PUF-Based Software Protection for Low-End Embedded Devices

Florian Kohnhäuser[✉], André Schaller, and Stefan Katzenbeisser

Security Engineering Group, TU Darmstadt, Darmstadt, Germany
{Kohnhauser,Schaller,Katzenbeisser}@seceng.informatik.tu-darmstadt.de
http://www.seceng.de/

Abstract. In recent years, low-end embedded devices have been used increasingly in various scenarios, ranging from consumer electronics to industrial equipment. However, this evolution made embedded devices profitable targets for software piracy and software manipulation. Aggravating this situation, low-end embedded devices typically lack secure hardware to effectively protect against such attacks. In this work, we present a novel software protection scheme, which is particularly suited for already deployed low-end embedded devices without secure hardware. Our approach combines techniques based on self-checksumming code with Physically Unclonable Functions (PUFs) to establish a hardware-assisted software protection. In this way, we can tie the execution of a software instance to a specific device and protect its program code against manipulations. We show that our software protection scheme offers a high level of security against static adversaries and demonstrate that dynamic adversaries require considerable resources to perform a successful attack. To explore the feasibility of our solution, we implemented the protection scheme on an ARM-based low-end commodity microcontroller. A further performance evaluation shows that the implemented solution exhibits a fair overhead of ten percent.

1 Introduction

In recent years, the Internet of Things (IoT) [2] became one of the biggest buzzwords in the technology industry. With the IoT, billions of smart, interconnected embedded systems are proliferating virtually every aspect of our life. Nowadays, those devices can already be found in many everyday-life objects, such as consumer electronics, mobile devices, cars, smart-meters, or home appliances. On top of that, low-end embedded devices are widely used in industrial automation environments. However, the fact that embedded systems are increasingly deployed and typically lack effective security mechanisms aroused the interest of hackers, who started to realize that embedded devices are profitable targets. In practice, there are many attack scenarios on embedded devices.

One of the most tempting scenarios is the illegitimate reproduction of embedded systems, where an adversary reproduces existing devices by copying their firmware to counterfeit, cheaper hardware. Selling those cloned devices, adversaries cause financial loss for the manufacturer of the original system. In 2005,

M. Conti et al. (Eds.): TRUST 2015, LNCS 9229, pp. 3–21, 2015.
DOI: 10.1007/978-3-319-22846-4_1

KPMG estimated the sales lost to counterfeiters for fake electronic goods at 100 billion dollars [17]. Another attack scenario is the removal of license checking code, also referred to as software cracking. As an example, the attacker's mobile device may contain an application which requires a license. Instead of purchasing a license, the attacker bypasses the license check by manipulating the software. A further scenario is the play-back of Digital Rights Management (DRM) protected media on hardware media players, such as TV streaming devices. An attacker might insert code in the decryption function of the DRM player to intercept and extract the decrypted media. A famous example is the bypass of the DVD DRM encryption system CSS [26].

In order to protect against such attacks, the execution of the software must both be tied to a particular device and be secured against manipulations. To realize an effective hardware-software binding, hardware support is required. With hardware support, the security of a protected program rests on a secret, e.g., a cryptographic key or a piece of code implemented in a physical module. Prominent examples are the Trusted Platform Module (TPM), USB dongles, and cryptographic coprocessors. Nevertheless, integrated circuits dedicated to security are complex in their design, provoke deployment issues, occupy additional space on the underlying board, and lead to higher production costs. For this reason, especially legacy or low-end embedded devices lack hardware security mechanisms. However, as these devices are widely deployed and increasingly become the target of attacks [25], there is the need for a security solution that requires no specifically designed hardware.

1.1 Contributions

In this work, we explore a novel software protection approach, which is particularly suited for low-end embedded devices. Our approach combines and extends a self-checksumming code technique [15] with SRAM PUFs in commodity hardware [24] to protect a program against modifications and tie its execution to a dedicated device. Due to the usage of an intrinsic PUF as a secure key storage, our approach does not require any hardware modifications and thus can be easily retrofitted to already deployed devices. Furthermore, relying on a PUF significantly decreases the attack surface, as the secret is stored involving the PUF's physical properties. This makes physical attacks much more complicated compared to solutions based on non-volatile memory [1]. In order to explore the applicability of our solution, we implemented the proposed scheme on a low-cost ARM Cortex microcontroller. Various security parameters allow for a balancing between security and performance. Finally, a security and performance evaluation reveals that we achieve a substantial level of security with a performance penalty of ten percent.

1.2 Structure

In Sect. 2 we introduce PUFs and summarize existing work on tamper-resistant software. Section 3 presents our software protection solution. In Sect. 4 we evaluate the security of our approach. Section 5 depicts our implementation and evaluates its performance. Eventually, Sect. 6 concludes this work.

2 Related Work

Physically Unclonable Functions. Physically Unclonable Functions (PUFs) are physical objects that exhibit unique physical microstructures induced by manufacturing process variations. When a PUF is queried with a stimulus (challenge), it generates an unpredictable but repeatable response, which depends on the challenge and the PUF's physical structure. Typically, a PUF is assumed to exhibit characteristics of robustness, unclonability, unpredictability and tamper-evidence [22] (see Sect. 5.1). There exist various PUF implementations, ranging from optical and analogue PUFs to electronic PUFs [22]. The most significant PUFs for electronic circuits are delay- and memory-based PUFs. The former, e.g., arbiter PUFs or ring oscillator PUFs, utilize delays in their electronic circuits to generate the response. Memory-based PUFs exploit metastable states of digital memory primitives, such as SRAM or flip-flops, whose cells show a tendency to either initialize with the value zero or one. As not all memory bits show a stable initialization behavior, those bits introduce noise, which needs to be taken care of. For this purpose, Fuzzy Extractors are applied that remove the noise effect, which enables a robust reconstruction of an identifier [11].

Software Integrity Protection. Software integrity protection techniques deter attackers from modifying a particular software. They prevent adversaries from performing unauthorized actions, such as skipping a license check or playing a DRM-protected media file without the correct key.

Self-checksumming code is a common approach to protect a program against tampering [3,7,15]. The idea behind this concept is to equip a program with the functionality to verify its own integrity at runtime by calculating checksums for parts of its code. After a checksum is computed, it is compared to a precomputed reference checksum, indicating if the checked code has been tampered with. In this case a tamper-response (e.g., program termination) is initiated.

Oblivious hashing techniques pursue another approach. Instead of verifying parts of the program's machine-code, oblivious hashing mechanisms compute a hash value over the program's execution trace. In order to verify the program integrity, this hash value is then compared to a reference value. Typically, the hash value is computed over assignments and execution branches. Thus, instructions that monitor changes to variables and control flow are interweaved with the original code [8,16].

Result checking is a simple mechanism where, instead of verifying the program's code integrity, the result of certain computational operations is verified [5]. Checking the outcome of a computation can be considerably faster than performing the computation itself. For instance, a general sorting computation has order of $\mathcal{O}(n \cdot log(n))$ time complexity, whereas validating a sorted sequence takes $\mathcal{O}(n)$ time.

PUF-Based Software Protection. Gora, Maiti and Schaumont [12] proposed a system that implements a PUF instance on an FPGA to protect software and bind it to one hardware instance. At device start, they derive an 128-bit AES key from the PUF, utilize this key to decrypt the actual software code that was stored encrypted beforehand, and finally execute the decrypted software.

In a similar work of Schaller et al. [24], the authors presented an anti-counterfeiting solution which exploits inherent PUF characteristics from on-chip static random-access memory (SRAM) found in commodity devices. The authors propose to extract a unique device-dependent key from the SRAM PUF found in commodity devices. Using this key, the second-stage bootloader as well as the kernel of the device is decrypted during device start-up.

Nithyanand and Solis [23] show that traditional PUFs cannot solve the software protection problem in offline settings because they are vulnerable to *observe once, run everywhere* (OORE) attacks. To solve this problem, the authors propose the use of *intrinsic personal PUFs* (IP-PUFs). IP-PUFs are PUFs that are intrinsically and continuously involved in the computation of the program to be protected. In their proposed system, an IP-PUF computes the ordering of nodes in the control flow graph and enforces a random permutation of those nodes.

In summary, there are existing approaches that allow software to be integrity protected and tied to one device using PUFs. However, the security level existing solutions provide is comparatively low, since an adversary can dump the decrypted software at runtime. Once the adversary is in possession of the decrypted software, he can modify it or run it on other devices. By contrast, this work pursues a different approach, where self-checksumming code is combined with PUF responses to additionally provide security against attacks at runtime. Furthermore, the developed solution does not require any hardware modifications, which allows the deployment on commodity or legacy devices.

3 PUF-Based Software Protection Solution

Our software protection solution consists of four basic mechanisms: two *check* and two *response* functions. Check functions measure the authenticity of the device and the integrity of the program. Response functions read these measurements, decide whether they indicate a healthy or a manipulated state, and initiate a program misbehavior if a manipulation has been detected. In order to protect a software with our protection scheme, both functions are repeatedly integrated into the software's program code.

In more detail, the first check function measures the integrity of the software by hashing its native program code (see Sect. 3.1). The second check function computes a unique bitstream on the basis of a device-dependent SRAM PUF response to measure the authenticity of the device (see Sect. 3.2). If those two measurements indicate a manipulated state, the first response function redirects branches to random locations in the program text segment and the second response function corrupts the program's execution stack (see Sect. 3.3). Hence, if the program or the execution environment has been manipulated, both response functions cause a malfunction of the program.

3.1 Code Integrity Check

Principles. The integrity of the executable is measured by multiple self-checksumming code segments at runtime. Each segment consists of a hash function which computes a hash value over a predefined section in the program's text segment. The hash value represents the integrity status of the checked section. It is later used by response functions to decide whether the program has been tampered with. Depending on the spatial separation of the hash function and the response function, a hash value is either stored in a register or on the stack.

For stealth and security reasons, each hash function is inlined in the code, preferably with some spatial separation to other hash functions, and gets executed as the control flow passes the code location where the hash function is inserted. It is desirable that each inserted hash function is executed at least once at runtime, but not so frequently that the protected program suffers from a huge runtime overhead. In practice, profiling tools can be utilized to identify suitable code locations. We propose to let multiple hash functions measure a contiguous and relatively small part of the program. Thus, each integrity measurement consumes only little time. In addition, the effort for an attacker to remove the software protection increases.

In order to increase the effort even more, each code segment is measured multiple times by different hash functions. The so-called *overlap factor* indicates how often a code section is checked by different hash functions. Its value must be well-chosen to achieve a balance between security and performance according to the application scenario. To avoid that hash functions suspiciously measure large parts of the program, we recommend to split the program code in sections of equal size. These code regions are then uniformly assigned to hash functions till the overlap factor for each code region is saturated.

Hash Function Design. The design of our hash function is based on the work by Horne et al. [15]. With $d = [d_1, ..., d_n]$ being data in a code section which is protected by a hash function h, c being an odd multiplier constant, and $h_i(d)$ being the hash value in iteration i, our hash function can formally be defined as:

$$h_i(d) = \begin{cases} 0, & i = 1 \\ h_{i-1}(d) + c \cdot d_i. & 1 < i \leq n \end{cases} \tag{1}$$

We deviated from Horne's approach by not multiplying $h_{i-1}(d)$ with c in each iteration. This allows us to construct arbitrary complex mutually checking code regions (see Sect. 3.4). One reason we build on the code integrity check by Horne et al. is the hash function's size and speed. A large and slow hash function would fairly expand program size as well as runtime overhead, since the hash function is inlined frequently into the original program. However, the most important reason is stealth. An attacker who can locate all hash functions is able to break the code integrity check, for instance, by overwriting hash functions with code that always writes the respective expected hash value in memory. The proposed hash function neither contains any suspicious operations nor provides any characteristic pattern. In addition, its implementation in native program code can easily be diversified. Thus, each hash function can be customized, leaving the attacker no weak point for pattern matching attacks (see Sect. 4.1).

In order to customize hash functions, the odd constant c can be randomized, the addition can be replaced by a subtraction or an XOR operation, or a further constant can be added or subtracted after the multiplication with c. In addition, the hash function's implementation in native program code can be diversified, among others, by permuting the instruction order, permuting the assignment of variables to CPU registers, or diversifying particular instructions. A further possibility is to split the hash function code into multiple segments which are inserted with spatial separation in the original program code. With these techniques, it is straightforward to generate multiple million different hash function implementations.

Another attack vector is the code read operation performed by the hash function. It allows an attacker to find the location of hash functions by searching the code for addresses within the text segment, or observing if and where certain registers obtain values within the text segment at runtime. To mitigate this threat, we propose to implement Horne's memory access obfuscation approach [15] which uses an additional offset when addressing data in the program text segment (e.g., with the instruction `LDR Rd, [Rn, Rm]` on ARM-based platforms). In this way, text section addresses neither appear in the code nor in a register at runtime.

3.2 Device Authenticity Check

Principles. Recent work by Schaller et al. [24] have shown that SRAM modules present in several microcontrollers can be used as a PUF instance. In the device authenticity check mechanism, we use the microcontroller's SRAM PUF start-up values to compute a device-dependent bitstream. Since the SRAM PUF is unique and highly integrated in the microcontroller, the bitstream is unique for each embedded device. For these reasons, our response functions utilize the bitstream to authenticate the device at runtime.

The code for the bitstream generation is inserted into the device's bootloader. Hence, the bitstream is generated each time the device is starting up. In particular, a pseudorandom number generator (PRNG) is applied to allow for a variable bitstream length. In this way, a tradeoff between performance and security can be achieved. A larger bitstream takes more time to compute at device start-up but provides more unique values that can later be verified by response functions. Alternatively, it would be possible to gradually create the bitstream during program execution. However, as this further increases the execution overhead, we decided to precompute the entire bitstream in advance.

PRNG Bitstream Generation. Generating the PRNG bitstream comprises an enrollment and a reconstruction phase. The enrollment phase is performed at a trusted site, e.g., by the software integrator, and involves taking a reference

PUF measurement and equipping the device's bootloader with code and helper data to reconstruct a unique and reliable bitstream. During reconstruction, which is performed after deployment at the side of the user, the equipped bootloader is executed. Thus, the actual bitstream is generated using the PUF start-up values and additional error correction methods. To correct the raw PUF start-up values from noise, they are processed by error correction mechanisms. For this purpose, we integrate a Fuzzy Extractor (FE) based on the design by Bösch et al. [6] in the bootloader. The techniques used in the following to restore a predefined secret from SRAM cells are based on the work by Schaller et al. [24].

The *enrollment phase* is performed during the deployment of our software protection scheme once for each device. Initially, a unique random secret S is chosen. Using the FE with a reference PUF measurement and the secret S as input, so-called Helper Data is generated and stored on the device. The Helper Data is required in the reconstruction phase to retrieve S from a single noisy PUF measurement. Afterwards, the length for the PRNG bitstream is set, balancing security, speed, and storage consumption for the particular device and use case. At last, it is set at which location the bitstream is stored in memory during the reconstruction phase.

The *reconstruction phase* is executed each time the device is started. Initially, the bootscript measures and stores the noisy SRAM PUF values R'. Next, the FE reconstructs a secret S' using the current PUF measurement R' and the stored Helper Data as input. If the PUF measurement R' corresponds to the respective Helper Data, the reconstructed secret S' will match the original secret S. S' is then used to initialize the PRNG which finally generates a PRNG bitstream of the set length in memory.

3.3 Response Functions

Principles. Before a response function is inserted into the code, it is randomly selected whether the response function verifies a hash value, a value of the PRNG bitstream, or both values at once. If a response function verifies a hash value, it uses the hash value of the nearest preceding hash function. This ensures that hash values are verified shortly after they are measured, thwarting code manipulations promptly after they have been detected. If a response function verifies a value of the PRNG bitstream, it uses a random preferably nonrecurring bitstream value. The basic idea is to use a unique address in each PRNG bitstream access. Thus, a single address cannot be used as an attack vector for pattern matching attacks or as a watchpoint in dynamic analyses. However, if there are less PRNG bitstream values than deployed response functions available, some addresses must be used multiple times.

The overall goal of our two response functions is to provoke a malfunction of the protected program if the measured code integrity or device authenticity values are invalid. We would like to point out that a malfunction of the program may lead to a damage of the machine that is controlled by the program. However, the alternative to perform a deterministic action (e.g., a controlled program shutdown) would provide an easy attack vector for the adversary. In this scenario, the adversary could simply observe where the program shutdown is initiated, to locate the response functions in the code.

Indirect Branch Response. The indirect branch response is applicable on any branch in the program. When applied, an original branch is converted to an indirect branch whose target address is dependent on the verified values, i.e., either on a hash value, on a value of the PRNG bitstream, or on both values. The exact target address of the indirect branch is determined by a computation which meets the following requirements.

The output of the computation must equal the target address of the replaced original branch if the verified values correspond to their expected values. If at least one of the verified values is corrupted, the outcome of the computation must be a random address that lies within the program text segment. The latter requirement ensures that the computed target address is always a valid instruction that can be executed. If the computation of the target address would not generate a valid address in the text segment, program manipulations would immediately cause memory access violations. This would be very suspicious and allows the attacker to easily locate the response function with backtraces.

In practice, the behavior of the indirect branch tamper response is highly dependent on the program size and the structure of the program code (e.g., the number of functions in the program). We observed, on average, about two function calls until a memory access violation occurred after the indirect branch response was executed.

As an additional requirement, the computation of the target address must be simple. In order to improve stealth, its implementation should be short and should not contain unusual instructions. To improve stealth even more, each deployment of the indirect branch response function should be customized, for instance, with the techniques presented in Sect. 3.1.

Stack Manipulation Response. In contrast to the indirect branch response, the stack manipulation response can be deployed at arbitrary locations in the program code. When deployed, we propose to use one stack manipulation response per hash function to ensure that each code measurement is eventually verified by a response function.

The idea behind the stack manipulation response is to corrupt the execution stack if the verified values are invalid. Hence, in case of an unauthorized modification, local variables, function arguments, register copies, return addresses, and other data that lies on the stack, are altered. As a result, the program continues execution with incorrect values.

A simple way to accomplish a modification of all values on the stack is to shift the stack pointer. Shifting the stack pointer has two benefits. First, it mixes up stack frames, which complicates a backtracing the program. Second, it modifies the return address and thus provokes a program crash when the currently executed function returns. If an eventual program crash as a tamper-response is not desirable, we propose to alter values on the stack directly.

3.4 Mutually Checking Code Regions

Since the presented protection mechanisms secure the entire program code and at the same time are also part of the program code, they secure each other against modifications as well. Although this enhances the security of a protected software, it comes at the cost of emerging circular dependencies in the deployment process. These mutual dependencies occur because at some point code protection measures, consisting of a hash function and a response function which verifies the hash function's value, circularly check each other.

In the work by Horne et al. [15], code regions are assigned to hash functions in a left-to-right pass which generates no mutual dependencies. However, with this approach, the overlap factor is comparatively low at the beginning and the end of the program code. In fact, their overlap factor goes down to a factor of one in the first and last few bytes of the program code. By contrast, we propose a uniform assignment of hash functions to code regions and a subsequent solving of the upcoming circular dependencies. Thus, we can ensure a consistent overlap factor throughout the entire program code.

When solving cyclic checks, the first step is to transform mutually checking code regions into an equation system. For this purpose, we initially deploy all protection mechanisms into the software and build a temporary protected binary. The protected binary contains the final code, except for the response functions' reference values and additional placeholder values. We propose to insert one freely selectable 32-bit placeholder value per code integrity measure to facilitate solving the equation system. Next, we utilize the fact that hash values can be written as the sum of multiple data values. With $d = [d_1, ..., d_r, ..., d_p, ..., d_n]$ being a list of n 32-bit words in a code section, where d_r is a reference value, d_p is a placeholder value, and c being the hash function's multiplier constant, a hash function h which measures this code section on a 32-bit microprocessor can be written as:

$$h(d) \equiv \underbrace{c \cdot d_r + c \cdot d_p}_{l} + \underbrace{\sum_{\substack{i \neq r \\ i \neq p}}^{n} c \cdot d_i}_{r} \quad (mod\ 2^{32})\ . \tag{2}$$

In this way, hash values are divided in a variable part l, containing the reference value d_r and the additional placeholder value d_p which are to be solved, and a fixed part r, containing the rest of the code segment. Since the code data d_i and the multiplier constant c are fixed after deployment, r can easily be computed. Next, reference values must be expressed in relation to hash values and PRNG bitstream values. The exact dependence between PRNG, hash, and reference value is given by the response function in which the reference value is used. Finally, these relations are combined to one linear Diophantine equation system which is then solved according to the approach of Lazebnik [20].

4 Security Evaluation

Information security mechanisms like cryptographic primitives or secure protocols are commonly designed to be secure in the black-box model. However, we assume a much more challenging scenario where the attacker is in possession of the endpoint devices and thus has access to the implementation and power over the execution environment. This security model is referred to as white-box model [14]. Taking the white-box model as a basis, we specify two attacker models, the *static attacker* and the *dynamic attacker*. We generally expect both attackers to be familiar with our software protection model, albeit we assume that they do not know the particular deployed protection code, the location of the protection code, and aspects of our protection scheme which are randomized at deployment. The following sections specify the attacker models and evaluate the security of our software protection scheme against the respective model.

4.1 Static Attacker Model

Specification. A static attacker has the ability to perform static analysis on a device in his possession, i.e., he can read and modify all the data stored on the device. For instance, the attacker can read and modify the content of the external memory, like the flash memory or the RAM, or the internal memory, including the software with its hard-coded secrets and cryptographic keys. Additionally, we presume that the static attacker can run the program and observe its input-output behavior.

The static attacker model is a reasonable assumption for an experienced attacker who lacks the ability to debug the protected program. This may be the case due to the employment of anti-debugging techniques implemented in software (e.g., the exhaustion of breakpoint registers, or the use of API functions to check if a debugger is present) or in hardware (e.g., the physical removal of debugging ports).

Evaluation. Using a disassembler, a static attacker can analyze native program code and reverse engineer the protected program. In the worst case, the attacker would comprehend the complete code and thereby know how he can circumvent our protection mechanisms. In practice, though, this task is highly laborious, as even a small program consists of a few thousand lines of machine code.

One possibility to accelerate the analysis process is to look for outstanding instructions or specific patterns in the code. In a pattern matching attack, the attacker reveals the location of the protection code by extracting a pattern from found protection mechanisms and then searching the entire program code for that pattern. Therefore, we specifically avoided the use of suspicious operations by performing short and common computations only. The implementation of our hash function requires approximately 30 bytes (48 bytes with code access obfuscation) and the response function between 12 and 18 bytes. Additionally, we demonstrated in Sect. 3 that both mechanisms can easily be diversified repeatedly.

In another technique called collusion or differential attack, an adversary compares multiple versions of a protected program to spot the location of the inserted protection mechanisms in their differences. In order to protect against this attack, we can distribute our protection scheme to many devices with the same deployment preferences. In this way, a collusion attack would only reveal the location of the Helper Data which does not leak any information. A further approach would be to diversify the entire program in the deployment process [18].

A very common technique applied during a static analysis is the examination of the program's execution flow. With the deployment of the indirect branch response function, branches are replaced with indirect branches whose target addresses are dependent on hash values and values of the PRNG bitstream. As both values are not known to a static analysis tool, our approach can significantly reduce the amount of useful information that an attacker can extract from a control flow analysis.

The unpredictability of the PRNG bitstream in offline attacks has an additional advantage. Since both response functions occasionally utilize a value of the PRNG bitstream for their operation, their exact behavior cannot be predicted with static analysis techniques. Furthermore, both response functions can be used to perform essential operations in the program, i.e., branches or stack allocations. As a result, it is hardly possible for a static attacker to remove the hardware-software binding.

4.2 Dynamic Attacker Model

Specification. A dynamic attacker inherits all abilities from the static attacker. Furthermore, the dynamic attacker has the ability to read and modify all the data on a device at runtime. With these abilities, the attacker can interrupt a program at any time, single-step through the program code, and inspect or modify memory values at runtime. Moreover, the attacker has the capability to modify a program's execution environment. He might force the program to use bogus dynamic libraries, altered operating system functionalities, or run the program in a virtual machine.

We are aware that an attacker with the stated abilities and enough resources in time and money is capable of breaking any software security mechanism. Therefore, our goal is to increase the effort for a successful attack to a level where an attack becomes uneconomical.

Evaluation. One of the most powerful debugging features when analyzing a protected program are watchpoints. Watchpoints are used to halt the execution whenever the program accesses predefined memory locations. A dynamic attacker can use this technique to locate a large fraction of all response functions by recurrently setting watchpoints on values of the PRNG bitstream while executing the program with various input. In addition, by setting watchpoints on addresses within the program text segment, the attacker can locate hash functions. A subsequent tracing of the hash functions' hash values can reveal the location of all remaining response functions. Having located all response functions, the attacker can remove the verification of hash and PRNG bitstream values and thus disable our software protection. Although the described approach is eventually successful, it requires a significant amount of effort from an attacker. Furthermore, the effort can be arbitrarily augmented by increasing the number of hash functions, setting a higher overlap factor, or obfuscating access on hash values and values of the PRNG bitstream.

Another common dynamic analysis technique is tracing. Tracing a program involves logging information during the program's execution, such as the execution path, memory values, or register values. A dynamic attacker may trace back program crashes or abnormal program behavior to localize response functions. During the design of our response functions, we ensured that there is a large spatial and temporal separation between the execution of the response function and its impact on the program, i.e., a program crash or a program misbehavior. Thus, the attacker has to examine a large portion of the trace back to finally localize a single response function.

Profiling is an additional dynamic analysis technique which involves measuring particular runtime performance values. In general, our protection mechanisms do not consume exceptionally much CPU time or memory. But yet, profiling a protected program may reveal the location of deployed hash functions when the execution of a hash function takes exceptionally long time compared to the execution time of the original program. Anyhow, profiling requires about the same effort as the above described approach with watchpoints, as the protected program must be examined multiple times with different input. On top of that, the profiling approach is less reliable than the watchpoint approach, because code that is often run through need not be part of a hash function.

Emulation is a further dynamic analysis technique. An emulator is software which simulates the behavior of a particular hardware platform. With emulation, an adversary can bypass the hardware-software binding by emulating particular PUF start-up values. In addition, an adversary can redirect data access to the unmodified version and code access to the modified version of a protected program, to bypass our code integrity protection. Nevertheless, emulation attacks are unpractical, because the software has to run in an emulator and cannot run directly on the hardware of an embedded system. In addition, the performance is slower, an emulator is hard to implement, and the protected programs PRNG bitstream must be extracted.

With temporary modifications or on-the-fly writes in memory, the attacker modifies a code region before its execution and recovers it to its original form afterwards. If an adversary inserts his modification just before it is executed and restores the original code immediately after the modified code has been executed, we cannot defend against this attack. However, this requires the attacker to permanently attach a debugger to the program, to write a debugger script which performs the attack without manual intervention, and to accept a loss in performance because of multiple code manipulations at runtime.

5 Proof of Concept

In order to explore the applicability of our software protection scheme, we implemented and evaluated it on the Stellaris EK-LM4F120XL microcontroller. The Stellaris board is a low-end embedded system featuring an 80 MHz ARM Cortex-M4F microprocessor. During deployment, the protection mechanisms are inserted into the source code of the program to be protected. Subsequently, the LLVM compiler framework [19] with Clang front-end [10] is used to compile the equipped source code to the final protected binary. For this purpose, we wrote a Python script which controls LLVM, Clang, and additional external tools and libraries to automatically build the protected program. In the following sections we give details on the intrinsic PUF instance of the Stellaris board, the implementation of the protection scheme, and the performance of our implementation.

5.1 PUF Characteristics

Before using a SRAM PUF instance in security critical applications, it is crucial to characterize the SRAM start-up values for constructing an efficient Fuzzy Extractor (FE) and extracting a secret with full entropy. In order to obtain sufficient measurements, we used a hardware setup comprising 15 Stellaris boards. The Stellaris boards were connected to a custom microcontroller, which in turn was connected to our terminal PC. This setup allowed us to repeatedly query each of the boards automatically. In particular, the microcontroller was programmed to toggle an individual device, executing the modified bootloader (see below for details) and sending the SRAM start-up values over UART back to the microcontroller. Subsequently, the measurements were forwarded to the terminal PC, where they were saved and post-processed. Using this setup we generated 1000 measurements per device.

In the following we present numbers for metrics, which are generally used to evaluate the quality of a PUF instance. The *Hamming Weight* (HW) of measurements from the same device indicates a potential bias towards zero or one. This metric provides a first impression on the entropy present in the start-up values. The *Within-Class Hamming distance* (WCHD) indicates the robustness of measurements from a single device. In particular, it shows how many bits were flipped during repeated start-ups and therefore represents the noise level. The *Between-Class Hamming distance* (BCHD) reveals the independence of start-up

Table 1. Metrics from 15 Stellaris boards with 1000 measurements per device.

Metric	Value [%]
Fractional HW (min; max)	43.29; 53.69
Fractional WCHD (max)	5.25
Fractional (avg)	49.33
min-entropy (min)	5.86

values from different devices and thus shows whether the PUF can be used to uniquely identify a given device. Lastly, the *min-entropy* was calculated to quantify the randomness of the start-up values. To do so, we adapted the well-known approach to calculate min-entropy [21], assuming independence between all bits from the start-up pattern [4,9] and each individual bit to be a binary source. In Table 1 numbers for these metrics are shown, attesting that the Stellaris board has almost ideal PUF characteristics.

5.2 Implemented Protection Mechanisms

Hash Function. In Sect. 3 we stated that it is vital for the security of our protection scheme to deploy syntactically different hash functions. In order to have precise control over the hash function's native program code, we inline hash functions as ARM assembler code in the program source code. To avoid the usage of unusual instructions in our ARM assembler version of the hash function, we implemented the hash function in C and compiled it to get an assembly language prototype. A so generated ARM assembly prototype is show in the following code snippet. It hashes a code region from address `0x26c` to `0x2ac` with the multiplier constant c = 3:

```
movs  r1, #0             // hash = 0
movw  r2, 0x26c          // start = 0x26c
movw  r3, 0x2ac          // end = 0x2ac
loop:
ldr   r4, [r2], 4        // tmp = data[i], start++
add   r4, r4, r4, LSL#1  // tmp = 3*tmp
add   r1, r4             // hash = hash + 3*tmp
cmp   r2, r3             // if (start < end)
blt   loop               // then goto loop
```

PRNG Bitstream Generation. During deployment, we substitute the pre-existing Stellaris bootloader with a modified version that contains our PRNG bitstream generation code. Besides the standard initialization code, the modified bootloader contains code for the extraction of the PUF start-up values, a FE based on the design by Bösch et al. [6], Helper Data to reconstruct a predefined secret, and a PRNG based on the Keccak (SHA-3) implementation of Herrewege et al. [13]. At first, the bootloader extracts 240 bytes of PUF start-up values. For this purpose, we added ASM code to the power-on reset vector, which configures a GPIO to be used as a UART port. In a next step the code iterates over the memory region of the SRAM, putting each byte out over UART. Afterwards, the original code is resumed, relocating the firmware to SRAM and executing it. Next, the FE reconstructs a predefined 128 bit secret using the PUF start-up values and the Helper Data. Here, we reuse Keccak in the privacy amplification phase of the FE. The reconstructed secret is used to initialize the PRNG. We use Keccak as a PRNG, primarily because of its compact size and speed on ARM devices. For the length of the bitstream, we suggest to use 2^{17} bits, which provides

4096 unique values and consumes 16 KiB of memory at runtime. Nevertheless, the bitstream length can be set to an arbitrary value, for instance, to consume less storage.

Indirect Branch Response Function. During deployment, existing branches in the original source code are overwritten with the code of the indirect branch response function. The target address of the indirect branch is computed by the sum of the verified values and a specific offset, modulo a unique value, plus a unique value. The following code snippet in C syntax illustrates an indirect branch to a function, which takes no argument and returns void (e.g., `void foo(void)`):

```
void (*foo)(void);
foo = ((*hash_value + *puf_prng_value + *offset) % *modulo) + *shift;
foo();
```

If the hash value and the PRNG value match their expected values, *offset*, *modulo*, and *shift* adjust the indirect branch to match the original target address. In order to provide no constant value as an attack vector for pattern matching attacks, *modulo* and *shift* are randomized between certain bounds in each deployment of the indirect branch response function.

Stack Manipulation Response Function. We insert each stack manipulation response function randomly between the location of the corresponding hash function and the subsequently executed hash function. Our implemented stack manipulation response function sums all values to be verified and checks whether the result is equal to the expected value. If it is not, the stack pointer is either incremented or decremented by a random value between 4 and 24 bytes.

5.3 Performance Evaluation

Due to the lack of open source applications for the Stellaris platform, we developed our own evaluation program. The evaluation program encrypts and decrypts a 16 bytes string using AES 128 bit, sends the plaintext and the ciphertext to the UART port, and measures the amount of CPU cycles consumed from the start to the end of the main function.

For the deployment of our software protection scheme, we used the following security settings. We inserted one code integrity check mechanism in each function of the evaluation program. As 9 of the 11 deployed functions are executed at runtime, we generate a coverage of 82 %. This is a realistic scenario, as a real application will certainly contain functions that are not always executed at runtime (e.g., whose execution depends on specific user input). In addition, we used a PRNG bitstream length of 2^{17} bits, which corresponds to a size of 16 KiB. For the deployment of the response functions, we inserted the stack manipulation response in each circular dependent code region and the indirect branch response in the remaining code.

Runtime Performance. In our runtime evaluation, we deployed the evaluation program with the above mentioned security settings and a variable overlap factor. Figure 1 illustrates the relative average runtime overhead for various overlap factor preferences. The runtime of the original unprotected program is represented with an overlap factor of zero and an overhead factor of one. As the overlap defines how many times a code region is checked by different hash functions, an increasing overlap factor increments the amount of code lines that each hash function has to check. With an overlap factor of nine, each hash function almost checks the complete text segment, which generates an overhead of approximately half of the original runtime. It is evident that such an overhead is not acceptable in most applications. On the other hand, even when each code region is checked by three different hash functions, the runtime overhead is below 5 %. As this slow-down will only be noticed by sensitive users, we can easily recommend an overlap factor of three for a conservative usage.

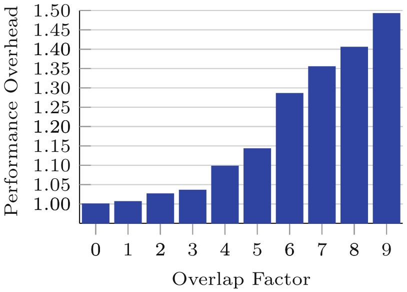

Fig. 1. Runtime performance comparison with different overlap factor settings.

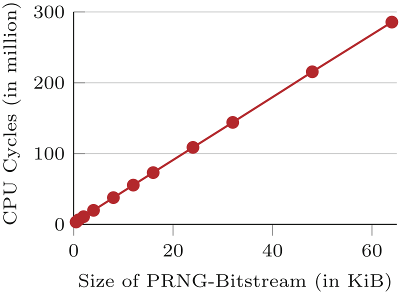

Fig. 2. Start-up runtime performance with varying bitstream size.

Another performance overhead originates from the generation of the PRNG bitstream at device start-up. Figure 2 depicts the amount of CPU cycles that is required to compute a bitstream of a specific length. For comparison, the original program consumes roughly 1.8 million CPU cycles.

The figure illustrates that there is almost a proportional relationship between the size of the PRNG bitstream and the amount of CPU cycles. Thus, compared with the calculation of the pseudorandom values, the extraction of the PUF start-up values and the execution of the Fuzzy Extractor barely uses any CPU time. The figure also shows that the generation of the PRNG bitstream consumes much more CPU resources than the execution of the actual program. However, it must be considered that the PRNG bitstream is only generated at device start-up. Assuming the embedded devices is clocked at 50 MHz, a bitstream size of 16 KiB delays the start of the device by 1.5 s which is likely to be acceptable for most applications.

Storage Consumption. The program size overhead of a protected program is dependent on the number of inserted hash functions, the choice of the response function, and the number of inserted response functions. For evaluation, we deployed our protection mechanisms using the previously mentioned security settings. In this way, we obtained a protected program which was on average 63 % larger than the equivalent unprotected program. Another storage overhead arises at runtime due to the operating of both check mechanisms. However, the hash functions' memory consumption is negligible, as each value resides just a short time in memory and only occupies 4 bytes of storage. In contrast, the values of the PRNG bitstream are kept in memory permanently and they consume 16 KiB of memory for our proposed bitstream length. Nevertheless, by setting another bitstream length, the runtime memory overhead can be adjusted as required.

6 Conclusion

In this work, we explored a novel hardware-assisted software protection approach, which combines existing software-based techniques with PUFs. Using a microcontroller's SRAM as a PUF instance, we overcome the drawbacks of traditional hardware tamper-proofing solutions. Our software protection scheme ties the execution of a software instance to a specific device, protects its program code against manipulations, and can easily be retrofitted to already deployed devices. To demonstrate our approach, we implemented it on a low-cost ARM-based microcontroller. By adjusting certain security parameters, we are able to balance security with performance. We showed that our software protection scheme offers a high level of security against a static adversary and demonstrated that a dynamic adversary requires a considerable amount of resources to perform a successful attack. A further performance evaluation showed that an extensive level of security is achievable with an acceptable performance degradation of ten percent.

References

1. Armknecht, F., Maes, R., Sadeghi, A.-R., Sunar, B., Tuyls, P.: Memory leakage-resilient encryption based on physically unclonable functions. In: Sadeghi, A.-R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security. Information Security and Cryptography, pp. 135–164. Springer, Heidelberg (2010)
2. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
3. Aucsmith, D.: Tamper resistant software: an implementation. In: Anderson, R. (ed.) Information Hiding, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
4. van den Berg, R., Skoric, B., van der Leest, V.: Bias-based modeling and entropy analysis of PUFs. In: ACM Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices TrustED (2013)
5. Blum, M., Kannan, S.: Designing programs that check their work. J. ACM JACM **42**(1), 269–291 (1995)
6. Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., Tuyls, P.: Efficient helper data key extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 181–197. Springer, Heidelberg (2008)

7. Chang, H., Atallah, M.J.: Protecting software code by guards. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 160–175. Springer, Heidelberg (2002)
8. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious hashing a stealthy software integrity verification primitive. In: Petitcolas, F.A.P. (ed.) Information Hiding. LNCS, vol. 2578, pp. 400–414. Springer, Heidelberg (2003)
9. Claes, M., van der Leest, V., Braeken, A.: Comparison of SRAM and FF PUF in 65nm technology. In: Laud, P. (ed.) NordSec 2011. LNCS, vol. 7161, pp. 47–64. Springer, Heidelberg (2012)
10. Clang: A C language family frontend for LLVM. http://www.clang.llvm.org/
11. Dodis, Y., Reyzin, L., Smith, A.: Fuzzy extractors: how to generate strong keys from biometrics and other noisy data. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 523–540. Springer, Heidelberg (2004)
12. Gora, M.A., Maiti, A., Schaumont, P.: A flexible design flow for software IP binding in commodity FPGA. In: IEEE Symposium on Industrial Embedded Systems IEEE SIES (2009)
13. van Herrewege, A., Verbauwhede, I.: Software only, extremely compact, keccak-based secure PRNG on ARM Cortex-M. In: ACM Proceedings of the 51st Annual Design Automation Conference (2014)
14. Herzberg, A., Shulman, H., Saxena, A., Crispo, B.: Towards a theory of white-box security. In: Gritzalis, D., Lopez, J. (eds.) SEC 2009. IFIP AICT, vol. 297, pp. 342–352. Springer, Heidelberg (2009)
15. Horne, B., Matheson, L., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper resistance. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 141–159. Springer, Heidelberg (2002)
16. Jacob, M., Jakubowski, M.H., Venkatesan, R.: Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings. In: ACM Workshop on Multimedia & Security MM&Sec (2007)
17. KPMG: Managing the Risks of Counterfeiting in the Information Technology Industry. http://www.agmaglobal.org/press_events/press_docs/Counterfeit_WhitePaper_Final.pdf. Accessed 23 June 2015
18. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: IEEE Symposium on Security and Privacy S&P (2014)
19. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: IEEE Symposium on Code Generation and Optimization (2014)
20. Lazebnik, F.: On systems of linear diophantine equations. In: Mathematics Magazine (1996)
21. van der Leest, V., van der Sluis, E., Schrijen, G.-J., Tuyls, P., Handschuh, H.: Efficient implementation of true random number generator based on SRAM PUFs. In: Naccache, D. (ed.) Cryphtography and Security: From Theory to Applications. LNCS, vol. 6805, pp. 300–318. Springer, Heidelberg (2012)
22. Maes, R., Verbauwhede, I.: Physically unclonable functions: a study on the state of the art and future research directions. In: Sadeghi, A.-R., Naccache, D. (eds.) Towards Hardware-Intrinsic Security, pp. 3–37. Springer, Heidelberg (2010)
23. Nithyanand, R., Solis, J.: A theoretical analysis: physical unclonable functions and the software protection problem. In: IEEE Symposium on Security and Privacy S&P (2012)

24. Schaller, A., Arul, T., van der Leest, V., Katzenbeisser, S.: Lightweight anti-counterfeiting solution for low-end commodity hardware using inherent PUFs. In: Holz, T., Ioannidis, S. (eds.) Trust 2014. LNCS, vol. 8564, pp. 83–100. Springer, Heidelberg (2014)
25. Schneier on Security: Security Risks of Embedded Systems. https://www.schneier.com/blog/archives/2014/01/security_risks_9.html. Accessed 23 June 2015
26. Wikipedia: DeCSS. http://www.en.wikipedia.org/wiki/DeCSS. Accessed 23 June 2015