Soteria: Offline Software Protection within Low-cost Embedded Devices

Johannes Götzfried
FAU Erlangen-Nuremberg
johannes.goetzfried@cs.fau.de

Tilo Müller
FAU Erlangen-Nuremberg
tilo.mueller@cs.fau.de

Ruan de Clercq
KU Leuven and iMinds
ruan.declercq@esat.kuleuven.be

Pieter Maene
KU Leuven and iMinds
pieter.maene@esat.kuleuven.be

Felix Freiling
FAU Erlangen-Nuremberg
felix.freiling@cs.fau.de

Ingrid Verbauwhede
KU Leuven and iMinds
ingrid.verbauwhede@esat.kuleuven.be

ABSTRACT

Protecting the intellectual property of software that is distributed to third-party devices which are not under full control of the software author is difficult to achieve on commodity hardware today. Modern techniques of reverse engineering such as static and dynamic program analysis with system privileges are increasingly powerful, and despite possibilities of encryption, software eventually needs to be processed in clear by the CPU. To anyhow be able to protect software on these devices, a small part of the hardware must be considered trusted. In the past, general purpose trusted computing bases added to desktop computers resulted in costly and rather heavyweight solutions. In contrast, we present Soteria, a lightweight solution for low-cost embedded systems. At its heart, Soteria is a program-counter based memory access control extension for the TI MSP430 microprocessor. Based on our open implementation of Soteria as an openMSP430 extension, and our FPGA-based evaluation, we show that the proposed solution has a minimal performance, size and cost overhead while effectively protecting the confidentiality and integrity of an application's code against all kinds of software attacks including attacks from the system level.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; E.3 [**Data**]: Data Encryption

Keywords

Software Protection, Trusted Computing, Embedded Systems

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2856129

1. INTRODUCTION

Although many software protection techniques such as crypters and packers were pioneered by malware, there always have been *legitimate* interests in hiding the internals of a program, such as the protection of intellectual property of software creators, hardening software against vulnerability detection, and protecting cryptographic keys of communication protocols. The possibilities to reverse engineer software, however, are steadily increasing as witnessed, for example, by the famous IDA Pro disassembler that fully supports full-scale Intel and AMD assembly as well as the instruction sets of embedded devices such as the TI MSP430.

In the past, most practical attempts to thwart reverse engineering were based on *obfuscation*, i.e., transformations making programs harder to analyze [8]. It is well known today that there exists both a class of programs which provably *cannot* be obfuscated [1] as well as a class of programs which *can* provably be obfuscated [31, 5]. Both classes of programs are rather small and cannot be generalized. It becomes possible to achieve perfect obfuscation (in a cryptographic sense) for programs in general only if a scheme for fully homomorphic encryption is available [14]. Since fully homomorphic encryption [15], however, is far from being practical these days it remains a theoretical construction. In summary, there is no practical obfuscation technique being perfect in a cryptographic sense, at least not for general programs. Consequently, for most practical programs it remains unclear whether they can be effectively obfuscated or not.

Hence, software protection remains a problem of large significance today, particularly for embedded devices, because attacks against them have a clear economic motivation. Potential attacks against smart meters for electricity and heat, for example, range from end-user customizers who want to tamper with the amount of consumed energy reported back to their supplier, to competing industrial entities who want to analyze a piece of software for the purpose of re-engineering [6].

As a solution, we present a lightweight software protection scheme that is designed for low-cost embedded devices such as the TI MSP430 for the first time. Our solution is called "Soteria" after the *ancient Greek personification of safety, preservation and deliverance from harm.* Soteria builds upon Sancus [22] which serves as the basis for our work regarding

its zero-software *Trusted Computing Base* (TCB) for attestation and integrity checking.

1.1 Contributions

Our contribution is the design and implementation of a system that provides offline software protection for low-cost embedded systems as an addition to the existing remote attestation and integrity checking capabilities of Sancus [22]. In detail, our contributions are as follows:

- While Sancus currently provides the *integrity* and *authenticity* of software modules, we add the capability to also guarantee *confidentiality*.
- We designed a scheme consisting of *software modules* and *loader modules*. The confidentiality of code and data of the protected software modules is guaranteed at any given point in time against all software attackers, including those who could gain system privileges. The loader modules themselves can be written in software and are not part of the minimal trusted computing base implemented in hardware. For bootstrapping reasons, however, the code of the loader modules is unprotected against reverse engineering.
- Based on hardware supported integrity checks, loader modules can decrypt a protected software module only if the integrity of both is not violated. The key for decryption is derived directly on the target system and hence, is not loaded from a trusted party during runtime. In particular, our solution does not have to attest its trusted execution environment to a remote party but bootstraps autonomously. In other words, Soteria is a solution for *offline* software protection.
- We implemented our approach by patching the openMSP430 core from OpenCores and provide a modified toolchain including the compiler and linker. For example, to get RAM executable and to disallow read access between different modules, we mainly modified its memory access logic. According to our design, Soteria software modules are fully backwards compatible with the original Sancus environment.
- Based on our FPGA-based evaluation, we show that Soteria has practically no runtime overhead but only a small loadtime overhead. We also show that Soteria's costs in terms of size and power consumption are minimal. For example, the power consumption raised by only 0.2% when compared to Sancus.

All parts of our implementation including the hardware design, a working FPGA implementation, the target device software, and the full toolchain are available as open source at `https://www1.cs.fau.de/soteria`.

1.2 Related Work

The drawbacks of software obfuscation have paved the way for hardware-assisted software protection techniques based on a *Trusted Computing Base* (TCB). Basic isolation concepts, like horizontal isolation of the system layer against applications, and vertical isolation of applications against each other, are available in modern operating systems since decades [27]. These basic isolation concepts are supported by hardware extensions like an MMU/MPU or CPU protection rings. To guarantee confidential execution of an application also in the presence of reverse engineers with system level privileges, however, stronger degrees of isolation are required. These degrees of isolation can only be provided by new hardware extensions that have an immutable trust anchor for user applications, such as the *Trusted Platform Module* (TPM) [30] which is deployed, for example, by Flicker [20].

For both x86 CPUs from Intel and AMD as well as embedded systems including the widespread TI MSP430 microprocessor, however, there is currently no dedicated hardware support for software protection. Only recently, Intel officially announced its future x86 extension called *Software Guard Extensions* (SGX) that will provide a general hardware base for software protection on x86 [17, 16, 21]. SGX allows the execution of an application inside a hardware-assisted virtual blackbox, a so-called *enclave* that cannot be manipulated or analyzed from its outside environment including code running in ring zero. SGX has many applications such as secure cloud computing, as shown by a recent project called Haven [2]. On ARM, similar goals as those of SGX are pursued by TrustZone. TrustZone allows only one enclave at a time, called the *secure world*, and is hence hardly useful for the purpose of protecting intellectual property of concurrently running applications. Although TrustZone is available in ARM since years, on mass market products like iOS- or Android-driven smartphones it is, if at all, only used during booting. Currently, it seems to be explored mostly in academic publications [24].

The situation is slightly different for trusted execution on embedded systems without MMU support: Sancus [22, 25, 26] enforces the integrity of software modules with the help of a dedicated program-counter based memory access logic in hardware. Sancus, however, currently only supports integrity checking and remote attestation without software protection which includes the confidentiality of code. Other approaches for remote attestation on embedded devices, such as SMART [11] and a recent proposal by Francillon et al. [13] without a trusted hardware anchor, do not solve the problem of offline software protection either. Last but not least, TrustLite [18] claims to be a powerful alternative to Sancus that is more flexible and efficient and can additionally deal with exceptions. TrustLite, however, is also bound to remote attestation rather than software protection in terms of anti-reverse engineering.

To sum up, there is today still a lack of lightweight software protection for low-cost embedded devices such as the TI MSP430.

1.3 Outline

The remainder of the paper is structured as follows: In Section 2, we give necessary background information about the design of Sancus which serves as a basis for our implementation. In Section 3, we introduce the attacker model and design overview of Soteria with a focus to its security properties from an architectural point of view. In Section 4, we give precise information about our implementation, in particular about its hardware, software and toolchain extensions. In Section 5, we evaluate Soteria regarding its performance, size and power consumption. In Section 6, we give an outlook over limitations and future enhancements. Last, we conclude our work in Section 7.

2. BACKGROUND: SANCUS

Since we build upon Sancus, this section provides a brief overview of Sancus including its design goals and security properties. For more detailed information about Sancus, please refer to its original publication [22].

2.1 General Overview

Sancus is a security architecture for networked embedded devices that supports third-party software extensions. It enables software from different mutually untrusted parties to run on the same device while providing strong guarantees that software remains untampered. This includes support for isolating different software modules, remotely attesting software modules to detect tampered software, and authentication of messages to software providers.

Sancus has a small hardware-only trusted computing base, and uses a minimal amount of hardware features. Its attacker model assumes the attacker to be in complete control of all software, as no software is part of the TCB. The results received from a module can always be validated to see if they are genuine. The system, however, makes no guarantees on availability and confidentiality.

The Sancus project consists of a hardware description of a Sancus-enabled openMSP430 core, as well as a C compiler for Sancus-enabled devices. The compiler supports annotations in the source code that allow developers to easily develop code for this architecture. The full project is provided as open source.

The system model of Sancus is as follows. An Infrastructure Provider (*IP*) owns a set of networked nodes (N_i). Each node consists of a low-end microcontroller, in our case an MSP430, with a single address space for instructions and data. Different Software Providers (SP_j) make use of the infrastructure provided by *IP*. These *SP*s can make software available on the nodes of the infrastructure by compiling their software into a software module ($SM_{j,k}$). An *SM* consists of a binary file which consists of a text and data section, as well as header information that specifies which regions of memory are protected/unprotected. An *SM* can only be loaded onto a node that is part of the infrastructure on behalf of a software provider.

Sancus provides the following security properties: First, software modules are isolated from each other. This ensures that other *SM*s cannot read or modify each other's state. Second, the hardware TCB performs remote attestation, allowing the system to make strong guarantees on the integrity and authenticity about the running software on the system. Third, a software provider can verify any software module loaded on *IP*. Fourth, tamperproof communication is provided between modules with integrity and authenticity guarantees. Fifth, software can securely link to each other, meaning that software modules can call each other with the assurance that the intended module is being called.

2.2 Sancus Module Layout

As was mentioned before, Sancus isolates software modules using program-counter based memory access control. An overview of the resulting memory layout is shown in Figure 1. Two types of memory can be distinguished: There is *protected memory* assigned to a specific module, and everything else referred to as *unprotected memory*. The protected memory region is divided into two sections, one for code and constants and one for protected data. The boundaries of these regions are stored in dedicated registers which are added to the processor architecture. These registers are used as inputs to the memory access logic which compares them to the current program-counter to enforce the access rights.

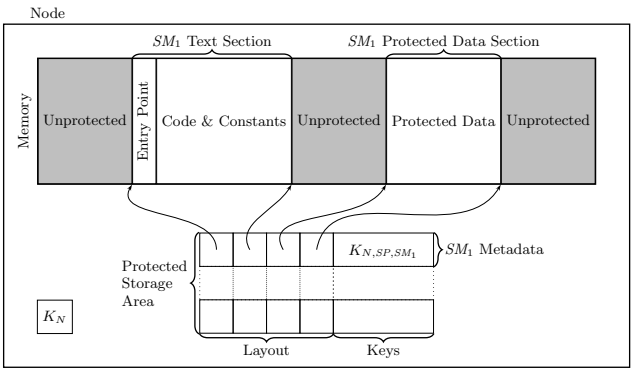

Figure 1: The layout of a Sancus software module in memory [22].

The code in the text section can *only be executed* when the program-counter is either at the entry point or the module is already executing. Note, however, that it is possible to *read the code* from anywhere. Contrary to that, the module's data can only be read or written when the program-counter is in the module's text section.

Sancus adds two custom instructions which can be used by application developers to isolate their application, namely `protect` and `unprotect`. When `protect` is called, the implications are threefold: The memory boundaries are checked for overlap with existing ones and are written to their respective dedicated registers, enabling access control. The last step is deriving the module's key and storing it in its dedicated register. Executing `unprotect` clears the boundary registries, thereby deactivating the memory protection.

2.3 Sancus Key Derivation

Sancus introduces three types of keys to the system. First, the node key K_N is bound to the hardware and only known by the *IP*. $K_{N,SP}$ is used by the *SP* to deploy modules to a specific node. It is derived from K_N together with the provider's unique public ID using a key derivation function *kdf* as follows:

$$K_{N,SP} = kdf(K_N, SP)$$

Third, $K_{N,SP,SM}$ is shared between the *SP* and *SM*. It is obtained from $K_{N,SP}$ and the identity of *SM* as follows:

$$K_{N,SP,SM} = kdf(K_{N,SP}, SM)$$

Figure 1 shows that this key is stored in the protected storage area alongside the memory boundaries.

3. SOTERIA ARCHITECTURE

In this section, the architecture of Soteria is explained in detail. Our contribution is maintaining the confidentiality of code and data in a low-cost embedded system without the need to trust any software component. First, in Section 3.1, we introduce our attacker model which serves as a basis for Soteria. Second, in Section 3.2, we present the architecture design of Soteria, including the decryption key derivation.

In Section 3.3, we describe how Soteria could be deployed in practice. Section 3.4 explains how confidentiality can be maintained after decryption. Finally, in Section 3.5, we give a high level explanation of the security properties that Soteria is able to guarantee.

3.1 Attacker Model

In our model, we assume that an attacker wishes to violate confidentiality of the code and data of an arbitrary *SM*. To achieve this, an attacker can mount all kinds of *software* attacks but *no hardware* attacks. To be more precise, an attacker is allowed to control all peripheral components, including tampering with the communication to other devices. Furthermore, each piece of software, also privileged software like an operating system is allowed to be under the control of attackers, because access to *SM*s is solely prevented by hardware.

However, hardware attacks are excluded from our attacker model, in particular attacks like RAM dumping, chip probing and fault injection are excluded. If hardware attacks are considered at all, only ROM dumping is allowed as it does not harm our solution. Also note that, as the attacker is able to control privileged software, he or she can easily restrict availability of the overall system. Denial of service (DoS) attacks are excluded from our attacker model as they have no impact on the confidentiality of code and data.

3.2 Architecture Design

For the Sancus-based design of Soteria we thought of two possible concepts: (1) Putting a loader stub together with the code that should be encrypted into one module and decrypting it in-place, and (2) designing a separate loader module. In the first case, the loader stub has to reside within RAM in addition to or instead of ROM and at least parts of the loader code would be duplicated for each encrypted module wasting space. Therefore, we decided to use the second concept, i.e., to design a separate loader module. With this concept, a new loader software module SM_L provided by SP_L is responsible for decrypting and protecting another module SM_E provided by SP_E. The decryption key for SM_E is derived from the loader key K_{N,SP_L,SM_L} and the unique identifier $\widetilde{SM_E}$ of SM_E which consists of the name and the current version of SM_E. The decryption and protection of SM_E within SM_L must run atomically to guarantee the confidentiality of code and data of SM_E at any point in time.

Soteria maintains the confidentiality of code and data based on a zero-software TCB with two different mechanisms. First, before loading an encrypted module, the code resides encrypted within ROM or RAM such that no other module is able to read it. Second, after an encrypted module has been loaded, a program-counter based memory access logic ensures that no other module can access code or data of the decrypted module. The remainder of this section gives an overview of how loading encrypted modules works.

In Figure 2 and Figure 3, the loading process of a module within Soteria is illustrated: At first, only the loader is protected and active. Then, SM_L derives E_{SM_E}. Next, SM_E is checked for integrity, gets decrypted and protected, and finally SM_L is able to unprotect itself. The detailed loading process is as follows:

1. SM_L is loaded like an ordinary Sancus module and typically has a code section within ROM and a data section within RAM.

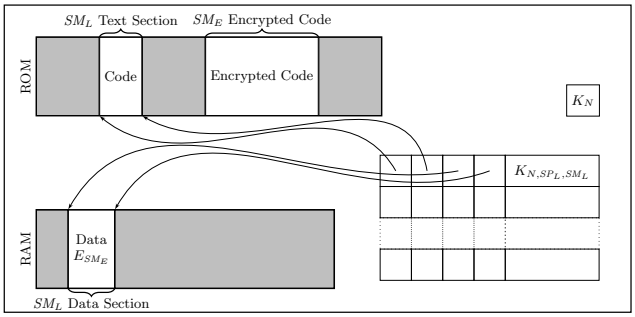

Figure 2: Loading Steps 1+2: SM_L is active, protected and has derived E_{SM_E}.

2. If SM_L is instructed to load another encrypted module, it first derives the decryption key E_{SM_E} from its own module key K_{N,SP_L,SM_L} and the unique identifier $\widetilde{SM_E}$ of the encrypted module SM_E it is about to decrypt:

$$E_{SM_E} := K_{N,SP_L,SM_L,SM_E} = kdf\Big(K_{N,SP_L,SM_L}, \widetilde{SM_E}\Big)$$

3. The module SM_E is decrypted and checked for integrity simultaneously by using authenticated decryption with the key E_{SM_E}. If the integrity property is violated, all intermediate data is wiped and the loading process is aborted.

4. The module SM_E gets protected and implicitly derives the key K_{N,SP_E,SM_E}.

5. The loading process is finished and SM_L is now able to load the next encrypted module or to unprotect itself.

Note that steps 3 and 4 need to be performed atomically by SM_L. Hence, it has to be ensured that no other module runs between these three steps, e.g., by disabling interrupts globally. All other steps do not need to be performed atomically as E_{SM_E} is securely stored within the protected data section of SM_L and therefore cannot be read out by any other module even if it runs while E_{SM_E} has already been derived. For encryption and integrity checking, we use AES-128 [9] in CCM mode of operation [12] which provides authenticated encryption, i.e., encryption and integrity checking is done at the same time with the same key. We chose CCM over other authenticated encryption modes such as GCM [10] or OCB [23] because of its simplicity and the fact that CCM is not patent encumbered. Another advantage of CCM is that only the encryption routine of AES-128 is needed which is more efficient in terms of runtime compared to the decryption routine. For the detailed CCM parameters see Section 4.2.

Although the code for the loader and the encrypted code for the module usually reside within ROM, it is possible to place it in RAM as well. In any case, it is cryptographically guaranteed that no tampered module is loaded, and that the valid module is never decrypted by a tampered loader. For more details, see Section 3.5.

3.3 System Deployment

Sancus specifies an Infrastructure Provider (*IP*), Software Provider (*SP*) and Software Modules (*SM*s). In practice, a software provider SP_j receives K_{N_i,SP_j} once from the *IP* for

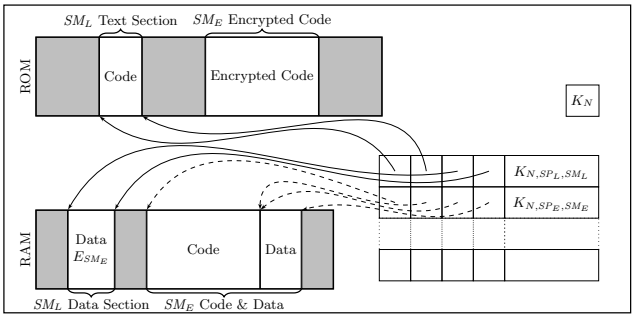

Figure 3: Situation after Step 4: SM_E is decrypted, protected and has derived K_{N,SP_E,SM_E}.

each node N_i that should run software of SP_j. For each $SM_{j,k}$, the software provider can then derive $K_{N_i,SP_j,SM_{j,k}}$ on its own, which can without the knowledge of K_{N_i,SP_j} only be derived from node N_i in hardware. Thus, for each $SM_{j,k}$ a secret shared key $K_{N_i,SP_j,SM_{j,k}}$ between N_i and SP_j exists which is used to prove its integrity and authenticity.

To the above scenario, Soteria adds SP_L, the software provider of the loader, and SM_L, the loader module which is responsible for loading encrypted modules $SM_{E,1}, \ldots, SM_{E,n}$, as shown in Figure 2. For the deployment of our system there are basically two possibilities: (1) $SP_L = SP_E$, i.e., the software provider of the encrypted modules is also the software provider of the loader module, and (2) $SP_L \neq SP_E$, i.e., the software provider of the encrypted modules is different from the software provider of the loader.

Encrypting modules, i.e., preparing the ROM image for a target device or preparing software updates can be done on the host of SP_E with knowledge of a previously exchanged encryption key, but has to be done differently in both cases. The key exchange is trivial for (1), because the software provider has direct access to the key K_{N,SP_L,SM_L} and can therefore derive $E_{SM_{E,1}}, \ldots, E_{SM_{E,n}}$ (see Section 3.2) and encrypt the modules $SM_{E,1}, \ldots, SM_{E,n}$. For case (2), however, SP_L and SP_E have to cooperate for the key management, meaning that SP_E has to get $E_{SM_{E,k}}$ for each module $SM_{E,k}$ by telling SP_L the unique identifier $\widetilde{SM_{E,k}}$ of $SM_{E,k}$. Although SP_E does not need to give code to SP_L but only the unique identifier of the module, SP_E needs to trust SP_L. In particular, SP_E needs to rely on the fact that SP_L only provides a loader for the module on a specific target device but keeps the derived decryption key secret.

Similar to *IP* being the trusted party for remote attestation, SP_L is the trusted party regarding confidentiality. If only confidentiality is needed but no attestation is required, it is sufficient for SP_E to communicate with SP_L if $SP_L \neq SP_E$. Note that *IP*, which is the root of trust in the overall system, is of course able to violate confidentiality at any time. This is not a drawback to our proposed system, as the hardware is considered trusted and all parties need to trust *IP* as in Sancus.

3.4 Access Control

Assuming an encrypted module SM_E has been loaded correctly, its code now resides within RAM and needs to be protected by the program-counter based memory access logic. In theory, reading the text section only has to be disabled for modules that were encrypted, as the others are stored in cleartext within ROM, but we decided to disable read access from anywhere for all modules.

From/To	Entry	Text	Protected	Unprotected
Entry	`r-x`	`r-x`	`rw-`	`rwx`
Text	`r-x`	`r-x`	`rw-`	`rwx`
Unprotected/ Other SM	`r-x`	`---`	`---`	`rwx`

Table 1: Access rights from/to a module with prohibited read access to code [22].

Table 1 shows the different access rights for the whole address space. The only part of a module that remains readable and executable is the entry point. All other areas of a module, i.e., the text section and the data or protected section, are not read- or writeable from any other module or unprotected code, which is ensured by adding new checks to the memory access logic of Sancus. There is a new kind of memory violation that is triggered if another module or unprotected code tries to read the text section of a module. All violation signals are combined into a single signal that resets the processor immediately when triggered. Although it would be possible to return NULL on read violations and to ignore write violations, there is no other reasonable choice for handling execute violations than triggering a reset because system level attacks are included in our attacker model and thus no appropriate handler routine can be called. Furthermore, we cannot fully guarantee availability due to attacks from the system level such that not triggering a reset would not improve availability.

In addition to an extension of the memory access logic, we take care that RAM is wiped completely in hardware after a reset has been triggered. Consequently, no fragments of code or data of a previously encrypted module can be found after a reset. This is particularly useful if a violation is triggered, because a malicious module could otherwise trigger a reset intentionally, and, when the module is reloaded after the reset, read out portions of RAM which still contains cleartext of encrypted modules.

3.5 Security Properties

After explaining the architecture of Soteria in the previous sections, we briefly describe its security properties now. In detail, Soteria can guarantee the following security properties:

- *Isolation*: Every module is completely isolated from other modules regardless of their privilege level. No other module or unprotected code can read or write from or to the code and data section of a given module. This property is partly inherited from Sancus, but was extended by the capability to reject read-access to the code section of a module.

- *Remote Attestation*: A remote party can cryptographically verify that a specific module has been loaded on a given device with a previously defined state. Based on remote attestation, tamperproof communication is possible as well, because messages can be combined with authenticity and integrity information before they are sent. This property is most widely inherited from Sancus, but the host tools for HMAC generation had to be modified slightly.

- *Secure Linking*: A software module on a given node can call a function of another module on the same node while guaranteeing that it is calling the intended function of the intended destination module. This property is inherited from Sanucs.

- *Confidentiality*: Confidentiality of code and data for encrypted modules can be guaranteed at any given point in time, i.e., before modules are loaded as well as afterwards. Furthermore, confidentiality can be ensured *offline* due to mutual integrity checks between the loader and the module that is about to be decrypted, without the need for remote attestation. This property is not present in Sancus but a novel contribution of Soteria.

- *Integrity*: While for Sancus the software provider is able to ensure the authenticity and integrity of a module through remote attestation only after communicating with it, Soteria is able to guarantee the integrity of encrypted modules *offline*, meaning that manipulations are already detected at load time rather than communication time. This property is a novel contribution of Soteria.

The confidentiality property is guaranteed by two mechanisms. Before load time, modules are encrypted and thus considered confidential. After load time, modules are protected by the program-counter based memory access logic and are therefore considered confidential as well. The only possibility of attackers to violate the confidentiality is by compromising the loading process, which is prevented by our design as follows: If the loading position or protected sections of the loader are tampered with when the module SM_E is about to be loaded, the key E_{SM_E} is derived incorrectly because K_{N,SP_L,SM_L} is derived wrongly. The authenticated decryption then fails and the loading process immediately aborts. If the encrypted module SM_E is tampered with before loading, the authenticated decryption fails as well and loading aborts again. Tampering is not possible while the loading takes place because authenticated decryption and protecting SM_E is performed atomically.

In summary, confidentiality is always preserved and any attempts to tamper with encrypted modules are detected immediately. The only property an attacker is able to affect is availability, but as explained in Section 3.1, we exclude denial of service from our attacker model. Our solution provides no DoS protection and DoS is out of scope for this paper. For example, an attacker could repeatedly trigger resets or monopolize the CPU with an endless loop to impact availability.

4. IMPLEMENTATION

In this section we describe the implementation details of Soteria. First, in Section 4.1, we describe the changes we made to the hardware in order to guarantee the security properties mentioned in Section 3.5. Second, in Section 4.2, we describe the software we developed for the target device, i.e., for the modified openMSP430. Finally, in Section 4.3, the toolchain that is necessary to deploy our system is explained. All software belonging to the toolchain runs on the host, meaning that it does not increase the trusted computing base.

4.1 Hardware

Our implementation is based on the openMSP430, an open-source implementation of the MSP430 from Texas Instruments [29] in Verilog provided by OpenCores. Sancus is built on the openMSP430 and was the starting point of our implementation. The MSP430 is a 16-bit processor with a von-Neumann architecture. It has support for 8-bit and 16-bit peripheral components, such a timer, UART and GPIO. During our development, we simulated the modified openMSP430 using Icarus Verilog directly on the host system. For the evaluations, we ported our implementation to an FPGA and used the TimerA component, UART and GPIO. For more information on the evaluation, see Section 5.

First of all, we introduced a new violation that is triggered if a module tries to read the code section of any other module. The address of the memory access is checked against the boundaries of the code section of every protected module and is only allowed if the program-counter is within the code section of the module currently being executed. Otherwise a violation is triggered, which is OR'ed with the other memory access violations.

While implementing our concept, we came across a fundamental limitation of the openMSP430. Although it has a von-Neumann architecture, it distinguishes between ROM and RAM by mapping them to specific locations within the 16-bit address space. As a result, it turned out that RAM was not executable, which is also mentioned as a drawback on OpenCores. To support our proposed design, we need to decrypt a module to RAM, protect and execute it. We therefore needed to patch the openMSP430 to make RAM executable. Our current implementation of the openMSP430 has executable RAM and the ability to switch execution freely between code from any region, so that unprotected code that resides, for example, within ROM, can directly jump to previously encrypted modules which reside within RAM after decryption.

To prevent the exploitation of deliberately triggered resets (see Section 3.4), we developed a memory wiping component directly in hardware. The component is inserted between the openMSP430 itself and the RAM. When a reset is triggered, it starts wiping the RAM word by word for all 16-bit words in memory. While this takes place, the memory wiping component holds the reset line, simulating to the openMSP430 that the reset is still not released. The openMSP430 therefore remains paused while the entire memory is being zeroed out from start to end. After wiping is finished, the reset line is released and the openMSP430 dispatches the first instruction which is referenced by the reset vector. This procedure ensures that RAM is completely wiped before the very first instruction is dispatched after a reset.

4.2 Software

In addition to hardware modifications, we have written software that runs on the target device, namely the openMSP430 processor. As we want to protect third-party software, we needed to provide two components: (1) a library which supports encrypted modules while being fully compatible with non-encrypted legacy modules of Sancus, and (2) the loader module itself, which is capable of integrity checking, decrypting and protecting an encrypted module.

In Sancus, modules are represented by a structure consisting of a unique ID, the software provider ID, the module name and the boundaries for the code and data section. This

structure contains all values necessary for traditional modules to be loaded, and the boundaries are those that are considered by the memory access logic at module runtime. For Soteria, we extended the structure with new boundaries for the code section of the encrypted module, i.e., the boundaries are valid only for the encrypted code but not for the code section of a running module. In a typical scenario, the boundaries for the encrypted code point to ROM, while all others, i.e., those for the code and the data section, point to RAM. The authentication tag for encrypted modules is stored directly after the encrypted code section such that the upper boundary for the encrypted code section points directly to the integrity information. Our new library is fully compatible with existing non-encrypted legacy modules. On the one hand, new encrypted modules can of course be used in conjunction with our loader, but on the other hand, they can also be directly passed to existing functions of the old Sancus library. Our library also includes a function to destroy a module. When it is called, all the module's protected data and code within RAM is first wiped and then the `unprotect` function of the Sancus library is invoked. It is basically a convenience routine to allow the programmer to destroy a module with only a single call.

We also provide an open implementation of the loader for the openMSP430 such that it can be used by every software provider SP_L. The loader accepts encrypted modules and basically provides two functions to the user: The load routine which takes an encrypted module, derives the decryption key for the destination module, performs integrity checking, decrypts the module to RAM and finally protects the just decrypted module. As already explained, these steps have to be performed atomically. The second routine, that is provided by the loader is a routine which causes the loader to wipe its protected data section and afterwards unprotect itself. This routine is used to destroy the loader.

The key derivation is done in hardware, as explained in Section 3.2, using the HMAC functionality provided by Sancus which is based on *Spongent-128* [4]. To this end, the unique identifier $\widetilde{SM_E}$ which is a concatenation of the module name and the module version number is passed to the HMAC function which implicitly uses K_{N,SP_L,SM_L} as key.

The decryption is implemented in software using AES-128 [9] in CCM mode of operation [12]. We implemented the CCM mode according to RFC 3610 [32] with an authentication tag length of sixteen bytes, a two byte length field and no associated data, i.e., data that just needs to be authenticated. With this choice we are able to decrypt software modules with a maximum length of 64 kilobytes which is also the maximum addressable size on the openMSP430 processor. In addition to the decryption key and the encrypted module, CCM with the given parameters also requires a thirteen byte nonce which does not need to be secret but must be different for each CCM en- or decryption with the same key. We simply use the unique identifier $\widetilde{SM_E}$ (padded with zeros) as nonce because it is changed for every new version of the module SM_E and thus satisfies the nonce properties. If the module name concatenated with the module version number exceeds thirteen bytes, a cryptographic hash of $\widetilde{SM_E}$ is calculated and truncated to thirteen bytes. The calculation of the hash then only needs to be done by the toolchain and not on the target device because the nonce can securely be stored along with the authentication tag.

We built our implementation onto the *tinyAES* implementation combined with the CCM implementation of mbed TLS. Consequently, our authenticated decryption routine is implemented entirely in C and it has been optimized for size. The overall size of the key schedule, the encryption routine and the CCM implementation is about two kilobytes. Memory protection is enabled by calling the special instruction `protect` as described in Section 2.2.

4.3 Toolchain

Aside from hardware and target device software, we had to implement toolchain components in order to successfully deploy our system. The whole toolchain is based on LLVM [19], msp430-gcc [28] and pyelftools [3]. The toolchain consists of three Python tools that need to be called after successfully compiling different source files with LLVM:

1. `ld`: The linker which produces a single ELF file from the different object files and generates separate sections for each software module. Two sections are generated for encrypted software modules. One is placed in the RAM address range and contains all cleartext code like it would be after the decryption during the loading process. The other is placed in the ROM address range and is 16 bytes larger than the section within RAM to reserve space for the integrity information that is produced by the CCM authenticated encryption. This second section is just filled with zeros by the linker and will later be overwritten by our encryption script.

2. `hmac`: This script, which was inherited from Sancus and only needed to be modified slightly, adds HMACs to the ELF file for secure linking.

3. `crypt`: This script performs authenticated encryption of the cleartext code and places the result within the section in the ROM address range. It implicitly computes integrity information in CCM mode and stores it directly after the encrypted code. To this end, the script has to derive the encryption key from the key of the loader and the unique identifier of the encrypted module. The derivation is fully automated and only the vendor ID, node key and name of the loader module need to be supplied as arguments. After the code has been encrypted, the original section within RAM is stripped from the ELF file so that the resulting image can be flashed to the target device or distributed securely, as it no longer contains any unencrypted data.

All described scripts work directly on ELF files to avoid using different intermediate formats. The toolchain is easy to use because very few information needs to be provided. Modules that need to be encrypted are identified by their name: If a module starts with `crypt_`, the scripts transparently add new sections for this module, encrypt it, add integrity information and reserve space in RAM to store the plaintext at runtime. Of course, all components of the toolchain run on the host instead of the target device, are therefore not part of the device trusted computing base, and consequently do not consume space on the target device.

5. EVALUATION

In this section, we evaluate Soteria regarding its performance, impact on chip size and power consumption. All data

in this section was produced using a Xilinx XC6VLX240T Virtex-6 FPGA, with the core running at 20 MHz.

5.1 Performance

When evaluating the performance of Soteria, we have to distinguish between the runtime overhead and the time a module needs to be loaded. First of all, there is a constant overhead for resetting the processor as all data memory is wiped in hardware before execution resumes. Since the MSP430 has no caches and memory can be written directly, wiping takes exactly 2+DMEM_SIZE/2 cycles, because it is wiped by writing successive 16-bit words which corresponds to two bytes. Our reference configuration, for example, hase a program memory of 48 kilobytes and a data memory of 10 kilobytes, so 5,122 cycles are needed for wiping.

Once an application is running, our solution imposes no additional overhead compared to plain Sancus, meaning that the additional program-counter based memory access checks do not have a performance impact on the critical path. During our experiments, we also verified, using the TimerA component of the MSP430, that routines of modules built with our solution execute in the same amount of cycles as those built with plain Sancus. This applies to unencrypted legacy modules as well as to encrypted modules after they have been loaded.

The main performance overhead of our solution is incurred by loading of an encrypted module. Protecting the loader imposes a constant overhead on all encrypted modules combined. This process takes 72,976 cycles, while destroying the loader (i.e., freeing all resources, wiping keys and calling `unprotect`) needs only 800 cycles. The loading overhead for an encrypted module depends on the size of the module and is dominated by the time needed for the authenticated decryption routine. In Table 2, the number of cycles needed to load modules with different sizes are shown. First, the key for authenticated decryption of the encrypted module is derived. This is done in hardware and the number of required cycles is therefore rather small. Furthermore, the unique identifier had the same length for all modules tested in Table 2 and thus the number of cycles needed for the key derivation is independent from the size of the encrypted module. The module then has to be decrypted and checked for integrity, which takes up most of the cycles because authenticated decryption is done in software using AES-128 in CCM mode of operation. Finally, the decrypted module needs to be protected to deny other modules access to the decrypted code. The protection process is dominated by the key derivation for the new module and therefore the execution time again depends on the size of the module that is about to be protected. Note that the total number of cycles to load an encrypted module is more than just the sum of key derivation, authenticated decryption and protection, because of additional boundary checks and a length calculation of the unique identifier. These additional computations, however, only depend on the unique identifier of the encrypted module and as it was identical for all of our tests, the additional overhead is constant.

The smallest module that was evaluated had a size of only 208 bytes. This is the smallest size possible for a module with at least one entry point and no additional data. The minimum size results from the stubs that are included by the Sancus compiler to be able to call the entry point. Other module sizes that have been tested range from 256 bytes up to one kilobyte. Considering the clock rate of 20 MHz, we need about 92 milliseconds to load an encrypted module of one kilobyte and 25 milliseconds for an encrypted module of 256 bytes size. While this might be some overhead, the primary reason for which is running AES in software, one has to consider that modules are only loaded when the system is started. Since no runtime overhead is imposed on the modules, short loading times below 100 milliseconds seem acceptable. Furthermore, minimum code size instead of maximum loading performance was the main focus. The implementation of AES-128 in CCM mode of operation only needs about 2 kilobytes of ROM and circa 200 bytes of RAM. Last but not least, we are planning to extend Soteria with hardware support for ciphers, as explained in Section 6.

To be able to produce these performance evaluations, a special version of Soteria was synthesized which does not trigger a reset if a memory violation occurs because interrupts, for example, can trigger an execute violation. The TimerA component of the MSP430 was used with the main system clock of 20 MHz in divide-by-eight mode. As this component only supports a 16-bit counter, it overflows for large measurements and an additional counter for the higher 16 bits needed to be implemented in software. This counter is incremented in the interrupt handler called for the overflow. Although only a single operation, i.e., the increment of the counter for the higher 16 bits, is executed within the interrupt handler, measurements which take more than 65536 cycles are not perfectly cycle accurate, but instead a higher number of cycles is measured than what is actually needed for the pure computation. This is due to the fact that entering and leaving the interrupt handler takes cycles, and those cycles are included in our measurements. Consequently, we give upper bounds for the actual performance results which are expected to be even less than those shown in Table 2.

	Sancus		Soteria		Overhead	
	REGs	LUTs	REGs	LUTs	REGs	LUTs
1 SM	1,897	3,686	1,938	3,894	41	208
2 SMs	2,110	4,100	2,150	4,322	40	222
3 SMs	2,323	4,378	2,363	4,620	40	242
4 SMs	2,536	4,778	2,576	5,034	40	256

Table 3: Number of slice registers and slice LUTs for Soteria compared to Sancus.

5.2 Area

We measured the area overhead of Soteria by considering the required slice registers and slice LUTs. The plain openMSP430 core needs 1,146 slice registers and 2,520 LUTs when synthesised for our FPGA. In Table 3, the number of slice registers and slice LUTs for Soteria in comparison to plain Sancus are shown in different configurations, i.e., for a specific number of supported modules. The slice register overhead Soteria incurs over Sancus is almost constant and equal to about 40. The number of additional slice LUTs depends on how many modules are supported, but it is small compared to the overall number of slice LUTs. The main overhead comes from the memory wiping logic and the additional read access prevention which is added for each module. Since the decryption routines are implemented in software rather than hardware, our solution imposes very little area overhead.

Size	Key Derivation	Authenticated Decryption	Protection	Total
208	13,504 (0.675)	383,344 (19.167)	26,984 (1.349)	424,312 (21.216)
256	13,504 (0.675)	463,088 (23.154)	30,464 (1.523)	507,536 (25.377)
512	13,504 (0.675)	888,456 (44.423)	49,024 (2.451)	951,464 (47.573)
768	13,504 (0.675)	1,313,816 (65.691)	67,584 (3.379)	1,395,384 (69.769)
1024	13,504 (0.675)	1,739,176 (86.959)	86,144 (4.307)	1,839,304 (91.965)

Table 2: Number of cycles (ms) needed for loading modules with different sizes.

5.3 Power

We have analyzed the power consumption with the static power analysis tool *Xilinx XPower Analyzer*. Soteria had an overhead of about 0.2% compared to Sancus, regardless of the number of supported modules. We also tried to measure the power consumption experimentally, but could not find any difference between Soteria and Sancus running on our FPGA. The overall power consumption of the FPGA reported by *Xilinx XPower Analyzer* was about 3.537W for Soteria and about 3.530W for Sancus.

6. LIMITATIONS AND FUTURE WORK

Our goal is to protect the intellectual property of code and data against powerful software attackers, including those who could gain system privileges. Our attacker model, however, excludes all kind of hardware attacks such that attackers with physical access are potentially able to acquire sensitive data while circumventing the protection of Soteria, e.g., by chip probing or fault injection. Protection against physical access of code and data is therefore part of future work.

Although we designed and implemented Soteria as a fully working software protection solution, there still remain open questions and future work. The program-counter based memory access logic prevents all jumps into module code different to the entry point from other modules or unprotected code. Thus, we currently cannot handle interrupts because the access logic would prevent the execution flow to return from an interrupt routine. There exists an approach that does leverage Sancus for handling interrupts [7], which could be combined with Soteria in future.

The primary performance bottleneck of Soteria is the decryption routine which is implemented in pure software. To speed up the loading process of encrypted modules, the decryption routine must be implemented in hardware and provided to the programmer as separate new instruction, analogously to the instructions for protecting and unprotecting modules. AES, however, is pretty heavyweight for embedded systems and would increase the area drastically such that a more lightweight cipher optimized for hardware implementations must be considered.

7. CONCLUSIONS

In this paper, we presented a lightweight software protection solution with a zero-software trusted computing base. Only the hardware, i.e., a modified variant of the openMSP430, needs to be considered trusted to be able to guarantee confidentiality and integrity of code and data. To the best of our knowledge, this is a novel design and the confidentiality of code against reverse-engineering has not been provided by means of a program-counter based memory access logic before.

Based on software protection, all kind of intellectual property for digital contents can be protected. Our solution is therefore adaptable to new protection scenarios beyond software protection. We are aware of the dark side of our proposed technology ranging from divisive applications such as DRM to the threat of unanalyzable malware and backdoors. Nevertheless, software protection has many legitimate use cases that must be solved such as the protection of intellectual property, hardening software against vulnerability detection, and protecting cryptographic keys, to name but a few.

Acknowledgements

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89).

Availability

Soteria is free software. All parts of our implementation including the hardware design, the FPGA implementation, the target device software, and the toolchain are available as open source at `https://www1.cs.fau.de/soteria`.

8. REFERENCES

[1] Barak, B., Impagliazzo, O. G. R., Rudich, S., Sahai, A., Vadhan, S., and Yan, K. On the (Im)Possibility of Obfuscating Programs. In *CRYPTO (Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology)* (Santa Barbara, California, USA, Aug. 2001), D. B. Joe Kilian, Ed., Springer-Verlag, pp. 1–18.

[2] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014), USENIX Association, pp. 267–283.

[3] Bendersky, E. pyelftools – Library for analyzing ELF files and DWARF debugging information, Feb. 2015.

[4] Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varici, K., and Verbauwhede, I. Spongent: A lightweight hash function. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds., vol. 6917 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 312–325.

[5] Canetti, R., and Dakdouk, R. R. Obfuscating Point Functions with Multibit Output. In *EUROCRYPT '08 (Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology)* (Apr. 2008), Springer-Verlag, Berlin, pp. 489–508.

[6] Chikofsky, E. J., and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw. 7*, 1 (Jan. 1990), 13–17.

[7] Clercq, R. D., Schellekens, D., Piessens, F., and Verbauwhede, I. Secure Interrupts on Low-End Microcontrollers. In *25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014)* (2014), IEEE, pp. 1–6.

[8] Collberg, C., and Nagra, J. *Surreptitious Software: Obfuscation, Watermarking and Tamperproofing for Software Protection*. Addison-Wesley Longman, Amsterdam, July 2009.

[9] Daemen, J., and Rijmen, V. The Block Cipher Rijndael. In *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds., vol. 1820 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 277–284.

[10] Dworkin, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication, May 2012.

[11] Eldefrawy, K., Francillon, A., Perito, D., and Tsudik, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium* (San Diego, United States, Feb. 2012).

[12] Fouque, P.-A., Martinet, G., Valette, F., and Zimmer, S. On the Security of the CCM Encryption Mode and of a Slight Variant. In *Applied Cryptography and Network Security*, vol. 5037 of *Lecture Notes in Computer Science*. 2008, pp. 411–428.

[13] Francillon, A., Nguyen, Q., Rasmussen, K. B., and Tsudik, G. A minimalist approach to remote attestation. In *Proceedings of the Conference on Design, Automation & Test in Europe* (3001 Leuven, Belgium, Belgium, 2014), DATE '14, European Design and Automation Association, pp. 244:1–244:6.

[14] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., and Water, B. Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits. In *IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)* (Berkeley, CA, Oct. 2013), pp. 40–49.

[15] Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2009), STOC '09, ACM, pp. 169–178.

[16] Hoekstra, M., Lal, R., Pappachan, P., Rozas, C., and del Cuvillo, V. P. J. Using Innovative Instructions to Create Trustworthy Software Solutions. Tech. rep., Intel Corporation, 2013.

[17] Ittai Anati and Shay Gueron and Simon P Johnson and Vincent R Scarlata. Innovative Technology for CPU Based Attestation and Sealing. Tech. rep., Intel Corporation, 2013.

[18] Koeberl, P., Schulz, S., Sadeghi, A.-R., and Varadharajan, V. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 10:1–10:14.

[19] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[20] McCune, J. M., Parno, B. J., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 315–328.

[21] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C., Shafi, H., Shanbhogue, V., and Savagaonkar, U. Innovative Instructions and Software Model for Isolated Execution. Tech. rep., Intel Corporation, 2013.

[22] Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A. V., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA* (2013), pp. 479–494.

[23] Rogaway, P., Bellare, M., and Black, J. OCB: A Block-cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Trans. Inf. Syst. Secur. 6*, 3 (Aug. 2003), 365–403.

[24] Santos, N., Raj, H., Saroiu, S., and Wolman, A. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, Microsoft Research, ACM, pp. 67–80.

[25] Strackx, R. *Security Primitives for Protected-Module Architectures Based on Program-Counter-Based Memory Access Control*. PhD thesis, Department of Computer Science, KU Leuven, December 2014.

[26] Strackx, R., Noorman, J., Verbauwhede, I., Preneel, B., and Piessens, F. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes, Information Security Solutions Europe* (2013), H. Reimer, N. Pohlman, and W. Schneider, Eds., Springer, pp. 241–251.

[27] Tanenbaum, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, NJ, USA, 2008.

[28] Texas Instruments. GCC - Open Source Compiler for MSP430 Microcontrollers, Feb. 2015.

[29] Texas Instruments. Ultra-Low-Power MSP430 Microcontroller, Feb. 2015.

[30] Trusted Computing Group (TCG), Incorporated. *Trusted Platform Module (TPM) Part 1, 2 and 3: Design Principles, Structures, and Commands*, Specification Version 1.2, Revision 116 ed., Mar. 2011.

[31] Wee, H. On Obfuscating Point Functions. In *STOC '05 (Proceedings of the thirty-seventh annual ACM symposium on Theory of computing)* (Baltimore, MD, USA, Apr. 2005), ACM, NY, pp. 523–532.

[32] Whiting, D., Housley, R., and Ferguson, N. Counter with CBC-MAC (CCM). RFC 3610 (Informational), Sept. 2003.