{S}$) = 1 *and* $UR(t) \neq \mathcal{S}$.

5.2 PoU Security Definition:
A PoU scheme is secure if, for all Probabilistic Polynomial-Time (PPT) adversaries $\mathcal{A}$dv, there exists a negligible function negl *such that:*

$$Pr[\mathcal{A}\text{dv}, \textit{PoU-game}] \leq \text{negl}(\lambda)$$

6.1 Security Proof for Construction 2

Theorem 2. *Construction 2 is secure according to Definition 5.2 as long as VRASED is secure according to Definition 1.*

Proof. The proof is by reduction of *VRASED* security in Definition 1 to Construction 2's security. Suppose there exists adversary $\mathcal{A}$dv such that $Pr[\mathcal{A}\text{dv}, \text{PoU-game}] > \text{negl}(l)$. We then show that such $\mathcal{A}$dv can be used to construct $\mathcal{A}\text{dv}^*$ that breaks *VRASED* security game in Definition 1. $\mathcal{A}\text{dv}^*$ acts by selecting the same $\mathcal{S}$ and output σ used by $\mathcal{A}$dv to win PoU-game. Recall from Definition 5 and Construction 2 that, since $\mathcal{S}$ and σ allow $\mathcal{A}$dv to win PoU-game, it must hold that:

$$\sigma = HMAC(KDF(\mathcal{K}, \text{Chal}), \mathcal{S}) \text{ and } UR(t) \neq \mathcal{S}$$

According to Definition 1 and the fact that $UR = AR$, $(\mathcal{S}, \sigma)$ is a valid response in $\mathcal{R}$A-game. Therefore, $\mathcal{A}\text{dv}^*$ is able to win $\mathcal{R}$A-game

Construction 2 (PURE PoU). *Since $\mathcal{P}$rv has a VRASED-compliant architecture, where memory range $AR = UR$, the construction is defined as follows:*

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S})$*: $\mathcal{V}$rf outputs $\mathcal{S}$ and a random challenge* Chal $\leftarrow \$\{0,1\}^l$ *to $\mathcal{P}$rv.*
- **Install**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$(Chal, $\mathcal{S}$)*: $\mathcal{P}$rv performs three steps:*
  1. *Unprivileged software receives $\mathcal{S}$ and* Chal *and writes them into UR and MR, respectively.*
  2. *Call VRASED* `SW-Att`*'s remote attestation function to compute* H = $HMAC(KDF(\mathcal{K}, \text{Chal}), AR) = HMAC(KDF(\mathcal{K}, \text{Chal}), UR)$ *and write* H *into MR.*
  3. *Control returns to unprivileged software which is responsible for sending the content of MR to $\mathcal{V}$rf.*
- **Verify**$^{\mathcal{V}\text{rf}}$(H, Chal, $\mathcal{S}$)*: $\mathcal{V}$rf returns 1 if and only if* H = $HMAC(KDF(\mathcal{K}, \text{Chal}), \mathcal{S})$.

Construction 3 (PURE PoE). *Given that $\mathcal{P}$rv has an unmodified VRASED-compliant architecture, where memory range $AR = UR$, our construction is defined as follows:*

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}()$*: $\mathcal{V}$rf sends* 0 *and a random challenge* $C\text{hal} \leftarrow \$\{0,1\}^l$ *to $\mathcal{P}$rv.*
- **Install**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}(0, C\text{hal})$*: $\mathcal{P}$rv performs three steps:*
  (1) Unprivileged software receives $C\text{hal}$*, writes it into MR, sets all memory cells in UR to* 0*.*
  (2) Call VRASED SW-Att *– remote attestation function – to compute* $\mathsf{H} = HMAC(KDF(\mathcal{K}, C\text{hal}), AR) = HMAC(KDF(\mathcal{K}, C\text{hal}), \{0\}^{|UR|})$ *and write* H *into MR.*
  (3) Control returns to unprivileged Software, which is responsible for returning the content of MR to $\mathcal{V}$rf.
- **Verify**$^{\mathcal{V}\text{rf}}(\mathsf{H}, C\text{hal})$*: $\mathcal{V}$rf returns 1 if and only if* $\mathsf{H} = HMAC(KDF(\mathcal{K}, C\text{hal}), \{0\}^{|UR|})$.

with the same non-negligible probability that $\mathcal{A}$dv has of winning PoU-game. □

Note that Construction 2 uses *VRASED as-is and involves no additional trusted software or hardware.* Therefore, there is no additional effort for verifying its implementation. Also, Construction 2 requires unprivileged helper code to receive $C\text{hal}$ and $\mathcal{S}$, write them to memory, and invoke *VRASED* remote attestation functionality. This helper code is outside the TCB and has no impact on PoU security. If $\mathcal{P}$rv is infected and malware refuses to perform the update, such anomalous behavior is always detected by $\mathcal{V}$rf due to either lack of any response, or an invalid response ($UR \neq \mathcal{S}$).

6.2 Practical Considerations

PoU security relies on $\mathcal{A}$dv's inability to produce a valid H without installing correct new software on $\mathcal{P}$rv. However, it does not prohibit a network $\mathcal{A}$dv from spoofing PoU's request. Such an attack could lead $\mathcal{P}$rv to inadvertently install malicious software chosen by $\mathcal{A}$dv, while believing that the request came from $\mathcal{V}$rf. This can be easily prevented by authenticating $\mathcal{V}$rf's request before performing PoU. Fortunately, *VRASED* already supports $\mathcal{V}$rf authentication functionality as an optional part of its design; see Appendix B in [9]. This functionality can be seamlessly incorporated into PURE PoU construction to prevent installation of unauthorized software.

Another practical consideration is the need to protect the helper code. Recall that our construction requires it to receive the challenge, install the update and send H back to $\mathcal{V}$rf. As we argue in Section 6.1, this helper code does not have to be verified or trusted, since it does not impact PURE PoU security. Nonetheless, if helper code is not write-protected, malware can infect $\mathcal{P}$rv, modify or erase it, and simply leave $\mathcal{P}$rv. This simple attack would prevent $\mathcal{P}$rv from receiving and performing any subsequent update, and consequently put $\mathcal{P}$rv into a *non-updatable* state, which would require physical access to remedy. This might be problematic in settings where physical access to $\mathcal{P}$rv is hard. This attack can be easily circumvented by making helper code immutable, by either placing it in ROM, or adding MCU access control rules to enforce its immutability.

7 PROOFS OF MEMORY ERASURE (POE)

Several scenarios require a proof of memory erasure (PoE). For example, it can be used to ensure deletion of sensitive information when $\mathcal{P}$rv is no longer of use, or whenever there is a change of its ownership. Alternatively, PoE can be used in conjunction with a PoU to ensure that $\mathcal{P}$rv is in a pristine (malware-free) state before installing a software update.

PoE is also a two-party protocol between $\mathcal{V}$rf and $\mathcal{P}$rv, where the former requests erasure of a specific memory region on $\mathcal{P}$rv to be erased, i.e., zeroed-out. Based on this informal definition, PoE can be viewed as a special case of PoU, where newly installed software $\mathcal{S}$ is a priori known default value, e.g., a string of zeros $\mathcal{S} = \{0\}^{|UR|}$. $\mathcal{S}$ can be compressed and represented using a single bit, i.e., 0, during transmission in **Request**.

Since PoE is a subset of PoU, the PoE security definition is derived from Definition 5 by substituting arbitrary $\mathcal{S}$ with $\{0\}^{|UR|}$. We present PURE PoE scheme in Construction 3. Because its security proof follows the same arguments as that of Theorem 2, we omit it.

7.1 Practical Considerations

A typical use case of PoE is secure erasure of $\mathcal{P}$rv *data memory*, ensuring that no sensitive data remains. Nonetheless, it can also be used to provide a proof of absence of malware on $\mathcal{P}$rv by setting UR to $\mathcal{P}$rv's entire writable memory, both program and data segments. Since, akin to PoU, PoE needs helper code to receive the challenge and return the proof to $\mathcal{V}$rf, wiping out all of $\mathcal{P}$rv's writable memory would imply erasing this helper code as well, which is obviously problematic. Naïvely, $\mathcal{P}$rv can simply avoid erasing helper code. However, this does not guarantee malware-free state, since malware might manage to hide itself in the helper code region. The most intuitive remedy is to make helper code immutable, as with PoU, e.g., by housing it in ROM. Alternatively, we can leave helper code be, and attest its presence (along with zeros everywhere else). However, this alternative approach would require incorporating this helper code into PoE's definitions as well as modifying PURE PoE construction and its proof accordingly.

8 EVALUATION

We implemented PURE by extending *VRASED* with the ability to perform PoU, PoE, and PoR schemes. Our implementation is synthesized and executed using the commodity FPGA prototyping board Basys3. We report PURE additional overhead compared to an unmodified *VRASED* architecture.

Hardware. Table 2 shows PURE hardware overhead. Compared to *VRASED*, the number of Look-Up Tables (LUT) grows by 4 and the number of registers – by 3. With respect to unmodified OpenMSP430 architecture, this represents an overhead of 0.2% for LUT and 0.4% for registers. This is in addition to *VRASED* overhead of $\approx$ 6.6% and $\approx$ 5.8% for LUT and registers, respectively.

Memory. *VRASED* requires $\approx$ $4.5KB$ of ROM, most of which is for storing HACL* HMAC-SHA256 code. Since PURE re-uses the same HMAC implementation, additional memory is minimal: 50 bytes of ROM. Unprivileged helper code, for installing update and erasing memory requires additional 26 bytes of program memory. Execution of PURE services requires only slightly more RAM: 4

Architecture	Software (in bytes)			Hardware	
	ROM Size	Helper Code Size	Max Runtime Mem. Usage	LUT	Reg
OpenMSP430 [35]	-	-	-	1842	684
VRASED [9]	4500	112	2332	1964	721
PURE	4550	138	2336	1968	724

Table 2: PURE (additional) hardware and software cost

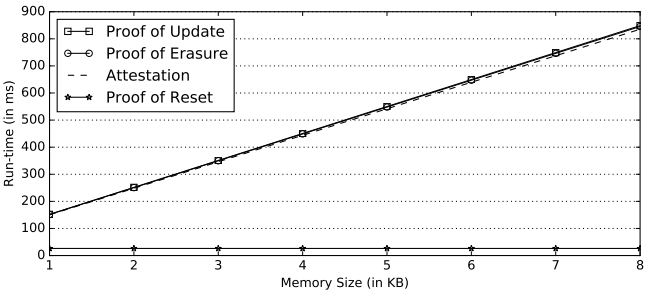


Figure 5: Total runtime of PURE services running at 8MHz

bytes for storing the operation code that defines which operation is executed for a given TCB call.

Runtime. Figure 5 shows runtime for each service. PoR requires 210, 385 cycles to compute an HMAC, resulting in total time of 26*ms* when running at 8MHz. Note that PoR does not require attesting memory and therefore this time is constant. PoU and PoE runtimes are similar, linear in terms of updated/erased memory size. Due to the execution of their helper codes, their runtimes are slightly higher (at most 1.6% higher) than the attestation runtime.

9 CONCLUSIONS

This paper described PURE: a set of three provably secure services targeting low-end embedded devices: system-wide reset, erasure, and software (more generally, memory) update. We argue that serial composition of these services allows $\mathcal{V}$rf to ensure that a remote $\mathcal{P}$rv has been re-configured to a functional and malware-free state, and properly re-initialized. PURE was implemented on a popular commercially available low-end embedded system, MSP430, and synthesized using a commodity Basys3 FPGA. Its additional hardware footprint overhead is $\approx 0.4\%$ and it takes on the order of 100-s milliseconds to generate proofs of update, erasure, and reset, altogether.

This paper motivates several directions for future work, including: (1) extending proposed services to swarms/groups of devices; (2) implementing and formally verifying these services for medium- and higher-end embedded systems; and (3) introducing new trustworthy applications, protocols, and systems that use these services as building blocks.

Acknowledgments: UC Irvine authors' work was supported in part by ARO under contract: W911NF-16-1-0536, and NSF WiFiUS Program Award #: 1702911. Authors thank ICCAD anonymous reviewers for their valuable comments.

REFERENCES

[1] Trusted Computing Group., "Trusted platform module (tpm)," 2017.
[2] Intel, "Intel Software Guard Extensions (Intel SGX)." https://software.intel.com/en-us/sgx.
[3] Arm Ltd., "Arm TrustZone." https://www.arm.com/products/security-on-arm/trustzone, 2018.
[4] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.
[5] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.
[6] F. Brasser et al., "Tytan: Tiny trust anchor for tiny devices," in *DAC*, 2015.
[7] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *Wisec*, 2017.
[8] J. Noorman, J. V. Bulck, J. T. Mühlberg, *et al.*, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, 2017.
[9] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Formally verified hardware/software co-design for remote attestation," in *USENIX Security Symposium*, 2019.
[10] 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.
[11] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*, 2003.
[12] R. W. Gardner, S. Garera, and A. D. Rubin, "Detecting code alteration by creating a temporary memory bottleneck," *IEEE TIFS*, 2009.
[13] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *ACM CCS*, 2011.
[14] 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.
[15] Trusted Computing Group (TCG), "Website." http://www.trustedcomputinggroup.org, 2015.
[16] 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.
[17] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification.," in *OSDI*, 2014.
[18] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *USENIX*, 2015.
[19] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *USENIX*, 2017.
[20] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS*.
[21] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*, 2012.
[22] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *IEEE S&P*, 2013.
[23] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, 2009.
[24] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "seL4: Formal verification of an OS kernel," in *ACM SIGOPS*, 2009.
[25] G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese, and D. Vendraminetto, "Formal verification of embedded systems for remote attestation," *WSEAS Transactions on Computers*, vol. 14, 2015.
[26] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto, "Secure embedded architectures: Taint properties verification," in *DAS*, 2016.
[27] 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 *International Conference on Computer Aided Verification*, pp. 359–364, Springer, 2002.
[28] 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.
[29] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Temporal consistency of integrity-ensuring computations and applications to embedded systems security," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018.
[30] 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.
[31] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSI Design*, 2004.
[32] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07*, 2007.
[33] Y. Lindell and J. Katz, *Introduction to modern cryptography*, ch. 4.3, pp. 109–113. Chapman and Hall/CRC, 2014.
[34] 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.
[35] O. Girard, "openMSP430," 2009.