SμV - the Security MicroVisor: a virtualisation-based security middleware for the Internet of Things

Wilfried Daniels
imec-DistriNet, KU Leuven
wilfried.daniels@cs.kuleuven.be

Danny Hughes
VersaSense
danny@versasense.com

Mahmoud Ammar
imec-DistriNet, KU Leuven
mahmoud.ammar@cs.kuleuven.be

Bruno Crispo
imec-DistriNet, KU Leuven
bruno.crispo@cs.kuleuven.be

Nelson Matthys
VersaSense
nelson@versasense.com

Wouter Joosen
imec-DistriNet, KU Leuven
wouter.joosen@cs.kuleuven.be

Abstract

The Internet of Things (IoT) creates value by connecting digital processes to the physical world using embedded sensors, actuators and wireless networks. The IoT is increasingly intertwined with critical industrial processes, yet contemporary IoT devices offer limited security features, creating a large new attack surface and inhibiting the adoption of IoT technologies. Hardware security modules address this problem, however, their use increases the cost of embedded IoT devices. Furthermore, millions of IoT devices are already deployed without hardware security support. This paper addresses this problem by introducing a *Security MicroVisor* (SμV) middleware, which provides memory isolation and custom security operations using software virtualisation and assembly-level code verification. We showcase SμV by implementing a key security feature: *remote attestation*. Evaluation shows extremely low overhead in terms of memory, performance and battery lifetime for a representative IoT device.

CCS Concepts • **Security and privacy → Embedded systems security**; *Malware and its mitigation*; • **Computer systems organization** → *Sensor networks*;

Keywords IoT, security, memory isolation, remote attestation

ACM Reference Format:
Wilfried Daniels, Danny Hughes, Mahmoud Ammar, Bruno Crispo, Nelson Matthys, and Wouter Joosen. 2017. SμV - the Security MicroVisor: a virtualisation-based security middleware for the Internet of Things. In *Middleware Industry '17: Middleware Industry '17: Proceedings of the Industrial Track of the 18th International Middleware Conference, December 11–15, 2017, Las Vegas, NV, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3154448.3154454

1 Introduction

The Internet of Things (IoT) is moving out of the lab and into the real-world, where it is being applied at large scale in diverse application scenarios such as: factory automation, smart lighting and the city-scale monitoring of human behaviour. Millions of sensors and actuators already connect intimate aspects of our everyday lives and critical industrial infrastructure with the Internet.

Despite the clear security risks, the vast majority of deployed IoT platforms provide extremely limited security primitives, which limits the application of security techniques that are common in mainstream computer systems. This is a critical problem for industrial IoT systems, where security concerns are seen as a major barrier to adoption [9].

This paper addresses the problem of poor security support on contemporary IoT platforms by introducing the concept of a *Security MicroVisor* middleware, which uses selective software virtualisation and assembly-level code verification to isolate a software-based Trusted Computing Module (TCM) from untrusted application software, which may contain a range of IoT security operations. The core contribution of SμV is that it is capable of providing security guarantees that were previously only possible via dedicated hardware. In contrast, SμV works with all standard Micro Controller Units (MCU) that support global interrupt disabling, are single threaded and have sufficient memory to support the preinstalled SμV module. To the best of our knowledge, these features are offered by all MCUs used in contemporary IoT products. By raising the security level of today's devices SμV helps to eliminate a key barrier to adoption of IoT technologies by industry.

We performed a benchmark evaluation of SμV on MicroPnP, a representative industrial IoT platform [18] created by VersaSense. MicroPnP has been deployed in a wide range of industrial scenarios, in over 10 countries. We use SμV to implement a case-study security feature: *remote attestation*. Our evaluation shows that load-time software verification feasible on IoT devices. Furthermore, the runtime overhead of SμV is very reasonable, incurring a decrease in battery life of under 1% in realistic application scenarios and consuming less than 4 KB of flash. Furthermore, the overhead of our exemplar security schemes is promising. Hourly software attestation reduces battery life by a maximum of 6.2%.

The remainder of this paper is structured as follows. Section 2 describes the design rationale and key mechanisms of SμV. Implementation details are discussed in section 3. Section 4 reports on the evaluation of SμV. Section 5 reviews related work. Finally, Section 6 concludes the study.

2 Design of SμV

SμV assumes that an adversary has full access to the network, but cannot physically tamper with the IoT device. The attacker may communicate with the IoT device over the network or prevent legitimate communication from occurring. SμV cannot prevent an attacker from rendering the IoT device unavailable. Furthermore, we assume that the trusted SμV is bug and exploit free and is deployed on the device by a trusted party prior to deployment.

Middleware Industry '17, December 11–15, 2017, Las Vegas, NV, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5200-0/17/12…$15.00
https://doi.org/10.1145/3154448.3154454

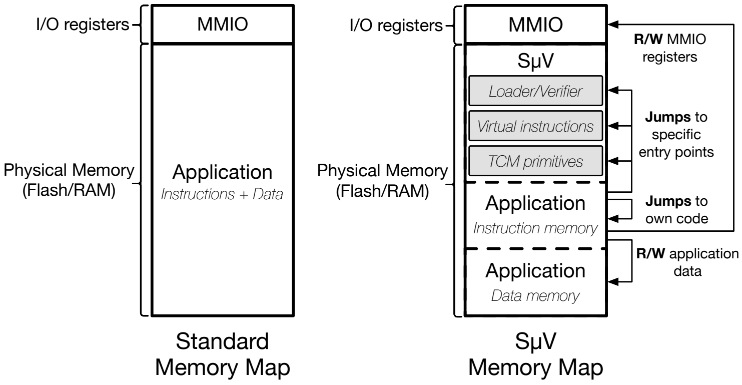


Figure 1. Standard (left) and SμV (right) memory map. In the standard case, memory is monolithic and operations are unrestricted. SμV splits application memory into instruction and data memory and restricts possibly sensitive operations.

2.1 The platform requirements of SμV

MCUs are optimized for low cost and low power operation. Hardware security features and Memory Protection Units (MPUs) are uncommon on these devices and millions of IoT devices are already deployed in critical applications without these features. A piece of malicious software running on such a device can execute arbitrary instructions and read or modify both data and instruction memory, including all resident key material.

SμV is a pure software solution which allows additional security features to be added to conventional MCUs, with guarantees that were previously only possible via dedicated security hardware. Target 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 (under 4 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. To the best of our knowledge this feature is offered by all major families of MCU.

2.2 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*. We visualize this in Figure 1.

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

1. *Control transfer*: branch and jump operations may only address the application instruction memory or the select entry points of the MicroVisor instruction memory which expose virtual operations. This allows for controlled interaction with the MicroVisor.
2. *Data memory access*: read and write operations may only address application data memory or Memory Mapped IO (MMIO) locations.
3. *Instruction memory access*: read and write operations may not address the application instruction memory or the MicroVisor memory.
4. *Deployment* of new applications may only occur through the MicroVisor. This property is enforced by preventing applications by the previous restriction that disallows an application to write in its 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: (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 safe *virtualized instructions*.

2.3 The SμV Toolchain

SμV provides a modified toolchain which allows the application developer to write software for the virtualised SμV architecture with the same ease-of-use as the underlying MCU architecture, and without restriction on high level development tools.

In a standard toolchain, the compiler produces human readable assembly files, which are processed by the assembler resulting in binary object files. Finally, the linker combines all object files together with relevant libraries in a single binary image that can be deployed on the MCU. SμV adds a post-processor which substitutes all unsafe dynamic assembly instructions with calls to their secure virtualized equivalents. Since this is performed when the application is in text ASM form, it can be implemented using simple regular expressions. Secondly, in the linker stage the addresses of the functions residing in the SμV are injected in the form of a symbol table. The SμV is preinstalled on the target MCU, and the application must be linked against these functions at their well-known addresses.

The security properties of SμV are maintained even when an adversary uses their own tool-chain or writes hand-crafted assembly by load-time verification as described in the following section.

2.3.1 Load-time verification

As described above, application deployment may only occur through the MicroVisor, which performs verification. Only *safe* instructions are allowed, which 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 and this results in the application being rejected by the verifier, 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, and in the case of a developer using the SμV toolchain, will have been transparently replaced by calls to safe virtual instructions contained within the protected memory space maintained by SμV.

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 MicroVisor. These virtual instructions check their arguments and will perform the matching operation, only where 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 MCU to reset. Using this approach, the security features of the MCU are enhanced, without sacrificing functionality. Furthermore efficiency is maintained as only inherently unsafe operations are virtualized, allowing all other operations to execute natively.

3 Implementation

A prototype of SμV has been implemented for the MicroPnP IoT platform [18], which offers an IEEE 802.15.4 radio and an 8-bit 10MHz AVR ATmega 1284p [1] MCU, with 16 KB of SRAM and 128 KB of flash. This section gives an overview of how SμV is implemented on the AVR architecture, with adjustments as outlined below.

Modified Harvard architecture: In section 2, we assumed MCUs with the common *von Neumann* architecture, where flash, RAM and any MMIO peripherals are mapped on to a single address space. The AVR family of MCUs use a *modified Harvard* architecture, where the flash memory holding the instructions and the volatile RAM containing the data have an isolated address space with separate instructions to read and modify each memory. Instructions can only be executed from flash, and due to this limitation we can simplify our approach to only protect instruction memory, while data memory operations remain unmodified and unrestricted. Static SμV data is placed alongside SμV code in instruction memory. Applications are not allowed to read the SμV code or data from the instruction memory, and is only permitted to jump within its own code-base or the well known entry points of SμV virtual instructions. The techniques used to restrict these instructions are identical to those used on a *von Neumann* architecture. Leaving the data memory unprotected has the additional advantage that operations on data memory do not incur any runtime overhead.

Bootloader feature: Instruction memory in the AVR is further subdivided into an application and a bootloader section. 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 is configurable before deployment of the IoT device using a physical programmer, but may not be modified at runtime. 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 read and write behaviour of the application still needs to be monitored, as applications should not be able to read SμV instructions or data from the bootloader section. Furthermore, by preventing arbitrary jumps into the bootloader section, SμV prevents the recent *bootjacking* exploit [10] which can be used by applications to claim write self-programming privileges from outside of the bootloader section.

Initialization of data memory: At boot time, variables with an initializer should have their value assigned before code executes. In order to do this, the instruction memory will hold all initial data memory values in addition to the application instructions. Bootstrapping code will copy the initial values from instruction memory to data memory. 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.

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. 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. This list is added as application meta-data at compile-time, and is checked by the SμV for validity at load-time. Applications with an incomplete list of unsafe 2nd words are rejected.

3.1 Remote attestation case study

Remote attestation is a protocol designed to detect and cure malware that is defined by Francillon et al. [6] as follows; a protocol $\mathcal{P}$ is comprised of the following components:

- $\mathsf{Setup}(1^\kappa)$: 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 Prover during commissioning.
- $\mathsf{Attest}(k, n, s)$: A deterministic algorithm used by the Prover that, given a pre-shared key k, a nonce n (provided by the Challenger) and internal state s, outputs an attestation token α.
- $\mathsf{Verify}(k, n, s, \alpha)$: A deterministic algorithm used by the Challenger 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. $\mathsf{Attest}(k, n, s) = \alpha$), and outputs 0 otherwise.

These components are used in the protocol $\mathcal{P}$ between Challenger and Prover as follows: i) both Prover and Challenger possess a pre-shared long-term key k generated by $\mathsf{Setup}(1^{\kappa})$ prior to deployment, ii) Challenger requests proof of the state of Prover and generates a nonce n, iii) Prover runs $\mathsf{Attest}(k, n, s)$ with s its current state, returning the resulting attestation token α to Challenger, iv) Challenger runs $\mathsf{Verify}(k, n, s, \alpha)$, with s the expected state. The output of Verify proves state s of Prover.

Applying SμV to support secure remote attestation Francillon et al. [6] also define a list of minimal properties that are required to support remote attestation, for which they argue that specialized hardware is necessary. We believe that SμV can provide equivalent support in software. We go over the list and explain how 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 ROM 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 placing it in the SμV ROM 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 MCU 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.

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 Challenger 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 Challenger, 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. Our case-study implementation uses HMAC-SHA1 implementation in AVR assembly. HMAC-SHA1 returns a 160 bit keyed hash, and the pre-shared cryptographic key we use is also 160 bits long. It should be noted that, while SHA1 is no longer collision free [7], HMAC, HMAC-SHA1 remains secure and not breakable. On the AVR platform, state can be stored in flash, SRAM, CPU registers and EEPROM. Our implementation we focus specifically on attesting the flash memory, as this is the only location that malware can execute from.

4 Evaluation

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

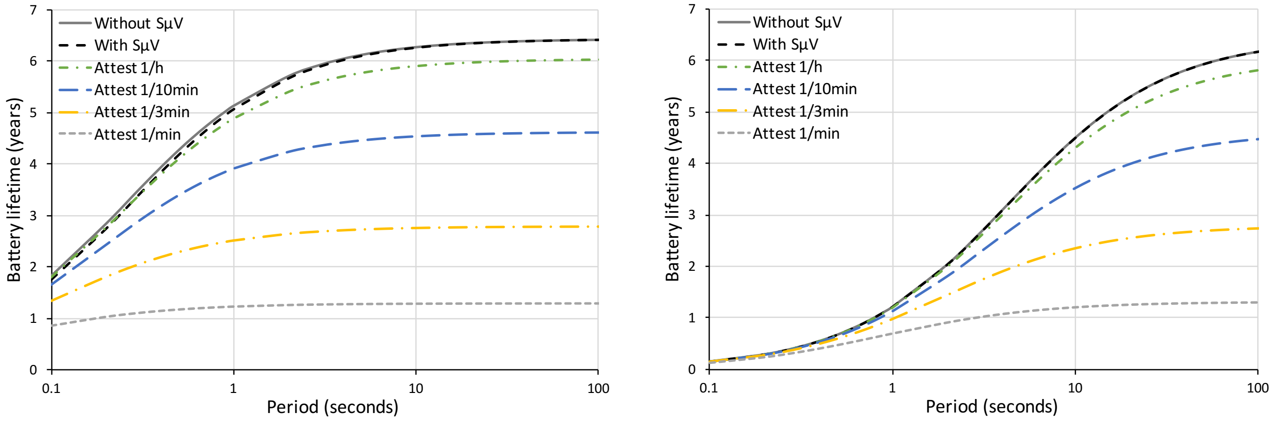
Four reference applications were selected to benchmark performance: (i) a cryptographic application which encrypts and decrypts a random 8 byte cleartext with a 128-bit key, using a software implementation of the lightweight SPECK block cipher [2], (ii) the same crypto application, but implemented in a modular fashion by introducing indirect calls through pointers, (iii) sampling temperature readings from a 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.

Due to space constraints, we provide only average and worst case performance data. A full table of our results for all applications is available online at: https://people.cs.kuleuven.be/~wilfried.daniels/suv.

4.1 Deployment Overhead

During wireless transmission of the application image, the size of the binary image impacts energy consumption and network overhead. During application loading, the time required to load and verify the image determines the total down-time of the wireless device.

Over-the-air binary image size For battery powered, wireless devices, remote software updates have a significant impact on battery life [8], as sending and receiving data over the radio is an energy consuming activity. The size of the application image is linearly correlated to the total energy spent during the software update, as it determines the time spent actively receiving data over the radio. We compared the size of the image for a MCU without SμV, with the size of an image of the same application for a MCU with SμV. The size overhead for the SμV-enabled application images averages 1.61%, with a **worst case increase of 2.17% in application size** in the case of the temperature sensing application. This overhead is caused by two different effects: i) on the AVR, unsafe single word instructions are replaced with a 2 word long instruction



(a) Crypto pointer application

(b) Sense temperature application

Figure 2. 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.

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 Once the application is transmitted, the device will go offline for the duration of the verification and installation of the new application. It is important that this time is minimized in order to maximize device uptime. For a MCU without SμV, verification is a simple check-sum to verify correct transmission. On a SμV-enabled MCU, 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). SμV introduces an average overhead of 4.16%, with a **worst-case overhead of 4.6% in application load times** in the case of the temperature sensing application.

4.2 Runtime Performance:

Execution speed: Execution time is directly correlated with the battery life of low power MCUs. Typically, after the application is done with its tasks, the MCU is 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. On average, execution time overhead amounts to 2.67%. 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. The **worst case overhead occurs in the case of the modular crypto application at 5.66%**.

Battery life: 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 each application against 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. Figure 2 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 when sleeping constantly is 6.5 years. The *Sense temperature* application has worse battery life and due to the busy waiting that is associated with IO intensive tasks, keeping the MCU 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 is a marginal overhead (<1%) 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 inperceptable on the graph whether the application is scheduled every 100ms 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. Figure 2 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%. 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.

Memory overhead: 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 MCU, a minimum amount of **13 bytes of RAM is required for virtual instructions**. Remote attestation inherently needs more stack space to temporary store full pages of flash and maintain intermediate states of the cryptographic functions, In total, **1318 bytes of RAM are required for remote attestation**. This overhead could be reduced by loading memory in smaller chunks, at the expense of performance. The *SμV core* requires 1070 bytes of flash, or a marginal 0.82% of the total flash memory available. *Remote attestation* consumes an additional 1538 bytes including the HMAC-SHA1 hash function. **Total flash used amounts to 2608 bytes or 1.99%** of the total 128 KB flash available on the platform.

5 Related Work

The design of SμV shares similarities with the Software-based Fault Isolation (SFI) approach proposed by Wahbe et al. [17]. 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 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.

The software fault isolation [17] techniques upon which the memory isolation of SμV is built have previously also been applied to low-power MCUs in Harbor [11]. Harbor focuses purely on providing a software-based Memory Protection Unit (MPU) to resource-constrained embedded devices by applying SFI sandboxing techniques, and adds no additional security operations. Harbor provides a hypervisor that is deeply tied to the SOS operating system 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 mote 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 MCUs.

Existing remote attestation approaches can be further classified as hardware or software-based depending on how they enforce the security properties, or elements thereof, that are required by remote attestation. The most relevant are SMART [5] and TyTAN [3]. SMART is a hybrid hardware/software approach that requires hardware modifications to the memory bus access logic of the MCU. SMART isolates and secures remote attestation code by storing it in a secure ROM inside the flash memory. 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 TOCTTOU attacks. If an error is detected, a hardware reset of the MCU is performed enforcing memory cleanup. TyTAN is based on an Execution-aware Memory Protection Unit (EA-MPU), a hardware component that provides memory access control enforcement based on the identity of code that attempts to access a data region. TyTAN adds interruptibility of the attestation process to support hard real-time applications. While SMART and TyTAN have much lower hardware requirements than a TPM, they are still difficult to provide in the lowest-end class of MCUs (e.g., Atmel AVR). More importantly, millions of devices are already deployed that would require hardware modification or replacement to implement these solutions.

Software-only attestation approaches based on timing techniques [12–15] 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 [16] and several concrete attacks have been also published [4]. We therefore do not consider these approaches to be secure.

6 Conclusions and Future Work

This paper introduced SμV, the Security MicroVisor middleware. SμV tackles the problem of limited security features on contemporary IoT devices through a three-fold approach: (i) selective software virtualisation of the MCU 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.

SμV is compatible with the vast majority of IoT MCUs 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. VersaSense is now in the process of rolling out SμV in their MicroPnP platform.

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%. The execution time of application code also increases minimally at 2.67%, and has little effect (<1%) 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%. Our case study shows that SμV can be used to securely implement remote attestation, which was previously believed to be impossible in pure software. Furthermore, hourly software attestation reduces battery life by a maximum of 6.2%, while providing rapid detection of malware.

Our future work will proceed along two fronts (i) focus on developing SμV into a framework which allows a wide range of security features to be easily added using the same techniques and (ii) formal verification of SμV to ensure implementation correctness and thereby guarantee the described security properties.

Acknowledgements

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[1] Atmel. 2009. AVR ATmega 1284p 8-bit microcontroller. http://www.atmel.com/images/doc8059.pdf. (2009). [Online; accessed 13-February-2017].

[2] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2015. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM Press, New York, New York, USA, 1–6. https://doi.org/10.1145/2744769.2747946

[3] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, New York, New York, USA, 6. https://doi.org/10.1145/2744769.2744922

[4] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM Press, New York, New York, USA, 400–409. https://doi.org/10.1145/1653662.1653711

[5] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust.. In *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[6] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. 2014. A Minimalist Approach to Remote Attestation. In *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 3001 Leuven, Belgium, 6.

[7] Google. 2017. Announcing the first SHA1 collision. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html. (2017). [Online; accessed 19-May-2017].

[8] Danny Hughes, Eduardo Canete, Wilfried Daniels, R Gowri Sankar, James Meneghello, Nelson Matthys, Jef Maerien, Sam Michiels, Christophe Huygens, Wouter Joosen, Maarten Wijnants, Wim Lamotte, Erik Hulsmans, Bart Lannoo, and Ingrid Moerman. 2013. Energy aware software evolution for Wireless Sensor Networks. In *IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. IEEE, 1–9. https://doi.org/10.1109/WoWMoM.2013.6583386

[9] Qi Jing, Athanasios V. Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. 2014. Security of the Internet of Things: perspectives and challenges. *Wireless Networks* 20, 8 (01 Nov 2014), 2481–2501. https://doi.org/10.1007/s11276-014-0761-7

[10] Julian Skidmore. 2014. BootJacker: The Amazing AVR Bootloader Hack! http://oneweekwonder.blogspot.be/2014/07/bootjacker-amazing-avr-bootloader-hack.html. (2014). [Online; accessed 10-August-2017].

[11] Ram Kumar, Eddie Kohler, and Mani Srivastava. 2007. Harbor: Software-based Memory Protection For Sensor Nodes. In *International Symposium on Information Processing in Sensor Networks (IPSN)*. IEEE, 340–349. https://doi.org/10.1109/IPSN.2007.4379694

[12] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. 2010. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*. Springer-Verlag, Berlin, Heidelberg, 16–29. https://doi.org/10.1007/978-3-642-13869-0_2

[13] Yanlin Li, Jonathan M McCune, and Adrian Perrig. 2011. VIPER: Verifying the Integrity of Peripherals' Firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM Press, New York, New York, USA, 3–16. https://doi.org/10.1145/2046707.2046711

[14] Arvind Seshadri, Mark Luk, and Adrian Perrig. 2008. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 372–385. https://doi.org/10.1007/978-3-540-69170-9_25

[15] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: : Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review* 39, 5 (oct 2005), 1–16. https://doi.org/10.1145/1095809.1095812

[16] Umesh Shankar, Monica Chew, and J. D. Tygar. 2004. Side Effects Are Not Sufficient to Authenticate Software. In *Proceedings of the 13th USENIX Conference on Security*, Vol. 13. USENIX Association, Berkeley, CA, USA, 89–101.

[17] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review* 27, 5 (dec 1993), 203–216. https://doi.org/10.1145/173668.168635

[18] Fan Yang, Nelson Matthys, Rafael Bachiller, Sam Michiels, Wouter Joosen, and Danny Hughes. 2015. μPnP: Plug and Play Peripherals for the Internet of Things. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. ACM Press, New York, New York, USA, 1–14. https://doi.org/10.1145/2741948.2741980