SμV—The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things

Mahmoud Ammar, Bruno Crispo, *Senior Member, IEEE*, Bart Jacobs, Danny Hughes, and Wilfried Daniels

Abstract—The Internet of Things (IoT) is shaped by the increasing number of low-cost Internet-connected embedded devices that are becoming ubiquitous in every aspect of modern life. With their cost-sensitive design, integrating hardware-based security mechanisms into such devices is undesirable. Therefore, securing these devices is a particularly difficult challenge, especially, due to their growing popularity as attack targets, via remote malware infestations. The vast majority of such devices are bare-metal, where they execute programs in fully-accessible and unprotected memories without any operating system and even without including any form of security. This is beside the fact that IoT operating systems offer little or no protection. This paper addresses this problem through the concept of a *Security MicroVisor* (SμV), which provides embedded devices that lack hardware-based memory protection units with memory isolation using software virtualisation and assembly-level code verification. More specifically, our contribution is two-fold. First, we propose SμV as a software-based memory isolation technique. We then formally verify the software architecture, written in C, to prove that it is memory-safe and crash-free. Second, we propose a software-based remote attestation, as an example of a fundamental security service that can be implemented on top of SμV, to detect malware-infected devices. We first describe the design and implementation of SμV. Then, we highlight the formal verification of software architecture, and characterize the remote attestation protocol. We evaluate the SμV implementation using an 8-bit AVR microcontroller that is widely used in IoT devices. Evaluation results show that SμV provides strong security guarantees while maintaining extremely low overhead in terms of memory footprint, performance, and power consumption. Furthermore, we extend the performance evaluation also to the remote attestation scheme, illustrating its limited overhead.

Index Terms—Memory isolation, IoT security, remote attestation, formal verification

1 Introduction

The Internet of Things (IoT) extends the boundary of the traditional Internet by including a wide variety of embedded devices that create values by connecting digital processes to the physical world. These devices are used at a large scale in diverse application scenarios such as automobiles, smart homes, factory automation, and smart cities. With the pervasiveness of IoT, two important trends can be observed. First, the network connectivity of devices keeps increasing as more tiny devices are connected to the Internet or local networks. Second, software extensibility of these devices is supported by incorporating third-party developed software modules. The combination of these two streams is advantageous as they enable a vast array of interesting services, ranging from over-the-air updates on smart devices to programmable sensor networks. However, they also have a negative impact on security, leading to various malware threats. Despite the specialized nature of embedded devices in terms of limited resources and computing power, they become attractive targets to a wide variety of remote attacks since they often process privacy-sensitive information and perform safety-critical tasks. Researchers have already shown how to remotely perform code injection attacks against Internet-connected embedded devices [1], [2], exploiting their support for software extensibility. High-profile examples of real cyber attacks include a smart hotel targeted ransomware attack [3], HVAC systems attack [4], and a casino-targeted database stealing attack through remotely controlling a vulnerable thermometer [5].

With the increase of the network connectivity, ensuring the security of embedded devices is critical since many of them are low cost with software running directly on the hardware, known as *bare-metal systems*. In such systems, the user application executes as privileged low-level software with direct access to the processor and peripherals, without going through operating system software layers. Unfortunately, due to the intrinsic limits of low-end IoT devices, which account for the majority of devices in the field, achieving equivalent security properties to mainstream computer systems [6] on embedded devices poses fundamental design

- *M. Ammar and B. Jacobs are with KU Leuven, Leuven 3000, Belgium. E-mail: {Mahmoud.Ammar, Bart.Jacobs}@cs.kuleuven.be.*
- *B. Crispo is with KU Leuven, Leuven 3000, Belgium, and also with the University of Trento, Trento 38122, Italy. E-mail: bruno.crispo@unitn.it.*
- *D. Hughes is with KU Leuven, Leuven 3000, Belgium, and also with VersaSense, Leuven 3001, Belgium. E-mail: Danny.Hughes@cs.kuleuven.be.*
- *W. Daniels is with VersaSense, Leuven 3001, Belgium. E-mail: Wilfried.Daniels@versasense.com.*

Manuscript received 8 Feb. 2019; revised 21 June 2019; accepted 30 June 2019; Date of current version 30 Aug. 2019.
(Corresponding author: Mahmoud Ammar.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2019.2928541

challenges. First, bare-metal devices satisfy strict runtime guarantees on extremely constrained hardware platforms with few KBs of memory and low CPU speed. This means that any proposed protection mechanism has to be light-weight and incur an acceptable overhead while keeping the price of the corresponding devices reasonable. Second, IoT devices are application-specific. As such, in contrast to general computer systems, they have different security needs depending on the application domain. Third, in many embedded systems, a single program has a full access to all hardware resources, where this violates with best security practices that require restricting access only to necessary resources. Accordingly, many traditional security methods cannot be applied to secure IoT systems, leaving the door open for the various remote cyber attacks.

Despite the clear security risks, a significant number of deployed IoT devices provide little or no protection against remote malware infestation. A number of research proposals have addressed this problem using security techniques such as remote attestation (RA) [7], [8], [9], [10], [11], [12], [13]. However, all of the aforementioned proposals and others (we provide a brief survey in the related work section) depend either on the existence of a hardware-based memory protection unit (MPU) or hardware support. While the latter solution is expensive, the former is not always valid as even basic features such as MPUs are typically not available on some IoT devices such as IETF Class-1 [14].

To address this challenge, we present a reliable software-based security architecture, named the *Security MicroVisor* (SμV)[1] [15]. SμV is a security middleware that uses selective software virtualisation and machine-level code verification to isolate a Trusted Software Module (TSM) from untrusted application software. The core contribution of SμV is that it is capable of providing strong isolation guarantees that were previously only possible via dedicated hardware. SμV works with all standard Micro Controller Units (MCU) that support global interrupt disabling, are single threaded and have sufficient memory to support the pre-installed SμV module. To the best of our knowledge, these features are offered by all MCUs used in contemporary IoT products [14], including all major families of MCUs from top vendors, such as AVR MCUs [16], NXP MCUs [17], and ARM Cortex-M MCUs [18]. Inkwood research, and other research institutes and industry parties estimate that lightweight MCUs that are currently employed in IoT products cover more than 25 percent of the IoT market. They also expect that by 2026 such MCUs will keep sharing at least 10 percent of the IoT market, occupying a market size of no less than $150 million, with a number of deployed devices exceeding 5 billions [19], [20], [21], [22].

To ensure reliability, in addition to verifying the correctness of the design, the SμV software architecture ($\sim$500 lines of code written in C) has to be bug-free, memory-safe, and behave as expected. Many errors in C programs such as illegal memory accesses are generally difficult to detect by traditional testing methods, since they do not cause a clean crash but have unpredictable and undesired effects that are not easy to diagnose. Since the goal of the SμV is to provide legacy IoT devices with strong isolation guarantees without any hardware modification, it should not have bugs or cause failures. Therefore, leveraging formal verification approaches, we formally verify some properties of the SμV implementation for ATmega devices to achieve a base level of reliability, as a first step towards a fully verified system. In particular, we use VeriFast [23], a modular software verifier, to formally verify the code of SμV and provide a proof that it is memory-safe and crash-free. As a consequence, it is secure against the well-known memory-related attacks.

Leveraging the SμV, we implement remote attestation, a well-known malware mitigation technique. To the best of our knowledge, the design of the remote attestation protocol is the first software-based approach that does not depend on any hard assumption (e.g., strict execution time), comparing with all previous software-based approaches [24].

Another major contribution of this paper is that we make the source code of SμV along with RA and verification results publicly available at [25] in order to ensure the reproducibility and verifiability of our findings.

We perform a benchmark evaluation of SμV on MicroPnP, a commercial representative IoT platform [26] that uses an ATmega 1284p microcontroller (10 MHz, 16 KB RAM, 128 KB Flash Memory) and time-synchronized, channel hopping IEEE 802.15.4e network chip [27]. We show that the runtime overhead of SμV is very reasonable, incurring a decrease in battery life of under 1 percent in realistic application scenarios and consuming less than 3 KB of flash. Furthermore, the overhead of our exemplar security scheme (RA) is promising, as hourly interactive remote attestation reduces battery life by a maximum of 6.2 percent.

Paper Outline. The remainder of this paper is organized as follows. Section 2 describes the design rationale and key mechanisms of SμV. Implementation details are discussed in Section 3. The formal verification of SμV is presented in Section 4. Security Analysis is provided in Section 5. Section 6 describes the SμV-dependent remote attestation protocol proposed. Section 7 reports on the evaluation of SμV and its accompanying RA technique. Section 8 reviews related work and Section 9 concludes the study and discusses some future directions.

2 Design of SμV

Memory isolation is the primitive used in almost all virtualization-based security systems [28], [29], [30], [31], [32], [33]. It denies any unauthorized access to sensitive physical memory locations. In this section, we provide an overview of the software mechanisms that SμV uses to provide memory isolation and support for various security features such as remote attestation. In particular, the SμV isolates itself and all accompanying secrets, crypto, and security-related functions from being tampered with by any untrusted user application that is loaded in the same physical memory.

The design of SμV shares similarities with the Software-based Fault Isolation (SFI) approach proposed by Wahbe et al. [34]. SFI prevents faults in untrusted software modules from corrupting other software on processors with a single shared address space and no memory protection. For each

1. This work is a significant extension of a paper published in 2017 at the industrial track of the ACM/IFIP/USENIX international middleware conference [15]. We proceed further in the description without relying on any prior knowledge of the published paper.

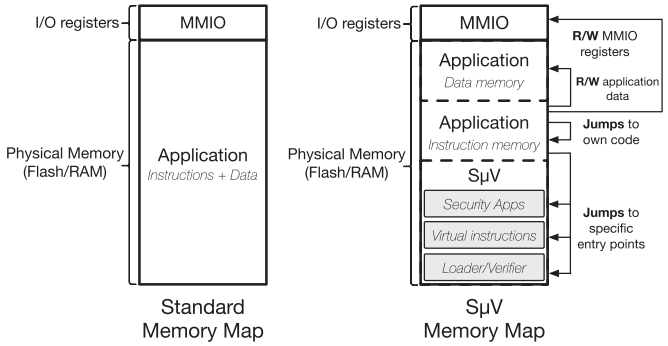


Fig. 1. Standard (left) and SμV (right) memory map. In the standard case, memory is monolithic and operations are unrestricted. SμV splits application memory into instruction and data memory and restricts possible sensitive operations.

software module, SFI reserves a logically separate portion of the application's address space. The isolation of module address spaces is maintained at runtime by rewriting unsafe instructions to verify target addresses. SμV also uses selective software virtualization and assembly-level code verification to ensure full isolation of the trusted software module (SμV) from untrusted application software.

2.1 Attacker Model

We assume a remote adversary who has a full access to the network with knowledge of a generic memory corruption vulnerability (e.g., the user software running on the SμV-enabled IoT device is buggy but not malicious). The attacker aims to inject her own malicious code, corrupt specific data, edit the content of any memory location, or directly manipulate security-critical outputs in the device itself by sending data to specific IO pins or over the air. We assume also that the adversary exploits the Arbitrary Memory Overwrite vulnerability, known as *Write-What-Where* vulnerability. It allows the adversary to write any data to any memory location that she targets. The adversary may have obtained any of the vulnerabilities through a variety of means (e.g., source code analysis of the user software, or identifying security flaws by reverse engineering a similar binary image that runs on a different unprotected device). Furthermore, we assume that IoT devices considered lack the Direct Memory Access Controller (DMA). In other words, the SμV is aimed at devices that either physically disable or lack DMA (which accounts for the majority of Class-1 IoT devices). Furthermore, we assume that the SμV is installed correctly on the IoT device by a trusted party. Finally, we assume that the attacker cannot physically tamper with the IoT device. However, physical attacks are not scalable, as physically tampering with a SμV-enabled IoT device affects only the tampered device. We consider that protection against physical attacks is orthogonal, and can be achieved by deploying the IoT device inside a tamper-proof protection shield [35].

2.2 The Platform Requirements of SμV

The vast majority of IoT devices are based on off-the-shelf Micro Controller Units, that are optimized for low cost and low power consumption. Hardware security features and Memory Protection Units are uncommon on these devices and millions of IoT devices are already deployed without these features. This essentially means that a piece of software running on such a device can execute arbitrary instructions and read or modify both data and instruction memory.

SμV is a pure software solution which allows additional security features to be added to conventional microcontrollers by ensuring first the isolation between trusted and untrusted modules. In this paper, we will use SμV to provide MPU-like memory protection and support for remote attestation. These basic MCUs are assumed to have the following characteristics:

1) *No memory protection*: The MCU is not required to provide any form of memory protection. Nor is it required that the MCU provides ROM memory. The MCU must, at a minimum provide sufficient flash memory to store the SμV MicroVisor (∼1 KB for typical MCU architectures).
2) *Single thread of execution*: We assume a single thread of execution. This is to guarantee atomic execution of critical code without preemption by other threads. This is typical for conventional MCUs.
3) *Interrupts*: The MCU must support the disabling of global interrupts to ensure the atomic execution of code without preemption by interrupt handlers. This feature is offered by all major families of MCUs.

2.3 Architecture of SμV

SμV reserves part of the memory for trusted software which we call the MicroVisor. This software is installed prior to the deployment of the IoT device using a physical programming device (e.g., SPI or JTAG). The MicroVisor code is considered immutable and resides in *virtual* ROM memory, which is enforced by the MicroVisor itself. The remainder of the device memory, from now on referred to as *Application Memory*, is available to untrusted applications. Application memory is further subdivided into *Instruction Memory*, which applications may execute but not read or write to and *Data Memory* for holding data. We visualize this in Fig. 1.

The trusted MicroVisor code is subject to no restrictions. Untrusted applications on the other hand are strongly restricted to the following rules:

1) *Control transfer*: branch and jump operations can only address the application instruction memory or one of the entry points of the MicroVisor instruction memory. This allows for controlled interaction with the MicroVisor.
2) *Data memory modification*: read and write operations can only address application data memory or Memory Mapped IO (MMIO) locations.
3) *Instruction memory modification*: read and write operations can not address the application instruction memory or the MicroVisor memory.
4) *Deployment* of new applications can only occur through the MicroVisor. This property is enforced by preventing applications, subject to the previous restrictions, from writing their own instruction memory; only the MicroVisor is allowed to do so. As a result, all new applications pass through the MicroVisor during loading.

Restrictions on application code are enforced at the instruction level through two basic mechanisms:

```
app_function:
...
// Load pointer register r30 with addr
load r30, 0xF512
// Calls the function r30 points to
icall
...
ret
```

Listing 1. Unsafe indirect call through pointer register. The icall instruction is inherently unsafe and will be replaced by SμV during verification

(i) incoming applications are *verified* by the MicroVisor at load-time to ensure that they adhere to the rules listed above, and (ii) certain inherently unsafe instructions which are nonetheless essential for normal operation are replaced by their safe *virtualized instruction* counterparts.

2.3.1 Verifier (Loader)

As described above, application deployment can only occur through the MicroVisor, which subjects the application code to assembly-level verification at load time on the IoT device itself. Only *safe* instructions are allowed, i.e., instructions that do not violate the above memory restrictions. Two types of illegal instructions can be distinguished: (i) instructions that statically jump to or access restricted memory, and (ii) instructions that jump to or access *any* memory dynamically and cannot be checked statically.

Most control transfer instructions, such as program-counter relative branches and calls, have their target address encoded in the instruction and can be checked statically by the verifier at load time. Store operations to static variables also use an immediate addressing mode and can be checked by the verifier. Any instruction of this type that has an illegal memory address as static argument is detected, resulting in rejecting the application by the verifier and canceling its deployment. Applications that contain instructions which cannot be statically checked are rejected outright. Instructions using indirect addressing belong to this category, such as jumps and stores that use a pointer register to hold their target address. These are common when using pointer logic or arrays in C. An example of such an illegal instruction is shown in Listing 1, where a function is called through a pointer. As supporting these operations is essential to normal operation of any processor, they must be replaced prior to deployment with calls to secure *virtualized* instructions provided in the MicroVisor that perform runtime checking of their arguments. SμV offers toolchain support that transparently replaces these operations for the developer.

```
app_function:
...
// Load pointer register r30 with addr
load r30, 0xF512
// Call suv_safe_icall residing in SuV
call suv_safe_icall
...
ret
```

Listing 2. Safe indirect call with at runtime checking of the dynamic argument by a subroutine in SμV

```
suv_safe_icall:
// Clear global interrupt flag
cli
//Check validity of target addr in r30
comp r30, 0xF000
branch_gt icall_failure
icall_success:
// Success,set interrupt flag and jump
sei
ijmp
icall_failure:
// In case of failure, soft reset
jmp 0x0000
```

Listing 3. Example virtualized indirect call subroutine in SμV. In this case the check is very simple: fail if target above 0xF000

2.3.2 Secure Virtual Instructions

The MicroVisor offers replacements for all unsafe dynamic instructions, which can be accessed via a call to a subroutine in the controlled MicroVisor memory. These virtual instructions check their arguments and perform the matching operation only when it does not break memory access or control transfer rules. Following the execution of the virtual instruction, the MicroVisor returns control to the application. Any operation which attempts to access an illegal memory addresses is trapped and causes the microcontroller to reset. Using this approach, the security features of the MCU are enhanced, without sacrificing functionality.

Listing 2 shows how the unsafe indirect call instruction from the running example in Listing 1 is replaced by a safe virtualized alternative in the form of a subroutine call to the SμV which is known to be safe and is therefore not rejected by the verifier. Listing 3 shows an example implementation for the secure virtualized indirect call operation residing in the SμV. The virtualized alternative will first disable global interrupts to ensure atomic verification of the target address. This is essential, as the adversary could schedule an interrupt in order to interfere with the outcome of the check. The check in this example is deliberately trivial: any address under `0xF000` is considered a valid target, where `0xF000` represents the first address of SμV memory. Next, if the target address is valid, global interrupts are re-enabled and a jump to the target is carried out. Please note that the implementation in Listing 3 is instantiated for AVR architecture, in which the attacker is not able to interfere between the instructions of enabling interrupts and performing the indirect jump, as it is monotonic. In other words, when using the *SEI* instruction to enable interrupts, the instruction following *SEI* will be executed before any pending interrupt; this is the case on Atmel ATmega's AVR architecture[2] [37]. As we discuss in Section 3, this paper proposes a general design, whereas the implementation addresses only the AVR-based Atmel ATmega devices. Other architectures will be considered as a future work.

2.4 SμV Toolchain Modifications

SμV provides a modified toolchain which allows the application developer to write software for the SμV architecture transparently and with the same ease-of-use as the underlying MCU architecture.

2. It is also true for x86's STI instruction [36].

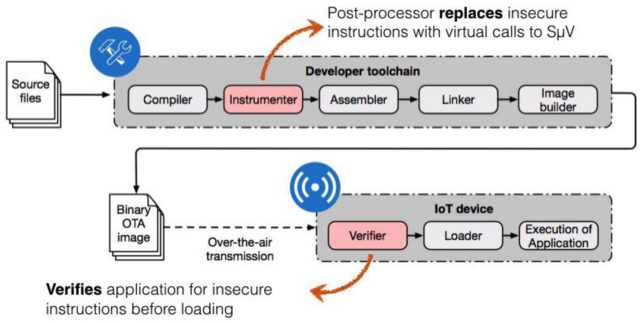


Fig. 2. SμV edited toolchain.

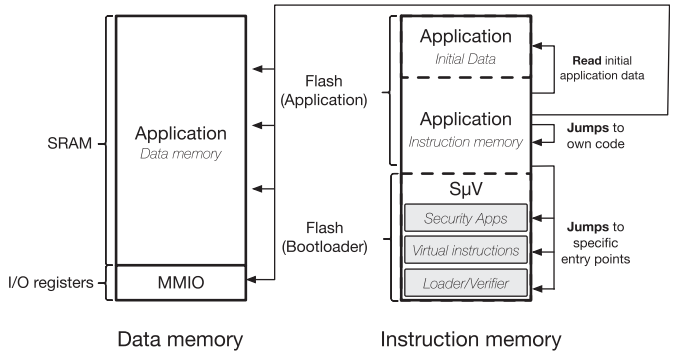


Fig. 3. The AVR's modified Harvard architecture requires an adjustment to the memory map proposed before. Data memory is entirely unprotected, as code can not be executed from there. The SμV is placed in a special bootloader segment at the end of the instruction memory, otherwise the Loader/Verifier can not write to instruction memory when loading new applications.

In a standard toolchain, application code passes through multiple tools: first the compiler produces human readable assembly files. These are processed by the assembler resulting in binary object files. Lastly, the linker combines all object files together with relevant libraries in a single binary image that can be deployed on the microcontroller.

The changes required to this pipeline for SμV are minimal. First, between the assembly and linking stages, we add a post-processor which substitutes all unsafe dynamic instructions with calls to their secure virtualized equivalents. Since this is performed when the application is in text ASM form rather than binary, simple regular expressions will suffice and no custom tools are required. Second, in the linker stage, the addresses of the functions that are considered public *entry points* in the SμV are injected in the form of a symbol table. The SμV is preinstalled on the microcontroller, and the application must be linked against these functions at their given addresses. Fig. 2 describes the modification of the toolchain in SμV.

It should be noted that all libraries must be recompiled with the previously mentioned substitutions carried out. Failing to do so will create an image where unsafe instructions appearing in library functions. These functions will be rejected by the load-time SμV verifier.

Additionally, the security properties of SμV are maintained even when an adversary uses their own tool-chain or writes hand-crafted assembly because the load-time verification of applications occurs by the MicroVisor on the embedded device itself.

3 SμV Implementation

A prototype of SμV has been implemented for the MicroPnP IoT platform [26], which offers an IEEE 802.15.4e [27] radio and an 8-bit AVR ATmega 1284p [37] microcontroller running at 10 MHz, with 16 KB of SRAM and 128 KB of flash. The AVR ATmega family of microcontrollers have a long track record of being used in IoT platforms, including the Zigduino [38], the AVR Raven [39] and Berkeley Mica Mote [40]. In Section 2, we outlined the design and general operation of SμV. Fig. 3 shows an overview of how SμV is implemented on the AVR architecture, which necessitated several adjustments as outlined below.

3.1 Modified Harvard Architecture

In Section 2, we assumed microcontrollers with the generic *von Neumann* architecture, where flash, RAM and any MMIO peripherals are mapped on to a single address space. The AVR family of microcontrollers use a *modified Harvard* architecture, where instruction and data memory are physically separate. In the case of the AVR, this means that the flash memory holding the instructions and the volatile RAM containing the data have an isolated address space. In a strict Harvard architecture, the instruction memory cannot be read from or written to, and instructions cannot be executed from data memory. However, the AVR uses a *modified* Harvard architecture, which relaxes this restriction by offering special instructions to read and modify instruction memory. This allows self programming and storage of data constants in flash.

Implications for SμV. Due to the modified Harvard architecture of the AVR, we can simplify our approach to only protect instruction memory, while data memory operations remain unmodified and unrestricted. Hence, SμV data is placed alongside SμV code in instruction memory, which is permitted by the modified Harvard architecture. The application is not allowed to read the SμV code or data from the instruction memory, and is only permitted to jump to itself or select SμV entry points. These restrictions on the instruction memory, in addition to the fact that data memory is not executable are sufficient to offer the same security guarantees previously mentioned. Leaving the data memory unprotected has the additional advantage that operations dealing with data memory do not incur any runtime overhead due to SμV. Fig. 3 shows the general memory map of both instruction and data memory, together with the class of operations that are allowed by application code. If code contains disallowed operations, it is rejected by either load-time verification or runtime virtualized operations.

3.2 Bootloader

Instruction memory in the AVR is further subdivided into an application and a bootloader section. Code residing in the bootloader section has special privileges over application code. The application can only read from instruction memory, while the bootloader code can read and write instruction memory. Any self-programming code has to be located in the bootloader. The size of the bootloader section is configurable before deployment of the IoT device using a physical programmer, but may not be modified at runtime.

Implications for SμV. The *Loader/Verifier* component of SμV requires write privileges to instruction memory. Therefore the natural place of the SμV is in the bootloader section of the instruction memory. The application is loaded in the application section, and as a result has no write privileges. This is convenient, as we do not need to enforce this through virtualized instructions. The read behaviour of the application still needs to be monitored, however, applications should not be able to read SμV instructions or data from the bootloader section. Fig. 3 shows the placement of the SμV core in flash memory.

3.3 Initialization of Data Memory

At boot time, the volatile data memory is empty. In order to comply with the C standard, variables with an initializer should have their value assigned before any code execution. In order to do this, the instruction memory will hold all initial data memory values in addition to the application instructions. Immediately before the application executes, bootstrapping code will copy the initial values from instruction memory to data memory.

Implications for SμV. Special care should be taken that no jumps from the application code to the initial data stored in instruction memory are made. The data stored in instruction memory may include illegal instructions that could be misused by an adversary to attack SμV. As the compiler always places the initial data in instruction memory straight after the application's instructions, only the address of the last valid instruction should be transmitted as extra metadata at load time. Any jump after that address is either invalid or a bootloader entry point. This is also visualized in Fig. 3.

3.4 Two Word Instructions

The AVR has a variable length instruction set. A standard AVR instruction is 2 bytes (1 word) long. As a result, the program counter and all jumps can only point to even bytes in the flash. Some instructions however require 2 words, with the 2nd word being a target address in either data or instruction memory.

Implications for SμV. There is a possibility that the 2nd word of an two word instruction unintentionally forms an unsafe normal length instruction. While these unsafe 2nd words appear inside the application's instructions, they should not be jumped to. This is accomplished by maintaining a list of unsafe 2nd words, which is enforced by both the load-time verification for static branches and jumps, and by the run-time virtualized instructions for dynamic jumps. While it is possible for SμV to generate a list locally on the node when the application is updated, we add an extra step to the tool-chain to pre-generate this list at compile-time. The list is added to the metadata shipped within the application image, and is checked by the SμV for validity at load-time. Applications with an incomplete list of unsafe 2nd words are rejected. Pre-generating the list reduces the overhead of loading and verifying a new application and moves it from the device to the machine generating the application image.

4 Formal Verification

The safety and security of today's omni-present bare-metal embedded systems critically depend on either a secure hardware module or the reliability of the Trusted Software Module that is installed in the memory of embedded devices. As studies show, most defects causing such systems to crash are either implementation and design errors or software extensions by third parties [41], [42]. SμV relies on selective software virtualization and assembly-level code verification to ensure full isolation of the TSM from other untrusted software modules. Obviously, errors in the design and implementation of SμV can result in code that may violate the SμV rules mentioned in Section 2.3. One way to ensure reliability and guarantee strong isolation is to prove that the code of the SμV itself is crash-free and memory-safe. Therefore, we employ VeriFast [23], a separation logic-based program verifier that relies on the Z3 theorem prover [43], to prove the aforementioned properties.

In the following, we briefly introduce VeriFast and then elaborate on the properties that have been verified.

4.1 Preliminary

With regards to the aforementioned attacker model (See Section 2), ensuring full reliability and high assurance of the strong isolation enforced by SμV involves formally verifying the correctness of both the design and implementation. The design of SμV share similarities with SFI [34], whose correctness has been formally verified in [44], [45]. Therefore, verifying the design of SμV is beyond the scope of this paper. Moreover, we assume that the actual MCU architecture, where we implement SμV atop of, is correctly designed and implemented by the manufactures. To this end, we only formally verify memory safety, and freedom from crash properties, to provide an acceptable level of reliability of the SμV code, that has been only implemented for AVR architecture.[3] We consider the verification of other properties (i.e., semantic correctness) to have fully verifiable system as a future work. On the other hand, formal verification can be achieved via either Theorem Proving or Model Checking. In this work, we use the former, as VeriFast is based on the Z3 theorem prover [43]. Hence, we assume that the implementation of VeriFast itself is correct and bug-free.

4.2 VeriFast

VeriFast [23] is a sound[4] and modular software verifier for C and Java programs. It is based on separation logic [46] that extends Hoare logic [47]. VeriFast checks and verifies certain safety properties of annotated C files. The annotations contain pre- and post- conditions expressed as separation logic assertions, ghost data structures (i.e., predicates), and ghost lemmas. The verification process in VeriFast is based on modular symbolic execution of the software module, where the body of each function of the program is executed symbolically, starting from the symbolic state identified by

3. As mentioned later on, future direction of this work is porting the design and implementation of SμV to Von Neumann architecture.

4. By "sound", we here mean "sound in the absence of bugs in the tool", in contrast to tools that are unsound by design, such as most bug finding and bounded model checking tools where false negatives are expected.

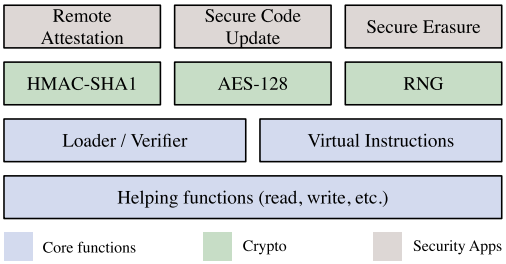


Fig. 4. The sof-tware architecture of the SμV C Module.

the function's preconditions, going through checking permissions of each memory location accessed by any statement inside the function, updating the symbolic state, and ending up with a symbolic state that should meet the function's postconditions. The safety property is verified if both pre- and post- conditions and any additional annotations are satisfied during the symbolic execution of each function. Thus, if VeriFast deems a program to be correct, then that program does not exhibit certain run-time errors that would violate the safety property.

4.3 Verification of SμV

4.3.1 Overview

The software architecture of the security MicroVisor consists of two parts:

- An extended toolchain consisting of a cross-compiler, linker and standard libraries that are needed to produce the executable binary file.
- A software module (written in C) that is installed in the memory of the embedded device before deploying any user application.

As illustrated in Section 2.4, the modifications of the standard toolchain are minimal which limited to identifying simple regular expressions that substitute all unsafe dynamic instructions with safe virtualized ones. The editing takes place using GNU Make [48]. The backbone component is the SμV C module that enforces access control and permits deploying third-party applications only if they comply with the rules described in Section 2.3. We formally verify the code of this module only. The addition of the toolchain is unverifiable as it is a set of commands written in GNU. However, we assume an untrusted toolchain. So, even if the toolchain behaves unexpectedly (e.g., by producing a binary file that contains unsafe instructions), the verified SμV on the target device will reject the application as long as it violates the rules. Hence, verifying the aforementioned properties only of this part is enough to ensure safety and guarantee an acceptable level of trust and reliability. Fig. 4 describes the software architecture of the SμV. The two layers at the bottom are the core functions that provide memory isolation and enforce access control. The top two layers are extensions for various security primitives. They are designed in a modularized way where modules are loaded and unloaded as needed. Among the various security services shown at the top layer, we consider only remote attestation in this paper, while we leave the rest for future work.

The verification of SμV is done by writing formal specifications through adding annotations to all function

```
...
typedef struct {
        uint32_t h[5];
        uint64_t length;
} sha1_ctx_t;

void sha1_init(sha1_ctx_t *state)
        //@ requires state_pred(state,?p,_,_,_,_,_,_);
        //@ ensures state_pred(state,p,0x67452301,
            ↪ 0xefcdab89,0x98badcfe, 0x10325476,
            ↪ 0xc3d2e1f0,0);
{
        //@ open state_pred(state,p,_,_,_,_,_,_);
        state->h[0] = 0x67452301;
        state->h[1] = 0xefcdab89;
        state->h[2] = 0x98badcfe;
        state->h[3] = 0x10325476;
        state->h[4] = 0xc3d2e1f0;
        state->length = 0;
        //@ close state_pred(state,p,0x67452301,
            ↪ 0xefcdab89,0x98badcfe, 0x10325476,
            ↪ 0xc3d2e1f0,0);
}
...
```

Listing 4. Simple example of using contracts in SμV

headers and any statement that is inside the bodies of these functions and performs an operation that requires permissions or affects the status of the memory (e.g., read, write, etc.). These annotations consist of contracts (pre- and post-conditions written in separation logic), predicates to describe data structure, lemmas and auto-lemmas (i.e., ghost functions), and ghost-code like folding and unfolding predicates.

4.3.2 Formal Specifications

We begin by annotating the code to indicate what it should do, and the verifier (in VeriFast) will then check whether the code complies with these annotations, providing proof steps that are automatically generated. Separation logic, the heart of VeriFast, is based on the concept of permissions [49]. That is, each activation record (created by the symbolic execution of the annotated SμV code) holds a number of permissions, where accessing a specified memory location is granted if it holds the corresponding permission to do so. These permissions are identified through the various annotations. For any illegal access or operation, where there is no authorization to perform it, VeriFast reports an error.

Function Contracts. For each function in the SμV, a contract is identified, which is a formal specification of what the method guarantees. In particular, the contract consists of pre-conditions, specified by the keyword *requires*, and post-conditions, specified by the keyword *ensures*. The pre-conditions describe the permissions that the function requires in order to execute successfully. For example, in Listing 4, to initialize the state of the hash function, the function requires the permission to access *struct sha1_ctx_t*. This permission is granted by the pre-conditions identified, which let the function access the corresponding *struct* regardless the current values of its elements, where the symbol_means that the function does not care about what values are stored in the corresponding variable. The post-conditions describe the permissions that are transferred from the function to its caller when the function executes. According to Listing 4, the function should end up with a specified case, in which the elements of the *struct sha1_ctx_t*

```
predicate state_pred(sha1_ctx_t *state, uint32_t *p,
    ↪ uint32_t h0, uint32_t h1, uint32_t h2, uint32_t
    ↪ h3, uint32_t h4,  uint64_t theLength) =
p == state->h &*& *p |-> h0 &*& *(p+1) |-> h1 &*& *(p+2)
    ↪ |-> h2 &*& *(p+3) |-> h3 &*& *(p+4) |-> h4 &*&
    ↪ state->length |-> theLength;
```

Listing 5. An example of a predicate in SμV

are filled in with custom values. Failing to do so will result in errors generated by VeriFast once the function is called. So, the contract says that the *sha1_init* function can access the memory locations of the corresponding *struct* and update the values of these locations. To this end, for each function call, the verifier in VeriFast verifies whether the implementation of that function satisfies its contract or not via symbolic execution. As it is shown in the example, all annotations in VeriFast are written inside special comments (/ * @...@ * / or //@...) which are ignored by the C compiler but recognized by the verifier (in VeriFast).

Predicates. To abstract over the set of permissions required by a function, permissions can be grouped and hidden via predicates. Predicates are named and parametrized assertions. In Listing 4, we use predicates in the contracts to hide the complexity of formalizing permissions needed. Listing 5 describes the body of the predicate *state_pred* that groups and hides the permissions to access the various elements (e.g., each record of the h array, and the variable *length*) in *struct sha1_ctx_t*. Considering that the permission to access (i.e., read and write) the field f of an object o with value v is denoted as $o.f \mid\!\!-> v$, the predicate indicates the permission to access any memory location of all elements attached as parameters along with the corresponding data structure, i.e., *struct*.

Predicates must be folded and unfolded explicitly using ghost statements, i.e., *open* and *close*, as shown in Listing 4. Verification of the code snippet in Listing 4 fails if any of the ghost statements is removed.

Loop Invariants. For each loop in any function of the SμV, we identify a loop invariant, a requirement by VeriFast, so that it can verify an arbitrary sequence of loop iterations by verifying the loop body once, starting from the initial symbolic state. To update the state in Listing 4, we have to iterate over the h array and update each record there. Thus, Loop invariants are required. This is clarified in Listing 6, in which, we use a predefined predicate by VeriFast, in the form of an *array assignment*, inside the loop invariant to control the size and values of the integer array. In the definition of any (pre-defined) predicate, some other pre-defined predicates might be used. Different types of lemmas are used in Listing 6 as we explain in the next section.

Lemmas and Auto-lemmas. (Auto)-Lemmas are ghost C-Like functions with the exception that they do not perform field assignment or call regular C functions. They are used for a customized need where other annotations cannot serve. The difference between auto-lemmas and lemmas is that the latter has to be called explicitly before the corresponding statement while the other is called by the verifier in VeriFast when needed. The major goal of lemmas is to transform some actual memory values into equivalent ones using other data types. For example, in Listing 7, the lemmas, i.e., *shiftRightNoUnderflow*, are used to verify that there is no arithmetic underflow once the arithmetic or bit-wise

```
/*@
        lemma void getStatePredFromElements(sha1_ctx_t
            ↪ *state)
            requires chars(?x, 20, _) &*&
                ↪ sha1_ctx_t_length(state,
                ↪ ?theLength) &*& x == (char *)
                ↪ (state->h);
            ensures state_pred(state,?p,?h0,
                ↪ ?h1,?h2,?h3,?h4,theLength);
        {
            chars_split(x, 16);
            chars_split(x, 12);
            chars_split(x, 8);
            chars_split(x, 4);
            chars_to_integer_(x, 4, false);
            chars_to_integer_(x + 4, 4, false);
            chars_to_integer_(x + 8, 4, false);
            chars_to_integer_(x + 12, 4, false);
            chars_to_integer_(x + 16, 4, false);
            close state_pred(state,_,_,_,_,_,_,_,_);
        }
        lemma void assume_ADD_NoOverflow_long( uint64_t
            ↪ x, uint32_t y)
            requires 0 <= x &*& 0 <= y &*& x <=
                ↪ ULLONG_MAX &*& y <= 65535 ;
            ensures x+y <= ULLONG_MAX;
        {
            assume(false);
        }
@*/
...
uint32_t a[5];
uint8_t t;
...
 //@ getStatePredFromElements(state);
//@ open state_pred(state,_,_,_,_,_,_,_,_);
/* update the state */
for(t=0; t<5; ++t)
        /*@ invariant t >= 0 &*& a[..5] |-> _ &*&
            ↪ state->h[..5] |-> _;  @*/
{
        uint32_t temp_h = state->h[t];
        //@ produce_limits(temp_h);
        uint32_t temp_a = a[t];
        //@ produce_limits(temp_a);
        state->h[t] = /*@ truncating @*/ (state->h[t] +
            ↪ a[t]);
}
//@ assume_ADD_NoOverflow_long(state->length, 512);
state->length += 512;
//@ close state_pred(state,_,_,_,_,_,_,_,_);
...
```

Listing 6. An example of a Loop invariant in SμV

operations are performed in the *rotl32* function. The body of each lemma is a proof of its outcome.

4.3.3 Verified Properties

The following properties are verified by the annotated version of the SμV through VeriFast.

Memory Safety. We prove that the SμV does not exhibit any undefined behavior as described by the C11 standard [50]. In particular, SμV is free from the following run-time errors: division by zero, integer overflow/underflow, buffer over/underflow, out-of-bounding array indexing, invalid pointer dereferences, illegal memory accesses, use after free, double free, problematic bit shifts, type conversions that would overflow the destination, and memory leaks.

Freedom from Crashes. Crash-free SμV is guaranteed at two levels. First, absence of run-time errors, that is verified in memory safety property, ensures absence of crashes as there is no segmentation faults (e.g., attempting to write read only memory), or exceptions (e.g., division by zero). Second, all SμV functions start execution by disabling all interrupts and

```
/*@
        lemma void shiftRightNoUnderflow( uint32_t x,
            ↪ uint8_t y)
                requires y >= 0 &*& x >= 0 &*& y <= 255
                    ↪ &*& x <= UINT32_MAX;
                ensures x >> y >= 0;
        {
                shiftright_limits(x, N32, nat_of_int(y));
        }
        lemma void shiftRight32NoOverflow( uint32_t x,
            ↪ uint8_t y)
                requires y >= 0 &*& x >= 0 &*& y <= 255
                    ↪ &*& x <= UINT32_MAX;
                ensures x >> y <= UINT32_MAX;
        {
                shiftright_limits(x, N32, nat_of_int(y));
                pow_nat_nat_minus(2, N32, nat_of_int(y));
        }
        lemma void OR_NoOverflow( uint32_t x, uint32_t y)
                requires y >= 0 &*& x >= 0 &*& x <=
                    ↪ UINT32_MAX &*& y <= UINT32_MAX;
                ensures ((uint32_t)(x | y)) <= UINT32_MAX;
        {
                Z zx = Z_of_uint32(x);
                Z zy = Z_of_uint32(y);
                bitor_def(x, zx, y, zy);
                Z zt = Z_of_uint32(UINT32_MAX);
                Z_of_uint32_ltOReq(Z_or(zx,zy), zt);
        }
...
@*/
uint32_t rotl32(uint32_t n, uint8_t bits)
        //@ requires bits <= 32;
        //@ ensures 0 <= result &*& result <= UINT32_MAX;
{
        //@ produce_limits(n);
        //@ produce_limits(bits);
        uint32_t result = 0;
        uint32_t temp1 = /*@ truncating @*/ (n << bits);
        //@ shiftRightNoUnderflow(n , 32-bits);
        //@ shiftRight32NoOverflow(n , 32-bits);
        uint32_t temp2 = n >>(32-bits);
        //@ produce_limits(temp1);
        //@ OR_NoUnderflow(temp1,temp2);
        //@ OR_NoOverflow(temp1,temp2);
        result = (temp1 | temp2);
        return result;
}
```

Listing 7. An example of using Lemmas in SμV

end up with enabling them. This atomicity feature sponsors preventing failures through scheduled interrupts.

4.3.4 Annotation Overhead

The security MicroVisor consists of 510 lines of code. In order to verify it, we added 372 lines of VeriFast annotations (or about 0.73 line of annotation for every line of code). The classification of these annotations is as follows: function contracts needed 60 lines of annotations (there were 30 pairs of requires/ensures, one pair for each function), only 1 predicate is defined throughout the entire code base, 16 loop invariants, 12 pairs of open/close ghost statements, and 38 specific (auto-)lemma's are identified and employed in the verification process. Furthermore, various built-in predicates and (auto-)lemmas are used too in verifying the SμV functions.

During verification, 4 bugs related to integer overflow and invalid pointers were discovered and fixed. In particular, one of these bugs could cause memory leaking of secrets. Moreover, initially, VeriFast did not pass some lines of code where integer overflow/underflow cases were likely to appear. In order to fix errors and overcome all warnings from VeriFast, along with annotations, around 39 lines of code had to be added (where some of them were just the result of splitting one statement to two or more statements).

5 Security Analysis

In the following, we informally analyze the security of SμV architecture, and show its robustness against the attacker model presented in Section 2.1.

Reliable Isolation. The main goal of SμV is to provide memory isolation, by which the TSM is sandboxed in a secure memory area and cannot be tampered with by any untrusted software module. Memory isolation is guaranteed by verifying *memory safety* and *control flow integrity* properties. The former has been verified as explained above. The latter is also verified by combining the memory-safety with atomicity property, as all SμV functions execute atomically by disabling global interrupts at the beginning and enabling them upon return (see Section 2.2). Thus, the execution may not scape a predetermined control flow graph, enforcing strong isolation.

Security of SμV Code. Assuming that the underlying MCU architecture is correctly designed and implemented, considering the verified memory-safety and absence of crashes properties (see Section 4.3.3), recalling that the design of SμV is inspired by the design of SFI whose correctness has been formally verified (see Section 4.1), and taking into account that all instructions of SμV are statically allocated in the memory conforming to the specifications of memory-safety property, the SμV code is secure against all remote memory corruption and code injection attacks (see Section 2.1), as it is free from all runtime vulnerabilities (see Section 4.3.3). Thus, the only possibility left for an adversary is to perform her attack exploiting a vulnerability in the untrusted software module running on the same device. We show the robustness SμV, in such case, in the following paragraph.

Robustness of Entire SμV Architecture. To exploit a vulnerability in the untrusted software module and execute any type of code injection or memory corruption attacks, an attacker must find a bug that can simplify overwriting, deleting, or compromising a sensitive, i.e., protected, memory area. Any untrusted software that runs on a SμV-enabled IoT device should be deployed and verified through the verifier software module of SμV on the IoT device itself. If it has unsafe instructions, it will be rejected outright. Any instruction is considered unsafe if it attempts, in any way, to compromise the SμV memory, i.e., breaking isolation. Thus, the installed software could be buggy but not violate memory isolation. Accordingly, any existing bug in the running software will not violate the data-access and control-flow policies that are enforced by SμV as shown in Fig. 1. For example, assuming that the running untrusted software has a buffer overflow vulnerability, the attacker will only be able to inject code in the non-executable data memory, as it is the only memory the untrusted software can write, rendering it useless and being treated as data. Furthermore, the security of SμV is maintained even when an adversary uses her own toolchain or write hand-crafted assembly because the verification process run at load-time on the device itself, assuming an untrusted compiler toolchain. In short, during deployment, any untrusted software

cannot replace the SμV or bypass the verifier function (that is part of the SμV) since the entire over the air deployment is a built-in module of the SμV.

Likewise, memory copy and compression attacks are prevented since the adversary, who is always bounded to the capabilities of untrusted software deployed, cannot write any executable memory. Furthermore, arbitrary code execution attacks are mitigated as all instructions with indirect addressing mode are checked by the SμV at runtime, thus, preventing the adversary from exploiting the right of writing any memory-mapped IO register.

That is, the entire memory-safe and crash-free SμV architecture is truly secure against the powerful remote attacker mentioned in Section 2.1.

6 Remote Attestation

Remote attestation (RA) is an interactive protocol that allows a trusted entity, denoted as a Verifier to check the internal state of an untrusted IoT device, denoted as a Prover, remotely over the network. The purpose of RA is to allow an uncompromised Prover to create a token that proves to the Verifier that it is indeed uncompromised and in an expected internal state. Conversely, if the Prover is compromised, the token should reflect this. A more formal definition is given by Francillon et al. [51]:

Definition of Remote Attestation: A protocol $\mathcal{P}$ is comprised of the following components:

- Setup(1^κ): A probabilistic algorithm that, given a security parameter 1^κ, outputs a long-term key k. This key is shared between both parties, and is preinstalled on the Proverduring commissioning.
- Attest(k, n, s): A deterministic algorithm used by the Prover that, given a pre-shared key k, a nonce n (provided by the Verifier) and internal state s, outputs an attestation token α.
- Verify(k, n, s, α): A deterministic algorithm used by the Verifier that, given a pre-shared key k, a nonce n, an internal state s, and an attestation token α, outputs 1 iff α reflects state s in the Prover (i.e. Attest$(k, n, s) = \alpha$), and outputs 0 otherwise.

These components are used in the protocol $\mathcal{P}$ between Verifier and Prover as follows:

1) Both Prover and Verifier possess a pre-shared long-term key k generated by Setup(1^κ) prior to deployment.
2) Verifier requests proof of the state of Prover by generating and sending an authenticated nonce, n.
3) Prover runs Attest(k, n, s), where s is the current state, returning the resulting attestation token α to Verifier.
4) Verifier runs Verify(k, n, s, α), where s the expected state. The output of Verify proves state s of Prover.

Applying SμV to Support Secure Interactive Remote Attestation. Francillon et al. [51] further analyze protocol $\mathcal{P}$ and its security characteristics, and define a list of minimal properties that are required to support remote attestation, for which they argue that specialized hardware is necessary. Depending on the reliability of the formally verified SμV software module (See Section 4), we believe that SμV can provide an equivalent support in software. We go over the list and explain how the MicroVisor accomplishes this.

1) *Invocation from Start*: The Attest routine should only be invoked from its first instruction. This is accomplished by placing it in the SμV memory area and allowing it to be called from the application when attestation is required. As with all other routines residing in the SμV, only a call to the entry point is allowed, forcing Attest to be run from the very first instruction.
2) *Exclusive access to secret k*: The secret k should only be accessible by the trusted remote attestation code. This is achieved by storing it in the SμV memory area alongside the attestation code. The untrusted application cannot read from this memory area.
3) *Uninterruptibility*: Even on a single threaded platform, the untrusted application may regain control after invoking Attest using interrupts (e.g., a timer expiring). This can cause unintended side effects such as the leaking of k and false positives. All major microcontroller families allow global interrupts to be temporarily turned off. This functionality is used to ensure atomic execution of Attest.
4) *Immutability*: The Attest code cannot be modified by untrusted code before invocation. This is guaranteed by placing the code in the SμV virtual ROM and executing it in-place. Untrusted application code is not allowed to modify this memory area.
5) *No leaks*: Under no circumstance should invoking Attest leak the secret k or any by-products except for the final return value α. This is guaranteed by above properties and additionally implementing the Attest routine in a way that erases these sensitive values from memory before returning.

From a practical point of view, Attest and Verify depend on computing a Message Authentication Code (MAC) of the state to be attested. The contents of flash, RAM memory, registers and any other volatile or non volatile memory can be considered state. When the Prover receives a request from the Verifier to attest a region of memory containing state s with nonce n, Attest is called to compute the MAC α of $(s||n)$ using pre-shared key k. The nonce n should be used only once and is essential to avoid replay attacks. The computed token α is sent back to the Verifier, where Verify computes the MAC of $(s||n)$ once again, this time with s the expected state of the segment of memory. If the computed MAC matches token α, the Prover has the expected state. If the MAC differs, the Prover's memory is compromised and necessary measures should be taken such as performing secure erasure [52].

Security of RA. Assuming that all cryptographic primitives, i.e., MAC scheme, used are secure and selective forgery resistant, the security of RA is inherited from the SμV architecture. That is, the RA scheme is secure w.r.t. the attacker model mentioned in Section 2.1.

6.1 Implementation of RA

As discussed previously, remote attestation requires the computation of a Message Authentication Code over the state of memory of underlying device. In our implementation, we use a highly optimized HMAC-SHA1 implementation which

returns a 160 bit keyed hash, and the pre-shared cryptographic key we use is also 160 bits long.

The selection of HMAC-SHA1 deserves extra clarification after the recent publication of SHAttered, a first real-world SHA1 collision resulting from joint work of Google and CWI Amsterdam [53]. Theoretical attacks on SHA1 have been known since 2005, but SHAttered is the first practical attack on the hashing algorithm. While SHA1 is no longer collision free, HMAC is significantly less sensitive to collisions of the underlying hash algorithm. To the best of our knowledge, HMAC-SHA1 is still secure and not breakable. Another prime example of HMAC's added resistance to collisions is HMAC-MD5. MD5 has known flaws since 1996, but HMAC-MD5 is still collision free today. Taking this in consideration, this work has no strong dependency on HMAC-SHA1 specifically and will preventively transition to another HMAC protocol (e.g., SHA-256) after formally verifying its code module to make sure that it is memory-safe and crash-free. HMAC relies on a pre-shared key. While a solution with public-key cryptography using digital signatures may give better security guarantees and facilitate key management, overheads in computation time and binary code size have proven to make this approach unfeasible for the low-power devices we target.

On the AVR platform, state can be stored in flash, SRAM, CPU registers and EEPROM. While remote attestation can be used on any of these memory types, in our implementation we focus specifically on attesting the flash memory. The reasons for this are twofold: (i) flash is the only memory from where code can be executed on a Harvard architecture, and as a result poses the biggest security risk when compromised, and (ii) the flash memory on the ATmega 1284p is the largest and slowest memory, and thus provides a good benchmark for worst-case performance.

7 Evaluation

We evaluate SμV by implementing reference applications and measuring the overhead imposed by the Security MicroVisor as well as the remote attestation service. More specifically, we focus on three key performance indicators: (i) development overhead, (ii) application deployment overhead, and (iii) runtime overhead: execution time, battery life, and memory footprint. For every performance metric we first consider the overhead imposed by SμV itself, before analyzing the overhead incurred by remote attestation.

Four reference applications were selected to benchmark performance: (i) a cryptographic application which encrypts and decrypts a random 8 byte cleartext with a 128-bit key, using a software implementation of the lightweight SPECK block cipher [54], (ii) the same crypto application, but implemented in a modular fashion by introducing indirect calls through pointers, (iii) sampling temperature readings from a SHT25 Sensirion sensor over the I2C bus, and lastly (iv) writing and reading a block of 256 bytes to the built-in EEPROM. We believe that these reference applications provide a good balance between the more computationally intensive and the more IO intensive tasks that are typical for an IoT device. The pointer version of cryptographic application is included to provide the worst case overhead for SμV.

7.1 Development

From the application developer's point of view, no development overhead will be perceived after the installation of the SμV toolchain. The modified toolchain will transparently replace unsafe instructions with secure virtual alternatives, link to the pre-installed Security MicroVisor and generate a binary image with the correct metadata. Any security features embedded in the SμV (i.e., remote attestation, secure deployment, etc.) will be available transparently.

The time required to port SμV to a different architecture may also be considered development overhead. For the purpose of this paper, we implemented and evaluated solely on AVR microcontrollers. The implementation and formal verification of SμV took about 480 man-hours. This includes the software running on the microcontroller itself as well as all additions to the toolchain. The time to port to a different architecture is likely to depend upon its inherent characteristics. For example, the variable length of the AVR instruction set causes more overhead than a fixed length instruction set due to the extra bookkeeping mechanisms required. Furthermore, architectures similar to AVR can borrow parts of this initial version of SμV, reducing porting overhead.

7.2 Deployment

In this benchmark, we compare the size of the image for a microcontroller without SμV, with the size of an image of the same application for a microcontroller with SμV, once deployed using a physical programming device (e.g., JTAG). Table 1 shows the results for each of the reference applications previously introduced. The size overhead for the SμV-enabled images are relatively small and average at 1.61 percent. This overhead is caused by two different effects: i) on the AVR, unsafe single word instructions are replaced with a 2 word long instruction that calls a safe virtualized version residing in the MicroVisor, and ii) the SμV-enabled image carries a small amount of extra metadata, such as a list of unsafe 2nd words and the address of the last valid application instruction.

Loading/Verification Time. For a microcontroller without SμV, verification is a simple routine that checks if all parts were transmitted, and in case of unreliable network communication all chunks will be check-summed to rule out a corrupted image. On a SμV-enabled microcontroller, verification additionally includes a static check of all instructions of the application, and a validity check of the metadata transmitted with the image (i.e., the address of last valid instruction and a list of unsafe 2nd words).

Table 2 shows local verification and load times for all reference applications, both with and without the Security MicroVisor.

The relative overhead introduced by the extra verification of SμV for our test applications averages at 4.16 percent, which is minimal.

7.3 Runtime

Finally, we evaluate runtime overhead in terms of execution time, RAM overhead and flash overhead.

Execution Time. Execution time is directly correlated with the battery life of low power microcontrollers. Typically, after the application is done with its tasks, the microcontroller is

TABLE 1
Overheads of SμV-Enabled Binary Image Sizes

Application	Without SμV	With SμV
Crypto	1414 B	1428 B (**+0.99%**)
Crypto ptr	1438 B	1458 B (**+1.39%**)
Sense temp	1012 B	1034 B (**+2.17%**)
Storage R/W	640 B	652 B (**+1.88%**)
Avg overhead		**1.61%**

put into a sleep mode to minimize current draw. The execution time of the application determines how much time is spent in the high current active state.

We once again compare the execution time of all reference applications with and without SμV, and display the results in Table 3. On average, the relative execution time overhead amounts to 2.67 percent. However, there is a difference between the more computationally intensive tasks (i.e. Crypto), and the more IO intensive tasks (i.e., temperature sensing and storage read/write). The IO intensive tasks spend a relatively large amount of time busy waiting for operations to complete, while the computationally intensive tasks are continually executing instructions. Due to this, relative overheads for computational tasks are higher than for IO intensive tasks.

As we have previously argued, the execution time of tasks is directly correlated with energy consumption. Longer execution times cause the microcontroller to remain in a higher power state for a longer amount of time, draining the batteries faster. The MicroPnP platform on which we are conducting the benchmarks consumes 3.54 mA when executing a task and 54.5 μA when idle. Every MicroPnP device is powered by a standard 3000 mAh battery pack. Based upon these values, we plot an estimation of the device battery lifetime for any task relative to the rate at which it is scheduled. Note that the network is a constant in this benchmark as no data is transmitted and only the energy consumed by the node-local code execution is considered.

Fig. 5 shows the impact of SμV for the worst-case CPU intensive *Crypto pointer* application, and the more IO intensive *Sense temperature* application. The baseline battery lifetime if the application is sleeping constantly is 6.5 years. In general, the *Sense temperature* application has worse battery life and produces a more S-shaped curve. This is due to the busy waiting that is associated with IO intensive tasks, keeping the microcontroller in an active state for a longer time. When comparing identical applications with and without SμV, we can see that for the CPU intensive *Crypto pointer* application there

TABLE 2
Overheads of SμV-Based Deployment Verification

Application	Without SμV	With SμV
Crypto	128.2 ms	133.8 ms (**+4.3%**)
Crypto ptr	128.6 ms	134.3 ms (**+4.4%**)
Sense temp	91.8 ms	96.0 ms (**+4.6%**)
Storage R/W	73.5 ms	75.9 ms (**+3.3%**)
Avg overhead		**4.16%**

TABLE 3
Overheads of SμV on the Execution Times of the Sample Applications

Application	Without SμV	With SμV	Relative overhead
Crypto	3.9460 ms	4.1695 ms	5.66%
Crypto ptr	3.9469 ms	4.1706 ms	5.67%
Sense temp	65.9048 ms	65.9364 ms	0.05%
Storage Write	859.6278 ms	859.6897 ms	0.01%
Storage Read	0.4382 ms	0.4468 ms	1.96%
Avg overhead			**2.67%**

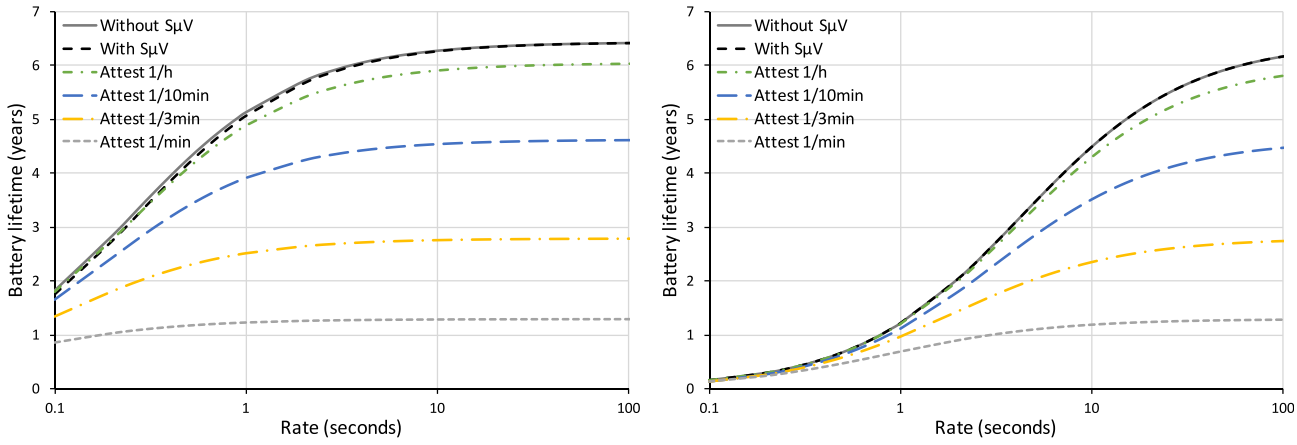
is a marginal overhead (<1 percent) at high scheduling rates, which disappears when the application is only scheduled once every 10 seconds. For the IO intensive *Sense temperature* application, SμV overheads are imperceivable on the graph whether the application is scheduled every 100 ms or every 100s. In most real world IoT applications, long IO intensive tasks will dominate over brief CPU intensive tasks, making any measurable reduction of battery life caused by SμV unlikely.

Remote attestation causes additional active CPU time, and as a result an increase in energy consumption. Remote attestation execution times depend on the amount of flash memory attested. Attesting the entire 128 KB of flash memory takes 7.6 seconds, attesting just the untrusted application (62 KB) takes 3.7 seconds. The impact on battery life largely depends on the frequency of remote attestation. Fig. 5 shows the battery lifetime of our reference applications when remotely attested at rates ranging from once every minute to once every hour. For both applications, an attestation rate of once every hour incurs a worst-case reduction in battery lifetime of 6.2 percent. The effect of a higher attestation rate is less prominent for applications with a higher sampling rate, where the energy consumption of the primary task overshadows attestation energy consumption.

RAM Memory. SμV incurs no static RAM overhead as all constants are stored in flash. The stack is used for short term data storage when calling any subroutine in the SμV (i.e., any virtual instruction, remote attestation, etc.). In order for these subroutines to properly function, the application can not use all available stack space. For correct basic operation of the microcontroller, a minimum amount of 13 bytes of free stack space is required at all time for the virtual instructions. Remote attestation inherently needs more stack space to temporary store full pages of flash and maintain intermediate states of the cryptographic functions, and requires about 511 bytes. This is a compromise where we trade memory usage for speed. Loading pages in smaller chunks is possible, but more time consuming.

Flash Memory. When evaluating deployment overheads, we quantified the extra size of the binary image to be installed in flash. In addition to these overheads, less total space will be available in flash due to space reserved for the preinstalled SμV.

The reduction of available space is shown in Table 4. The total space occupied by the SμV can be broken down in its components. The *SμV core* contains everything required to ensure memory isolation: the verifier, the loader, secure virtualized operations and any required data constants, and



(a) Crypto pointer application (b) Temperature sensing application

Fig. 5. Plots showing the estimated battery life for a computationally intensive application (Crypto) and an IO intensive application (Sense temperature), relative to the period at which they perform their tasks (*X*-axis). Different curves represent battery life without and with SμV, and of increasing rates of remote attestation.

TABLE 4
Flash Memory Utilization

Component	Size	% of flash 128 KB
SμV core	1070 B	0.82%
HMAC-SHA1	1296 B	0.99%
Remote attestation	242 B	0.18%
Total	**2608 B**	**1.99%**

amounts to 1070 bytes or a marginal 0.82 percent of the total flash memory available. The *Remote attestation* consumes an additional 242 bytes. Lastly, the *HMAC-SHA1* algorithm used by RA consumes 1296 bytes of flash. Total flash used amounts to 2608 bytes or 1.99 percent of the total 128 KB flash available on the platform.

As explained previously, on the AVR architecture the SμV has to be installed in a special bootloader section of flash memory. The bootloader's size is configurable. The Security MicroVisor comfortably fits in the 3th smallest size of 4096 bytes, occupying 63.67 percent of bootloader space.

8 Related Work

8.1 Memory Isolation

Many solutions have been proposed to ensure isolation of code and data, for both high-end and low-end devices. These solutions can be classified to hardware-only or software-only mechanisms.

Hardware-Based Isolation Techniques. ARM TrustZone [55] is a system on ship technology that implements hardware-based access control to isolate security critical applications within the same physical core. More recently, Intel started equipping x86 processors with Software Guard Extensions (SGX) [56] that allows the user to allocate private regions of memory via the use of enclaves, hardware-enforced protected areas of execution in memory. These enclaves can be managed by an untrusted operating system. Furthermore, SGX provides functionality for remote attestation and data sealing [57]. Haven [58] leverages Intel SGX processor extensions to isolate application binaries, seeking to protect largely unmodified legacy applications from an untrusted OS. Other hardware-based techniques are proposed for the same purpose [29], [30], [31], [32], [33], [59].

All of the aforementioned techniques target high-end devices which are not suitable for low-end embedded devices due to the constraints in terms of limited resources and computing power. Therefore, low-end devices have received an attention by some proposals. SMART [11] is a security architecture that enforces isolation between trusted and untrusted software modules through hardware-software co-design. The architecture requires minimal changes to the hardware and provides more functionality such as remote attestation. Sancus [10] is a hardware-only architecture that provides isolation between software modules along with other services such as remote attestation and secure linking. For both architectures, a prototype was developed based on Texas Instruments MSP430 processor. TrustLite [12] is a security architecture that depends on an Execution-Aware Memory Protection Unit (EA-MPU) to provide OS-independent isolation between software modules. Other security applications such as remote attestation are supported in TrustLite without the need to disable the interrupts, compared to other architectures. TyTAN [9] extends TrustLite with more security services in addition to real-time guarantees. [60] provides an extended comparison between hardware-based trusted computing architectures.

Software-Based Isolation Techniques. Software-based approaches were previously proposed to protect computer systems while nowadays they are only used to protect high-constrained low-end devices that cannot support hardware-extensions. Software-based Fault Isolation [34] is the first approach proposed in this domain. It is a software-instrumentation technique at the machine-code level for establishing logical protection domains within a process and thus preventing faulty software modules from affecting other trusted ones. SFI can be implemented in many ways: in a machine-code interpreter, in a machine-code rewriter, or inside a compiler. Harbor [61] focuses purely on providing a software-based Memory Protection Unit to resource-constrained embedded devices by applying SFI sandboxing techniques, and adds no additional security operations. In particular, Harbor provides a hypervisor that is deeply tied

to the SOS operating system [62] to enforce memory isolation, whereas, SμV targets bare-metal devices and can be used as a foundation to build embedded operating systems on top of. Harbor is composed of four components: (i) a binary rewriter, which is a desktop application that takes a binary image generated by the cross compiler and inserts run-time checks before the potentially unsafe instructions, (ii) a binary verifier running on the embedded device itself, ensuring that the incoming binary image is correctly sandboxed, (iii) the Memory Map Manager, which is an abstraction layer incorporated in the SOS operating system that aims to store and retrieve access permissions for a given address, and lastly (iv), a Control Flow Manager, which ensures that control can never flow out of the dedicated domain except when permitted by the rules in the Memory Map Manager. The complexity of the Harbor software stack is much higher when compared with SμV, both in terms of toolchain modifications as on the device itself. Furthermore, the codebase of Harbor is six times larger than SμV. The average execution time overhead is twice as high than SμV running on the same family of microcontrollers.

For an extended discussion about the principles and implementation techniques of software-based isolation, we refer the reader to Reference [45].

8.2 Remote Attestation

Remote attestation is a security service that allows a trusted entity to validate the software integrity of an untrusted remote device. Over the last years, several attestation techniques have been proposed. Such techniques fall into three main categories. First, *hardware-based* attestation [63], [64], [65], [66] uses dedicated hardware features such as a Trusted Platform Module (TPM) [67] to execute the attestation code in a secure environment. Even though such features are currently available in personal computers and smartphones, they are considered a relative *luxury* for very low-end embedded devices.

Second, *Software-based* solutions address the attestation problem without any specialized hardware. SWATT [13] is a time-based attestation technique that relies on response timing to identify a compromised node. The same concept is used in many other software-based solutions [7], [8], [68], [69]. In particular, these methods rely on the estimated upper-bound time required by a given configuration of the prover device to freshly compute the correct answer for the verifier. If the computation takes longer, then the presence of an attacker can be inferred. The inherent limitation of time-based assumptions have been discussed in the literature [70] and several concrete attacks have been also published [41]. Some memory-based attestation techniques are presented in [71], [72]. However, these techniques are not sufficient since they are applied only to RAM, whereas, the goal is to verify all writable memory contents in embedded devices. Attestation techniques based on self-modifying code and obfuscation are introduced in [73]. The main drawback of these techniques is that they are difficult to implement in embedded systems and they do not offer provable security guarantees.

Finally, to cover this end of the spectrum, a new wave of solutions have recently been proposed, which is *hybrid attestation* that depends on the previously mentioned security architectures [9], [10], [11], [12]. Hybrid attestation relies on a software-hardware co-design to provide a secure attestation protocol while minimizing the impact of the underlying hardware features.

SMART [11] is a hybrid hardware/software approach that requires hardware modifications to the memory bus access logic of the microcontroller. SMART isolates and secures remote attestation code by storing it in a virtual ROM inside the flash memory. Both the verifier and the prover share the same secret key prior to deployment for authentication purposes. The key is stored in a secure area inside the CPU utilizing the modified hardware-controlled memory and can be accessed only by SMART code. In line with other approaches, SMART relies on a challenge-response protocol for verifying the internal state of the prover by computing the HMAC of the entire memory. During the execution of the attestation process, interrupts are disabled in order to guarantee atomic execution and avoid the time-of-check-to-time-of-use (TOCTTOU) attack. If an error is detected, a hardware reset of the MCU is performed enforcing memory cleanup. SMART is the closest solution to our approach. However, it relies on hardware modifications while our approach is purely software-based.

SANCUS [10] is a security architecture for IoT devices, which is notable in that it has a pure hardware root of trust. SANCUS provides not only application isolation, but also integrity, authentication, and dynamic remote attestation. SANCUS requires hardware support via the Memory Access Logic (MAL) circuit, which enforces access rights for a single software module to achieve isolation and key protection. This can be guaranteed by having a Trusted Computing Base (TCB) on the targeted resource-constrained device. This is the only hardware needed for SANCUS. In order to minimize hardware costs, SANCUS does not rely on public-key cryptography and rather utilizes symmetric key cryptography, however, this complicates the key management process since the design of SANCUS requires three types of keys for each node connected to the network.

TyTAN [9] extends the TrustLite architecture [12] with dynamic loading, and both local and remote attestation guarantees for isolated software modules from mutually untrusted stakeholders. Compared to SANCUS and SMART, TyTAN adds interruptibility of the attestation process. The process of computing the answer to be sent to the verifier can be interrupted in the middle of its execution without compromising its security. This property may be required in hard real-time applications.

While SMART, SANCUS and TyTAN have much lower hardware requirements than a TPM, they are still difficult to provide in the lowest-end class of microcontrollers (e.g., IETF Class-1 [14]). More importantly, millions of devices are already deployed that would require hardware modification or replacement to implement these solutions. Very recently, a formally verified hybrid-based architecture, named VRASED, has been proposed [74]. VRASED instantiates a verifiable hardware/software co-design for remote attestation, to provide a level of security comparable to that in hardware-based approaches. For an extended comparison between the various remote attestation techniques, we refer the reader to Reference [24].

8.3 Formal Verification

A number of formal verification tools have been developed for verifying applications written in various programming languages. To narrow the focus, we highlight only the tools that support C programming language. Smallfoot [75] is an automatic verification tool that checks separation logic of sequential and concurrent programs that manipulate recursive dynamically-allocated data structures. Space Invader [76] is another static analysis tool that uses separation logic assertions to perform automatic verification of pointer-enabled programs. In [77] a model checker with the support of pointers and bit-vector operations is evaluated on a case study on Linux device drivers. The freedom of buffer overflow and division by zero along with pointer safety are verified by this model. Despite of suffering from several limitations with respect to reasoning about concurrently executing programs, bounded model checking and symbolic execution have been successfully applied to the source code [78] and to the object code [79] of kernel modules. VCC [80] is a competing tool to VeriFast that proves the correctness of annotated concurrent C programs and finds bugs in them. It extends C with design by contract features, where annotated programs are translated to logical formulas using the Boogie tool [81], which passes them to an automated SMT solver Z3 [43] to check their validity.

On the other hand, VeriFast has been used to verify various industrial applications written in either C or Java programming languages, including the Belgian electronic identity card, Linux's USB BP keyboard driver, and a network address translation (NAT) [82].

To the best of our knowledge, we are the first to provide a formally verified software-based security architecture for high-constrained low-end embedded devices using any of the formal verification tools.

9 Conclusion

This paper introduced the Security MicroVisor (SμV), a formally-verified pure software-based security middleware for high-constrained low-end IoT devices. In particular, SμV tackles the problem of lack of hardware memory isolation common to most IETF Class-1 devices. SμV isolates a trusted software module from untrusted application and uses it to contribute to pure software-based security applications, namely remote attestation, secure code update, and secure erasure. Nevertheless, the techniques embodied by SμV may be used to implement other security features. The SμV approach rests on three pillars: (i) selective software virtualisation of the microcontroller architecture, (ii) deployment-time verification of incoming code at the assembly level and (iii) toolchain modifications which allow developers to transparently compile their software for the virtual SμV architecture. Furthermore, SμV is compatible with the vast majority of IoT microcontrollers and, crucially, as SμV does not require additional hardware security features, the approach may be applied to improve the security of millions of IoT devices that are already in the field.

SμV is verified using VeriFast to be memory-safe and crash-free. As such, considering the verified properties, a high level of reliability, that is comparable to hybrid / hardware-based approaches, is achieved, with respect to remote-only attacks.

In our view, the overhead of SμV is extremely reasonable. Our evaluation on the ATmega 1284p shows a modest increase in the size of deployable code at 3.58 percent. The execution time of application code also increases minimally at 2.67 percent, and has little effect (<1 percent) on battery life. Code verification overheads during software updates are likewise feasible for embedded IoT devices, adding an average local overhead of just 4.16 percent. Secure deployment adds additional verification overhead, but end-to-end overhead including network transmissions are acceptable at 8 percent. Hourly software attestation reduces battery life by a maximum of 6.2 percent.

Future Directions. As a future work, we aim to have a fully-verified SμV by verifying other properties such as semantic correctness as well as the design specifications. Furthermore, porting the Security MicroVisor to other architectures such as Von Neumann is another direction for future work. Finally, a comprehensive and systematic comparison of the SμV with other hybrid architectures is an important aspect that we aim to address.

Acknowledgments

This research is supported by the research fund of KU Leuven and imec, a research institute founded by the Flemish government.

References

[1] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 15–26.

[2] T. Giannetsos, T. Dimitriou, and N. R. Prasad, "Self-propagating worms in wireless sensor networks," in *Proc. 5th Int. Student Workshop Emerging Netw. Exp. Technol.*, 2009, pp. 31–32.

[3] M. Kumar, "Ransomware hijacks hotel smart keys to lock guests out of their rooms," 2017. [Online]. Available: https://thehackernews.com/2017/01/ransomware-hotel-smart-lock.html, Accessed on: May 20, 2019.

[4] J. Vijayan, "Target attack shows danger of remotely accessible HVAC systems," 2014. [Online]. Available: https://www.computerworld.com/article/2487452/target-attack-shows-danger-of-remotely-accessible-hvac-systems.html, Accessed on: May 20, 2019.

[5] D. Trends, "Hackers broke into a casinos high-roller database through a thermometer in the lobby fish tank," 2018. [Online]. Available: https://www.digitaltrends.com/home/casino-iot-hackers-fish-tank/, Accessed on: May, 20, 2019.

[6] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 48–62.

[7] 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," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, no. 5, pp. 1–16, Oct. 2005.

[8] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the Integrity of Peripherals' Firmware," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 3–16.

[9] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. 52nd Annu. Des. Autom. Conf.*, Jun. 2015, Art. no. 6.

[10] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 479–494.

[11] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and al architecture for (establishing dynamic) root of trust," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, pp. 1–15.

[12] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 14.

[13] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Proc. IEEE Symp. Secur. Privacy*, May 2004, pp. 272–282.

[14] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," Internet Requests for Comments, IETF, RFC 7228, May 2014.

[15] W. Daniels, D. Hughes, M. Ammar, B. Crispo, N. Matthys, and W. Joosen, "S μ v-the security microvisor: a virtualisation-based security middleware for the internet of things," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.: Ind. Track*, 2017, pp. 36–42.

[16] Microship, "8-bit AVR microcontrollers peripheral integration," 2018. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/30010135D.pdf, Accessed on: May 20, 2019.

[17] NXP, "NXP Microcontrollers overview," 2017. [Online]. Available: https://www.nxp.com/docs/en/supporting-information/BL-Micro-NXP-Microcontroller-Overview-James-Huang.pdf, Accessed on: May 20, 2019.

[18] J. Yiu, "Arm cortex-m for beginners; an overview of the arm cortex-m processor family and comparison," *ARM Limited, White Pap.*, 2016.

[19] I. Research, "GLOBAL IOT MICROCONTROLLER (MCU) MARKET FORECAST 2018–2026," 2018. [Online]. Available: https://www.inkwoodresearch.com/reports/internet-of-things-microcontroller-market/, Accessed on: May 22, 2019.

[20] P. Herald, "Global IoT microcontroller (MCU) market analysis and research report by experts 2024, " 2019. [Online]. Available: https://perfectherald.com/press-release/global-iot-microcontroller-mcu-market-analysis-and-research-report-by-experts-2024/, Accessed May 22, 2019.

[21] GlobeNewsWire, "IoT MCU market to reach USD 4.61 billion by 2026–Reports and data," 2019. [Online]. Available: https://www.globenewswire.com/news-release/2019/03/27/1774199/0/en/IoT-MCU-Market-To-Reach-USD-4-61-Billion-By-2026-Reports-And-Data.html, Accessed on: May 22, 2019.

[22] MarketWatch, "IoT microcontroller market 2019 global analysis, segments, size, segmentation, share, demands, industry growth and recent trends by forecast to 2023," 2019. [Online]. Available: https://www.marketwatch.com/press-release/ iot-microcontroller-market-2019-global-analysis-segments-size-segmentation-share-demands-industry-growth-and-recent-trends-by-forecast-to-2023-2019-03-07, Accessed on: May 22, 2019.

[23] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java," in *Proc.3rd Int. Conf. NASA Formal Methods*, 2011, pp. 41–55.

[24] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey," *ACM Comput. Surveys*, vol. 49, no. 3, 2016, Art. no. 51.

[25] Mahmoud Ammar, "source code of verified security Micro Visor." 2019. [Online]. Available: https://github.com/m3mmar/verified_SuV

[26] F. Yang, N. Matthys, R. Bachiller, S. Michiels, W. Joosen, and D. Hughes, "μPnP: Plug and Play Peripherals for the Internet of Things," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 25.

[27] T. Watteyne, M. R. Palattella, and L. A. Grieco, "Using IEEE 802.15.4e time-slotted channel hopping (TSCH) in the Internet of Things (IoT): Problem statement," Internet Requests for Comments, RFC Editor, RFC 7554, May 2015.

[28] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 335–350, 2007.

[29] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 143–158.

[30] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. 29th Annu. Comput. Secur. Appl. Conf.*, 2013, pp. 289–298.

[31] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 430–444.

[32] Y. Cheng, X. Ding, and R. H. Deng, "Efficient virtualization-based application protection against untrusted operating system," in *Proc. 10th ACM Symp. Inf. Comput. Commun. Secur.*, 2015, pp. 345–356.

[33] S. Zhao and X. Ding, "On the effectiveness of virtualization based memory isolation on multicore platforms," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2017, pp. 546–560.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993.

[35] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *Proc. 17th Int. Conf. VLSI Des.*, 2004, pp. 605–611.

[36] Intel, "Intel 64 and IA-32 architectures software developer's manual - Instruction set reference A-Z," Intel, Tech. Rep., 2016.

[37] Atmel, "AVR ATmega 1284p 8-bit microcontroller," 2009. [Online]. Available: http://www.atmel.com/images/doc8059.pdf, Accessed on: Dec. 10, 2018.

[38] Logos, "Zigduino," 2013. [Online]. Available: http://www.logos-electro.com/zigduino/, Accessed on: Dec. 13, 2018.

[39] Atmel, "AVR Raven," 2007. [Online]. Available: http://www.atmel.com/Images/doc8117.pdf, Accessed on: Oct. 13, 2018.

[40] J. L. Hill and D. E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, Nov. 2002.

[41] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 400–409.

[42] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 73–88, 2001.

[43] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[44] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: Fully verified software fault isolation," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 289–298.

[45] G. Tan, et al., "Principles and implementation techniques of software-based fault isolation," *Found. Trends® Privacy Secur.*, vol. 1, no. 3, pp. 137–198, 2017.

[46] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th Annu. IEEE Symp. Logic Comput. Sci.*, 2002, pp. 55–74.

[47] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[48] R. Mecklenburg, *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. Newton, MA, USA: O'Reilly Media, Inc., 2004.

[49] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 259–270, 2005.

[50] I. Jtc, "Sc22/wg14. iso/iec 9899: 2011," *Information technologyProgramming languagesC*, 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm

[51] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proc. Conf. Des. Autom. Test Eur.*, 2014, Art. no. 244.

[52] M. Ammar, W. Daniels, B. Crispo, and D. Hughes, "Speed: Secure provable erasure for class-1 iot devices," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy*, 2018, pp. 111–118.

[53] Google, "Announcing the first SHA1 collision," 2017. [Online]. Available: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html, Accessed on: May 19, 2018.

[54] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Proc. 52nd Annu. Des. Autom. Conf.*, 2015, pp. 1–6.

[55] T. Alves, "Trustzone: Integrated hardware and software security," *White Paper*, 2004.

[56] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, vol. 10, 2013, Art. no. 10.

[57] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, vol. 13, pp. 1–7, 2013.

[58] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, 2015, Art. no. 8.

[59] A. M. Azab, P. Ning, and X. Zhang, "Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 375–388.

[60] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede, "Hardware-based trusted computing architectures for isolation and attestation," *IEEE Trans. Comput.*, vol. 67, no. 3, pp. 361–374, Mar. 2018.
[61] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-based memory protection for sensor nodes," in *Proc. 6th Int. Symp. Inf. Process. Sensor Netw.*, 2007, pp. 340–349.
[62] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proc. 3rd Int. Conf. Mobile Syst. Appl. Serv.*, 2005, pp. 163–176.
[63] W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symp. Secur. Privacy*, 1997, pp. 65–71.
[64] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 414–429.
[65] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2009, pp. 115–124.
[66] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Sci. Comput. Program.*, vol. 74, no. 1–2, pp. 13–22, 2008.
[67] Trusted Computing Group, "TPM main specification Level 2 Version 1.2," 2011. [Online]. Available: http://www.trustedcomputinggroup.org/tpm-main-specification/, Accessed on: Dec. 10, 2018.
[68] Y. Li, J. M. McCune, and A. Perrig, "SBAP: Software-based attestation for peripherals," in *Proc. 3rd Int. Conf. Trust Trustworthy Comput.*, 2010, pp. 16–29.
[69] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software attestation for key establishment in sensor networks," in *Distributed Computing in Sensor Systems*. Berlin, Germany: Springer, 2008, pp. 372–385.
[70] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *Proc. 13th USENIX Conf. Secur.*, 2014, vol. 13, pp. 89–101.
[71] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *Proc. IEEE 26th Int. Symp. Reliable Distrib. Syst.*, Oct. 2007, pp. 219–230.
[72] Y.-G. Choi, J. Kang, and D. Nyang, "Proactive code verification protocol in wireless sensor network," in *Computational Science and Its Applications (ICCSA)*. Berlin, Germany: Springer, 2007, pp. 1085–1096.
[73] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote Software-Based Attestation for Wireless Sensors," in *Proc. 2nd Eur. Conf. Secur. Privacy Ad-Hoc Sensor Netw.*, 2005, Art. no. 27–41.
[74] K. Eldefrawy, I. O. Nunes, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Formally verified hardware/software co-design for remote attestation," *arXiv: 1811.00175*, 2018.
[75] J. Berdine, C. Calcagno, and P. W. Ohearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *Proc. Int. Symp. Formal Methods Components Objects*, 2005, pp. 115–137.
[76] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn, "Scalable shape analysis for systems code," in *Proc. Int. Conf. Comput. Aided Verification*, 2008, pp. 385–398.
[77] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model checking concurrent linux device drivers," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 501–504.
[78] M. Kim and Y. Kim, "Concolic testing of the multi-sector read operation for flash memory file system," in *Brazilian Symposium on Formal Methods*. New York, NY, USA: Springer, 2009, pp. 251–265.
[79] J. T. Mühlberg and G. Lüttgen, "Verifying compiled file system code," in *Proc. Brazilian Symp. Formal Methods*, 2009, pp. 306–320.
[80] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *Proc. Int. Conf. Theorem Proving Higher Order Logics*, 2009, pp. 23–42.
[81] C. Le Goues, K. R. M. Leino, and M. Moskal, "The boogie verification debugger (tool paper)," in *Proc. Int. Conf. Softw. Eng. Formal Methods*, 2011, pp. 407–414.
[82] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens, "Software verification with verifast: Industrial case studies," *Sci. Comput. Program.*, vol. 82, pp. 77–97, 2014.

Mahmoud Ammar received the MCS degree from La Sapienza University of Rome, in late 2015. He is a PhD researcher with the KU Leuven Computer Science Department, where he is a member of the imec-DistriNet research group. His research interests include Internet of Things (IoT) security. He is particularly interested in issues concerning the physical layer security in class-1 IoT devices.

Bruno Crispo received the PhD degree in security from the University of Cambridge, United Kindom. He is a professor of computer science with the University of Trento, Italy, and visiting professor with KU Leuven, Belgium. His research interests include system and network security, mobile platform security and privacy, and access control. He is an associate editor of the *ACM Transactions on Privacy and Security* and a senior member of the IEEE.

Bart Jacobs is an assistant professor with the imec-DistriNet Research Group, Department of Computer Science, KU Leuven in Belgium. His research interest include logics and tools for formal modular program verification and programming language features for security, correctness, and concurrency.

Danny Hughes is a professor with the Department of Computer Science, KU Leuven, Belgium, where he is a member of the DistriNet (Distributed Systems and Computer Networks) research group and leads the Networked Embedded Software taskforce. He is also the chief technical officer of VersaSense, a KU Leuven spin-off company that provides industrial IoT solutions. His current research is on distributed software systems and the internet of things (IoT).

Wilfried Daniels received the PhD degree in computer science from KU Leuven, Belgium, in 2018. He is a senior developer with VersaSense, a KU Leuven spin-off company that provides industrial IoT solutions.