Software Protection using Dynamic PUFs

Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer

***Abstract*—Low-end computing devices are becoming increasingly ubiquitous, especially due to the widespread deployment of Internet-of-Things products. There is, however, much concern about sensitive data being processed on these low-end devices which have limited protection mechanisms in place. This paper proposes a Hardware-Entangled Software Protection (HESP) scheme that leverages hardware features to protect software code from malicious modification before or during run-time. It also enables implicit hardware authentication. Thus, the software will execute correctly only on an authorized device and if the timing of the software, e.g., control flow, was not changed through malicious modifications. The proposed ideas are based on the new concept of *Dynamic* Physically Unclonable Functions (PUFs). Dynamic PUFs have time-varying responses and can be used to tie the software execution to the timing of software and the physical properties of a hardware device. It is further combined with existing approaches for code self-checksumming, software obfuscation, and call graph and register value scrambling to create the HESP scheme. HESP is demonstrated on commodity, off-the-shelf computing devices, where a DRAM PUF is used as an instance of a Dynamic PUF. The protection scheme can be applied automatically to LLVM Intermediate Representation (IR) code through an AutoPatcher written in Python. For a sample program containing AES encryption and decryption routine, HESP introduces 48% execution time overhead and increases the binary file size by 32.5%, which is moderate compared to other schemes such as software obfuscation. It takes about 2.6 seconds on average for the tested programs to be patched and compiled through the modified compilation flow and scripts.**

***Index Terms*—Physically Unclonable Functions (PUFs), Software Protection, DRAM, Hardware-Software Binding**

I. Introduction

The number of low-end embedded computing devices is continuously growing, and this growth is expected to accelerate further due to the popular interest in the Internet-of-Things (IoT). These low-end devices are being deployed in a variety of settings from healthcare to industrial environments, where the integrity and tamper resistance of the software is critical. For example, in an industrial control system, compromising the integrity can lead to life-threatening hazards and economic loss. Yet, today, these systems still lack advanced security features, mostly due to cost constraints.

In the past decades, researchers have developed software schemes to make software tamper-resistant, which could be used in IoT devices deployed today. For example, code self-checking schemes [1], [2], [3] were proposed to verify the integrity of software statically or at runtime. In such approaches, a hash value or a checksum of a certain code block is computed. If the computed hash value matches an expected reference value, the code block is regarded as unmodified and can be executed normally. Moreover, the code that computes the hash value and the reference value can be included in a code block that is checked by another hash, which is a part of another code block that is hashed and checked, and thus hash functions will check each other mutually. In another branch of research, software obfuscation focuses on protecting software from reverse engineering [4], [5], making it difficult for the attacker to change the software to perform some malicious functionality. Software obfuscation can thus provide some protection from malicious modification. There are many obfuscation techniques, such as using aliases [6], reordering instructions [7], converting static data to procedures [4] and flattening the control flow graph [8], [9], [10]. Meanwhile, software diversity has been proposed to add randomness to each instance of an executable, so that an attack which targets one instance cannot be applied to other instances [11].

In a parallel research area, hardware support in the form of Physically Unclonable Functions (PUFs) can be readily used in low-end devices [12], [13], [14], [15], [16]. PUFs extract unique and stable physical features from the underlying hardware modules. These features emerge from fabrication variations of the hardware, are expected to be unique to each device, and are extremely difficult to replicate physically. Among the different PUF types, *intrinsic PUFs* leverage hardware already found on a computing platform, e.g., SRAM PUFs [15] and DRAM PUFs [17], [18]. Prior research [19] has shown that PUFs combined with self-checking schemes can protect software from modification and from running on unauthorized devices.

Meanwhile, in this work, we bind software execution to hardware with *Dynamic PUFs*. Especially, a Dynamic PUF can generate responses at device runtime, and the responses depend on the timing of the queries to the Dynamic PUF. This is different from existing types of PUFs which we call *Static PUFs*. We then propose the Hardware-Entangled Software Protection (HESP) scheme, which protects software from unauthorized modifications and from running on unauthorized platforms. To verify whether the software is running on an authorized device, the HESP leverages the unique output of a Dynamic PUF. In addition, due to the time-dependent Dynamic PUF responses and self-checksumming functions, the behavior of software is tied to the execution time of the software. Malicious modifications, such as changes to the control flow, will affect the execution time, causing queries to the Dynamic PUF to be triggered at unexpected times, resulting in wrong Dynamic PUF responses, thus making the software execute incorrectly.

W. Xiong and J. Szefer are with Yale University, New Haven, CT, USA. E-mails: {wenjie.xiong, jakub.szefer}@yale.edu

A. Schaller was with Technische Universität Darmstadt, Darmstadt, Hessen, Germany. E-mail: andre@andreschaller.de

S. Katzenbeisser is with University of Passau, Passau, Bayern, Germany. E-mail: stefan.katzenbeisser@uni-passau.de

A. Contributions

This paper is an expanded version of our conference publication [20], which introduced the concept of Dynamic PUFs. This work extends the prior paper with the following new contributions:

- We propose a new Hardware-Entangled Software Protection (HESP) scheme.
- We implement a fully automatic framework that deploys the HESP to C source code through a framework called AutoPatcher written in Python, which patches LLVM Intermediate Representation (IR) to apply the protection code.
- We develop a practical decay-based Dynamic DRAM PUF with a controlled PUF interface and DRAM cells on Intel Galileo Gen 2 platforms and evaluate programs protected by HESP on them.
- The HESP framework and the Dynamic PUF code developed in this project will be made available under an open-source license at https://caslab.csl.yale.edu/code/puf-software-protection/.

II. Dynamic PUFs

The proposed HESP protection scheme leverages Dynamic PUFs [20], which were introduced in our prior conference paper, to provide time-dependent PUF responses. With the time-dependent Dynamic PUF behavior, the software can execute correctly only with correct Dynamic PUF responses. This ensures that if the program does run, it is running on the authorized device and not modified as the Dynamic PUF instances are queried at the correct time.

Dynamic PUFs are a type of PUF. Generally, PUFs extract unique and stable physical features from physical objects [14], [15]. Silicon PUFs exploit unique features that emerge due to fabrication variations of integrated circuits and are regarded to be extremely difficult to be cloned physically. Given a challenge, a PUF can generate a stable response, which is a function of both the challenge and the physical features of the PUF, known as a challenge-response pair (CRP). Ideally, for the same challenge, each physical object used as a PUF will generate a unique PUF response.

Most existing PUFs are static: for each challenge, there is a noisy but stable response, dependent on the physical features of the hardware and independent of the timing of the PUF challenge. Delay-based PUFs [14] and most memory-based PUFs [15] are examples of Static PUFs.

Meanwhile, this work presents the notion of a *Dynamic PUF* and uses it for software protection. Different from the usual Static PUFs, Dynamic PUFs provide a time-dependent PUF response, i.e., the response depends not only on the challenge but also on the timing of the PUF query. Since the response depends on time, the Dynamic PUF has an extra query called *Dynamic PUF Reset*. A Dynamic PUF gives different responses even if queried with the same challenge, but at a different time relative to the most recent *Dynamic PUF Reset*.

A. Helper Data System (HDS) for Dynamic PUFs

The ideal Dynamic PUF responses should have two features: First, the responses at different times should be independent of each other, i.e., the response at time T_x should be independent of the response at time T_y, where $|T_y - T_x| > \delta_t$ and δ_t is the time resolution; Second, the responses should be stable. Since noise exists in raw Dynamic PUF responses, error correction needs to be performed on the extracted PUF responses.

To achieve both features, a set of Helper Data System (HDS) [21], [22] is needed to extract a PUF response that is specific to the query time from the raw PUF response, i.e., each query time T_x should have its own HDS entry.

To retrieve the PUF response for a certain query time, the query should indicate which helper data to use, by specifying an index idx to the HDS. If the wrong HDS entry is used, a wrong PUF response will be returned. Each Dynamic PUF query time T_x corresponds to an entry in the HDS indicated by idx. To generate enrollment data and the HDS, a trusted party needs to measure the desired PUF CRPs in an *enrollment* phase. This can be the same party who applies the HESP scheme. An HDS construction for Dynamic DRAM PUFs and its enrollment is described in Section VII-D.

B. Metrics for Dynamic PUFs

Other than the inter and intra distance metrics commonly used to evaluate Static PUFs, the time-dependent behavior of Dynamic PUFs can be evaluated with the following metrics:

- **Time Resolution:** Denoted by δ_t, time resolution indicates how sensitive the PUF responses are to the query time. Query times that differ by more than δ_t are considered to be different query times, and ideally, the Dynamic PUF should give different responses even after ECC is applied. Each realization of a Dynamic PUF has a physical limit of the time resolution.
- **Time Range:** Denoted by $[T_{min}, T_{max}]$, time range indicates the range of PUF query times relative to the PUF reset that can result in Dynamic PUF behavior. Within time range $[T_{min}, T_{max}]$, the PUF responses have enough entropy. In real applications, the user should query the Dynamic PUFs after T_{min} and before T_{max}. Thus, we also call T_{min} the initialization time of Dynamic PUFs. If an application needs time longer than T_{max}, then it needs to reset Dynamic PUF or switch to use a different Dynamic PUF before T_{max}.

III. HESP Scheme Overview

The goal of HESP is to protect software from malicious modifications and to bind the execution of the software to a specific device that it was compiled for.

To achieve both goals, HESP extends the to-be-protected software code in two aspects. First, code for *tamper detection* is used to detect potential malicious behaviors. Second, code for *tamper response* is used to change the software execution if a malicious behavior is detected. Figure 1 shows the main components of the framework used to realize the HESP.

A. Assumptions and Threat Model

We target low-end devices that may not have computation resources or extra hardware for tracking the execution flow of software. We assume devices to contain a Dynamic PUF (e.g., a DRAM PUF) and a trusted interface to access the Dynamic PUF (e.g., a Linux kernel module in our implementation). We further assume the attacker can only query the Dynamic PUF through the trusted PUF interface.

We assume the program to be protected has predictable execution time. The variation of the execution times are assumed to be within the time resolution δ_t of the Dynamic PUFs. For user input or I/O delay with unpredictable response time, the software may be designed such that the Dynamic PUF is reset after each user input or I/O delay, so the time waiting for the input is not monitored by the Dynamic PUF, whereas the execution time of the other parts of the software is monitored by the Dynamic PUF. In this way, the execution of the program (other than the user input) is always protected.

We assume that the self-checksumming code is able to hash the binary code that is actually executed. Thus, kernel-level attacks that manipulate physical and virtual memory mapping to confuse the self-checksumming code are not in scope [23].

We assume the attacker has access to one or more (protected) binaries and instances of devices with the correct Dynamic PUFs that the binaries that were compiled for. In Section IX-B, dynamic attackers who can run and observe the runtime behavior of the code and static attackers who can only inspect the code statically are discussed, respectively.

B. Tamper Detection

The first part of the HESP scheme is tamper detection, as shown in Figure 1 (ii). The goal is to detect whether the software is running on the authorized hardware device and whether the software was modified in any way by an attacker. Dynamic PUF query code and self-checksumming code are used for this purpose.

- **Dynamic PUF Query Code:** Dynamic PUF query code makes a PUF query, and the PUF response will later be used in the tamper response code. The protected software binary is only allowed to execute on an authorized device, and with correct timing. This is achieved by entangling the execution of the software with Dynamic PUF responses. As introduced in Section II, the responses of a Dynamic PUF depend on the device and the time of the query. The HESP-protected software can execute correctly only with correct Dynamic PUF responses, implying that the authorized device is used and the Dynamic PUF is queried at the correct time. Details of using the Dynamic DRAM PUF for HESP are given in Section VII.
- **Self-Checksumming Code:** The integrity of the to-be-protected software (including the patched protection code and HDS) is checked by self-checksumming code, like in [1], [2], [3]. The to-be-protected software is divided into segments. Each self-checksumming code instance computes a linear checksum function over one or more segments (which in turn contain self-checksumming code instances for checking other segments). The self-checksumming code instances thus mutually check each other. If the checksum result matches an expected value, the code segments that were checked are considered unmodified, as will be discussed in Section IV.

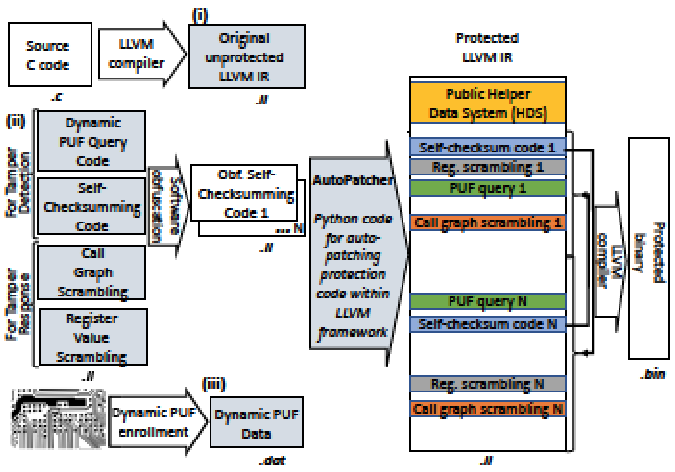

Fig. 1: Overview of the HESP framework for automatically patching the protection code into the software applications. The *.c* represents C code, *.ll* represents LLVM IR code, *.dat* represents auxiliary data such as enrollments from the Dynamic PUF, and *.bin* is the executable.

C. Tamper Response

If the tamper detection code observes a potential malicious behavior (e.g., wrong PUF response or wrong checksum), a tamper response function will then give a wrong result, changing the control flow of the software. We use two different tamper response functions: call graph scrambling and register value scrambling.

- **Call Graph Scrambling:** Call graph scrambling changes the control flow of the software [19]. Especially, we focus on function calls. In a call graph scrambling code instance, the callee function address is computed using the Dynamic PUF response, the checksum value and a reference value: $DestAddr = PUFresponse + Checksum + RefValue$. If the Dynamic PUF response or the checksum value is incorrect, the scrambling code will compute a wrong destination address resulting a jump to an incorrect destination address as callee function, and the program will misbehave and likely eventually crash. An alternate and more complex approach could be to jump to a random function. This would require a re-engineering effort for the software such that each function can potentially call any other function and all functions would have the same parameters [24]. However, this approach may provide more gracious failure behavior.
- **Register Value Scrambling:** Register value scrambling [19] changes the execution of the software. One of the system registers, such as the stack pointer, will be randomly chosen at compile time and the register will be modified at runtime, if a wrong PUF or checksum result is given, likely leading to a program crash.

Note that having a simple "error" function to be called when tamper detection is triggered is not a good solution, because entry points to the error function could give the attacker clues to reverse-engineering and removing the protections.

In this work, we do not consider an attack where the attacker manipulates the PUF responses by intentionally delaying the program to jump to a specific $DestAddr$ in call graph scrambling or write a specific value to a register in register value scrambling, causing a desired malicious behavior. Because the Dynamic PUF only generates a very limited set of responses compared to the whole address space or data value space, even if the attacker has fine-grained control of time and temperature. Thus, only a very limited set of $DestAddr$ or register values can be generated. Further, each device has a unique PUF, so the attack cannot be generalized to a number of devices without modifying the helper data system. Therefore, we do not consider this type of attacks.

D. HESP Framework

An overview of the framework to realize the HESP protection scheme is shown in Figure 1. AutoPatcher takes three inputs: (i) LLVM IR compiled from C source code of user program; (ii) tamper detection code and tamper response code templates to be inserted; and (iii) Dynamic PUF enrollment measurements that were taken on the authorized device that the software will execute on. AutoPatcher creates instances of the protection functions and inserts them as protection code instances into the LLVM IR code.

E. Execution Time of Software and the Time Resolution

Dynamic PUFs can help enforce or manage the control flow of the software. Dynamic PUFs have a time resolution δ_t (discussed in Section II). When the execution time deviates less than δ_t, the PUF response will remain the same, else it will change. Dynamic PUFs with coarse-grained time resolution can be useful when they are applied to programs conducting certain operations periodically at fixed time intervals to limit the time variation of the periodic operation, such as monitoring some states in a factory. In this case, δ_t should be chosen according to the use case. With Dynamic PUFs, if the execution time of the software deviates from the expected time, the execution path will change to prevent further execution.

IV. Self-Checksumming Code

The self-checksumming code uses hash functions to check the integrity of code segments (including the patched protection code and HDS). In order to protect the integrity of software, a dynamic mutual self-checksumming scheme was proposed in [2], [3]. We apply a similar approach here. Each instance of the self-checksumming code calculates a checksum over one part of the code. For example, as shown in Figure 2, the *self-checksum code 1* computes a checksum over the bottom part of the binary, as indicated by the arrow. Once a checksum value is computed, it is combined with a pre-computed reference value and a Dynamic PUF response in order to determine if the value computed at runtime is correct. The reference value is stored in the binary as well. The reference value for one instance of the self-checksumming code will be stored in a part of binary that is checked by another instance of self-checksumming code. Thus, there is a circular dependency among different self-checksumming instances.

Compared to static code integrity checking, our scheme makes use of the circular dependency to improve the tamper resistance. To modify the software, the attacker needs to calculate the hash for all the self-checksumming code instances for the modified software and replace all the existing reference values. Moreover, for a static attacker, who can only analyze the code statically, combined with a software obfuscation scheme that hides the address range covered by each self-checksumming code instance, it is infeasible to compute the checksum and to break the integrity protection. The software obfuscation scheme and how it is applied is discussed in Section IV-B.

A. Linear Self-checksumming Function over a Prime Field

Similar to other software integrity protection schemes for low-end devices [3], we use a linear checksum function. Although cryptographic hash functions could provide better protection, they have a huge performance overhead.[1] A linear hash function can provide sufficient protection for integrity, as shown in [2], [3], [19]. Especially, when combined with software obfuscation that hides the address range of the checksum code, a static attacker cannot compute the result of each checksum function.

The parameter of each checksumming function is the start and end address of the code segment to be protected and a constant multiplier c for the linear checksum function. The linear hash function is defined as follows: For a binary code segment $D = [d_1, d_2, ..., d_n]$, where d_i is the i^{th} 32-bit word in the code segment, the linear checksum function h on D is defined iteratively by $h_0(D) = 0$ and $h_i(D) = d_i + c * h_{i-1}$ for $0 < i \leq n$. Here, multiplier c is a non-zero number. Finally, $h_n(D)$ is the checksum result of code segment D. Equivalently, $h_n(D) = \Sigma_{i=1}^{n} c^{n-i+1} * d_i$.

To make sure that there exist reference values for the self-checksumming functions mutually checking each other, the checksum operation is performed over a finite field. Over a finite field, where division is possible, the checksum function is invertible, i.e., for any i, $d_i = [h_n(D) - \Sigma_{j=1,j\neq i}^{n} c^{n-j+1} * d_j]/c^{n-i+1} \mod q$. Note that each reference value is also a word in some code segment. In this way, equations about reference values can be built, and all the reference values can be solved. Details of building equations to calculate the reference values are given in Section VI-B. In the case of a 32-bit checksum, we chose the prime finite field with a size that is a prime number equal to $q = 2^{32} - 5$ [25].

B. Obfuscating Self-Checksumming Code Instances

To prevent a static attacker from circumventing the checksumming code instances, we obfuscate the self-checksumming code instances, especially the addresses that are used to determine the location and size of the code segment being checked by each instance, by converting static data values embedded in the code to values computed dynamically by functions. As shown in [4], [5], converting static data to procedures can protect data from being located by attackers

[1]In the controlled PUF, a cryptographic hash is used. But in the self-checksumming hash, a simpler hash using a prime filed is used. Because all code segments will be hashed in the self-checksumming, whereas in the controlled PUF only several PUF responses will be hashed. Consequently, a cryptographic hash in a controlled PUF does not affect the runtime of the program significantly, while the use of a cryptographic hash in the self-checksumming code is prohibitive.

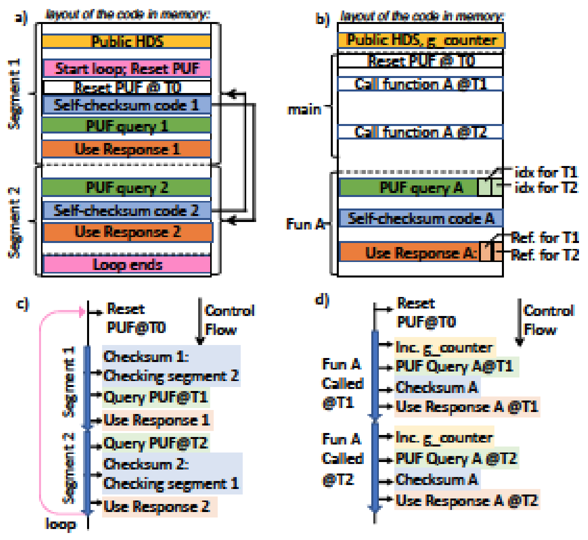

Fig. 2: Example code layout of HESP protected code for a) straight-line code in a loop and b) repeated function calls. Parts c-d) show the control flow of each protection example of a-b).

employing static code analysis. With the use of software obfuscation, a static attacker cannot easily reverse-engineer the address range of the code segment checked by each self-checksumming code instance, thus, he or she cannot compute the self-checksumming result of the modified code, and cannot circumvent the self-checksumming protections.

V. Examples of Deploying HESP

This section shows four code structures that may be found in typical programs, and how HESP is applied to them: first, a straight-line program; second, a program with repeated function calls, where a callee function is called multiple times; third, a program with a function called in a loop; and finally, a program with its control flow depending on time.

A. Protection of Straight-Line Code

The code layout of a simple protected program with two segments is shown in Figure 2 (a, c). At the beginning of the program, the Dynamic PUF is reset. Each segment contains a PUF query and a self-checksumming code. The PUF response depends on the execution time elapsed between the last PUF reset (at the beginning of the program) and the PUF query time. The self-checksumming code computes the checksum of a code segment to check the integrity of another segment. The destination address of the function call at $Use\ Response\ 1$ depends on the PUF response and the checksum, and the program will continue to execute segment 2 if both the PUF response and the checksum are correct.

B. Protection of Program with Repeated Function Calls

In this scenario, some code will be executed more than once, but in a deterministic way, as in Figure 2 (b, d). This could happen when a function is called several times repeatedly, e.g., $func\ A$ is called twice from the main function. The self-checksumming code will always return the same result if it is executed several times. However, since a Dynamic PUF is used, the PUF will return different responses each time the PUF query code in the function is executed (assuming the time between the two PUF queries is larger than the time resolution δ_t), and the PUF queries need to use different idx to the HDS entries at different times.

To make sure the functions will execute as designed if the PUF is queried several times, several idx are needed for PUF queries at different times and several reference values are needed when using the PUF responses. To achieve this, a global counter $g\_counter$ is inserted for each function to point to the correct idx and reference value to be used at runtime. Every time a function is called, the global counter will be increased by one. The PUF query code and response code will fetch the corresponding idx and reference value based on the value of the global counter and continue the correct execution only with a correct PUF value. In Figure 2 (b), the $Use\ Response\ A$ is called twice, and thus, has two reference values, one for each call.

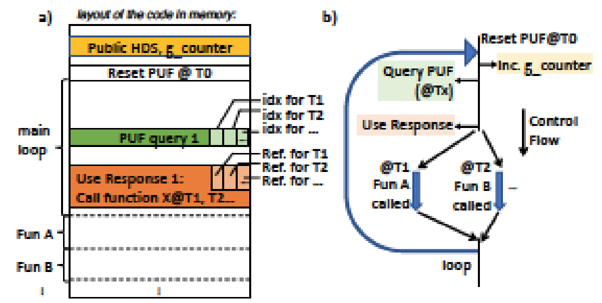

Fig. 3: Protection of program with control flow depending on time.

C. Protection of Program with Loops

Within a loop, the PUF query will be executed several times. The same method as in repeated function calls can be applied if the number of loop iterations is fixed. Another solution is to reset the Dynamic PUF at the beginning of each loop, as shown with the loop in Figure 2 (a, c). Every time the code block in the loop is executed, the PUF response will be the same (relative to when the PUF was reset). In this case, no reference counter value in the code is needed.

D. Program with Time-Dependent Control Flow

As the Dynamic PUF response depends on time, HESP enables program execution to depend on the time elapsed since the last PUF reset. For example, in Figure 3, the destination of a callee function now depends on the time of the function call, e.g., *func A* is called at T1 and *func B* is called at T2. To generate the desired destination address, the PUF responses at the different times are used with different reference values. Similar to the repeated function call, a global counter is inserted and increased at each loop iteration. The destination address is calculated based on the PUF response and the reference value pointed by the global counter. As shown in

Figure 3 (b), in each iteration of the loop, a function will be called. Effectively, a sequence of different functions is executed, e.g., first *func A* at T1 and then *func B* at T2. This technique flattens the call graph of the program, which is a well-known technique for software obfuscation [8].

VI. HESP Framework

In this section, we first introduce the parameters used for deploying HESP and then describe the steps of AutoPatcher inserting protection codes into executable binaries.

A. Security Parameters

HESP is parameterized with the values listed below:

- **Total Number of Code Segments** (S)**:** The to-be-protected software is divided into S segments. One self-checksumming code instance will be inserted into each code segment. Thus, S determines the total number of self-checksumming code instances to be inserted in the protected code. For the same source code, larger S means more checksums to be used in the tamper response functions. If there are more tamper response functions than S, some checksums will be used by multiple tamper response functions.
- **Self-Checksumming Overlap Factor** (C)**:** This parameter indicates how many self-checksumming code instances will check the same segment of the binary, or equivalently, how many code segments will be checked by one self-checksumming code instance. The bigger the overlap factor is, the more self-checksumming code instances the attacker needs to circumvent if he or she wants to modify a code segment. However, a big overlap factor will increase the runtime overhead to calculate the checksums.
- **Number of Tamper Response Code Instances** (R, R_C, R_R)**:** R denotes the total number of tamper response code instances. Each self-checksumming code instance will correspond to one or more tamper response code instances, so $R \geq S$. For each tamper response code instance, there should be one corresponding PUF query. Specifically, R_C indicates the number of call graph scrambling code instances that change the jump destination of a function call. For an attacker who wants to reverse-engineer the call graph, he or she needs to know the destination of each of the R_C tamper response code instances. This number also depends on the complexity of the call graph of the program. In addition, R_R indicates the number of register value scrambling code instances, and $R = R_R + R_C$.
- **Total Number of Dynamic PUF Queries** (P)**:** This parameter indicates the total number of Dynamic PUF queries conducted while executing the software. In the current setting, each tamper response code instance takes one Dynamic PUF response as input. If one tamper response code instance is executed several times, e.g., in a loop, a PUF query will be made in each iteration, then that tamper response code instance corresponds to multiple Dynamic PUF queries, and thus $P \geq R$. To reverse engineer the binary code, the attacker needs to know all Dynamic PUF responses. A large number of PUF queries inserted into the program will increase the attacker's efforts, but will also increase the size of HDS in the patched program and will require more activity on the PUF interface for more PUF responses.

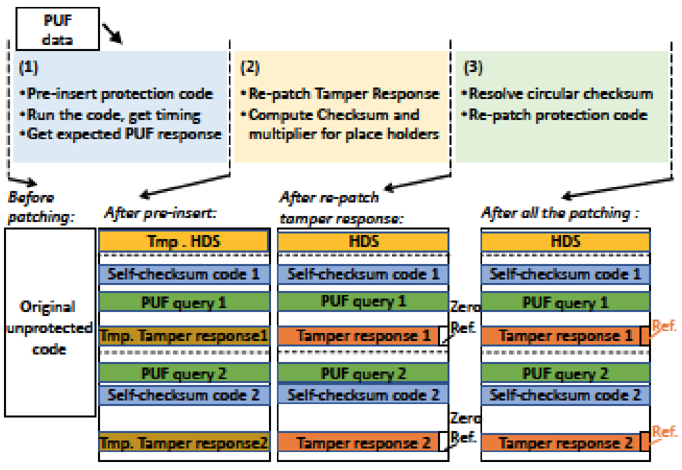
PUF data
(1)
•Pre-insert protection code
•Run the code, get timing
•Get expected PUF response
(2)
•Re-patch Tamper Response
•Compute Checksum and multiplier for place holders
(3)
•Resolve circular checksum
•Re-patch protection code
Before patching:
After pre-insert:
After re-patch tamper response:
After all the patching :
Original unprotected code
Tmp . HDS
Self-checksum code 1
PUF query 1
Tmp. Tamper response1
PUF query 2
Self-checksum code 2
Tmp. Tamper response2
HDS
Tamper response 1
Tamper response 2
Zero Ref.
Ref.

Fig. 4: Frameworks to automatically generate protected binary code.

B. Steps to Insert Protection Code

Due to the circular dependency among the self-checksumming code instances, and the initially unknown timing of the PUF queries, the protection code needs to be inserted in the following three steps, as shown in Figure 4:

1) The whole code binary is first divided into S segments. Depending on C, each checksumming code instance is assigned randomly to check C code segments, such that each segment will be checked by C self-checksumming code instances. Also, for each self checksumming function the non-zero multiplier c is chosen randomly. The Dynamic PUF queries and the self-checksumming code instances can be inserted to any place in the LLVM IR in the segment, and the tamper response function should be inserted such that it will execute after a self-checksumming code instance or a Dynamic PUF query. Especially, there should be enough time between Dynamic PUF query code and tamper response code for the Dynamic PUF interface to return the PUF response. In our implementation, we insert the Dynamic PUF query code right after the last tamper response code to ensure enough time to obtain the Dynamic PUF response for the tamper response code.

   Depending on the source code, different tamper responses can be inserted. For example, if there is a function call, a call graph scrambling code instance can be inserted, i.e., the destination of the callee function is tied to the checksum and Dynamic PUF response in the tamper response. Otherwise, a register value scrambling code instance can be inserted. With the above decision of where and which protection code instances to insert, the tamper detection code instances and the temporary tamper response code instances are created from the template and are inserted. The temporary tamper response code is the same as the real response code, except that the temporary tamper response code will always give the correct response even if the tamper detection code returns a wrong value. If there are repeated function calls or there is time-dependent control flow, global counters are also inserted (as discussed in Sections V-B and V-D).

Next, the pre-inserted code is compiled and executed, and the time when each PUF query happens is recorded by a kernel module monitoring the PUF queries. Now knowing the Dynamic PUF query times, PUF enrollments for the required delay times are conducted. With the enrollments, a set of HDS can be generated and inserted into the data segment of the binary. The PUF enrollments could also be done in advance, but then, all possible PUF query times need to be enrolled.

2) The code is re-patched with the real tamper response code instead of the temporary one. The reference values in the tamper response code are not known yet and are set to zero as placeholders.

After re-patching, the LLVM IR is compiled into binary, then the expected checksum of the code segments that each self-checksumming code instance checks are computed. Let D_i be the code segment to be checked by self-checksumming function i, then the checksum is computed by $H_{i,0} = h_{n_i}(D_i)$, where n_i is the size of D_i; note that the reference value placeholders in the tamper response code p_j are now treated as zeros. Moreover, the multiplier of the placeholder p_j in the code segment D_i is computed by $l_{i,j} = c^{n_i - k_{i,j}+1}$, where $k_{i,j}$ is the distance of the placeholder p_j from the beginning of the code segment D_i, and c is the multiplier of the checksum function. So the final checksum will be $h_{n_i}(D_i) = H_{i,0} + \Sigma_j l_{i,j} * p_j \mod q$, where p_j are the values of the placeholders to be solved in Step (3).

3) To calculate all the reference values in the tamper response code instances, a system of linear equations on the $q = 2^{32} - 5$ prime field is constructed. For example, in call graph scrambling codes, the destination of a jump is the sum of the checksum i, the PUF response i, and the reference value. Thus, the reference value p_i satisfies the following equation: $p_i = DestAddr - PUFresponse_i - Checksum_i = DestAddr - PUFresponse_i - (H_{i,0} + \Sigma_j l_{i,j} * p_j)$. Here, the p_j are the placeholders in the code segments D_i checked by the checksumming function i whose result is used by p_i. In this way, a system of linear equations on the placeholders is constructed and solved. A solution to the equations must exist on the prime field.

To patch a program with time-dependent execution (discussed in Section V-D), in addition to the original program, a separate file should be given as an input indicating the time-dependent control flow, i.e., the $DestAddr$ for each loop iteration. The compiler will generate the reference values accordingly.

Finally, the correct reference values are inserted into the placeholders in the response function in the binary.

C. Code Obfuscation

As discussed in Section IV-B, we convert static data to procedures to hide the address range of the code segment being checked by each self-checksumming function instance. In the evaluation, we keep the obfuscation scheme separate from AutoPatcher. There are open-source frameworks that can be used for automatic obfuscating programs [10]. Note that we only obfuscate the self-checksumming code, which is a small portion of the code. Thus, the obfuscation results in a small runtime overhead compared to the runtime of the whole program.

D. Dynamic PUF Interface

The Dynamic PUF query code assumes a Dynamic PUF interface on the platform. In this work, we use a controlled PUF interface (Section VII-E) that can be implemented in a Linux kernel module. The kernel module and application can communicate through fixed memory locations in the memory space of the application. The addresses of the memory locations are public. For example, the application writes to a pre-defined address to request a PUF response. The kernel module will monitor the address. If a new request is detected, the response will be written to another pre-defined address, so that the application can obtain the PUF response.

VII. DRAM PUFs as Dynamic PUFs

In this paper, we use DRAM modules to realize Dynamic PUFs. DRAM, which is found in many commodity IoT and embedded platforms, has been shown to exhibit PUF behavior. There are three types of DRAM PUFs: decay-based DRAM PUFs [26], [27], [17], [28], [29], [22], latency-based DRAM PUFs [18], and startup-based DRAM PUFs [30]. The PUF responses of the first two depend on the timing of the PUF queries, and we show that these two types of PUFs can be used as Dynamic PUFs.

A. Threat Model

A trusted Dynamic PUF interface is assumed. We implement the Dynamic PUF interface as a kernel module on commodity, off-the-shelf devices, thus software attacks on the kernel module or DRAM memory are not in scope. Consequently, this implementation does not protect against invasive physical attacks such as probing the hardware buses or memory. A Dynamic PUF realized fully in hardware or firmware could relax the assumption of a trusted kernel module – but cannot be directly deployed on commodity, off-the-shelf devices today. A trusted PUF enrollment environment with high bandwidth communication to the device is assumed, such as at the manufacturing site.

B. Dynamic PUFs using Decay-based DRAM PUFs

The decay-based DRAM PUF leverages the fact that DRAM cells lose data over time, which is also known as *DRAM decay*. Without refresh, each DRAM cell can only retain the data for a certain time, called *retention time*. Decay-based DRAM PUF responses are based on variations of the retention times of the DRAM cells. A decay-based DRAM PUF is composed of a set of cells within a DRAM region. The retention time of each DRAM cell is shown to be random and related to fabrication variations [17], [28].

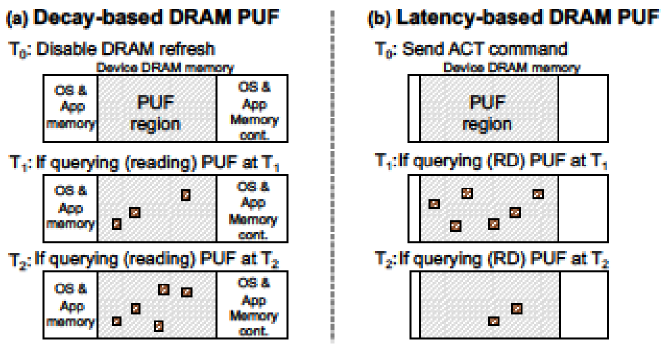
(a) Decay-based DRAM PUF
T0: Disable DRAM refresh
Device DRAM memory
OS & App memory
PUF region
OS & App Memory cont.
T1: If querying (reading) PUF at T1
T2: If querying (reading) PUF at T2
(b) Latency-based DRAM PUF
T0: Send ACT command
T1: If querying (RD) PUF at T1
T2: If querying (RD) PUF at T2

Fig. 5: Time-dependency of the responses of (a) decay-based DRAM PUF and (b) latency-based DRAM PUF: for both the response depends on the query time since Dynamic PUF Reset at T_0.

In Dynamic DRAM PUFs, the PUF challenge is the address range of a *DRAM PUF region* from which the PUF will be extracted. Upon a *Dynamic PUF Reset*, an *initial value* is written to the DRAM PUF region and the DRAM refresh of the region is disabled. Upon a *Dynamic PUF Query*, the DRAM PUF region will be read and the value is compared to the initial value to get the locations of bit flips in the DRAM region, which is the PUF response. Different initial values[2] are tested during enrollment to find cells and initial values that will give enough bit flips in the DRAM PUF region.

As illustrated in Figure 5 (a), querying the decay-based DRAM at different times (e.g., at T_1 and T_2) will result in different PUF responses. Here, we also use *decay time* to denote the time between the *Dynamic PUF Reset* and the *Dynamic PUF Query*. Due to the decay properties of the DRAM cells, the time resolution of the dynamic decay-based DRAM PUF can be in the order of a second.

However, DRAM has the additional property that each access to a DRAM cell (a read or a write) implicitly refreshes (*resets*) the whole DRAM row. When one wants to query the PUF region several times after each reset, the PUF region can be subdivided into sub-regions. Each sub-region consists of a set of DRAM rows, and these DRAM rows do not have to be adjacent to each other. Upon a *Dynamic PUF Reset* request, all DRAM PUF sub-regions have their initial values set and refresh is disabled. To query the Dynamic PUF, an index (*idx*) is provided as part of the *Dynamic PUF Query*. The index also determines which sub-region should be used for that query. Other sub-regions continue to decay.

A decay-based Dynamic DRAM PUF can be accessed *at system runtime*. It can be implemented on a commercial, off-the-shelf device by selectively refreshing DRAM as shown in Figure 5 (a), following [17]. First, on a Dynamic PUF Reset (T_0), the DRAM region is reserved, so that neither the operating system (OS) nor applications use it, its cells are initialized with the initial value (e.g., all zeros), and the refresh of the DRAM module is disabled. To allow the OS and other applications use DRAM without decay, a customized kernel module is needed to selectively refresh the address space in which the OS and applications reside by accessing each DRAM row in the address space[3], as was shown in [17]. In this way, the system can still run even when the refresh of the whole DRAM module is disabled. Upon a PUF query the content of the DRAM region is read as the raw PUF response.

C. Dynamic PUFs using Latency-based DRAM PUFs

The latency-based DRAM PUF leverages the fact that reducing the DRAM read access latency can induce bit flips in the DRAM, and the location of the bit flips are shown to be unique and stable. For example, as shown in [18], the active time (t_{RCD}) between a DRAM row is activated (using the ACT command) and the subsequent read or write request (using the RD or WR command) can be manipulated to cause bit flips. In today's DRAM modules, the time resolution of the latency-based Dynamic DRAM PUF is in the order of nanoseconds. Similar to the decay-based DRAM PUF, the access to the DRAM is in the granularity of each DRAM row. To allow multiple PUF queries after one PUF reset, the PUF region can be subdivided into sub-regions.

To access the latency-based DRAM PUF as a dynamic latency-based DRAM PUF *at runtime*, the *Dynamic PUF Reset* can be realized by first writing the initial value to the PUF region and then sending the ACT command to activate the PUF row in the DRAM. Then, the *Dynamic PUF Query* is realized by an RD command that is sent to read the value in that DRAM row, and the result is compared with the initial value. The raw PUF response is the location of all the bit flips (Figure 5 (b)). This, however, requires low-level control of the DRAM timing parameters and cannot be realized on commodity, off-the-shelf devices today, but could be realized on FPGAs or with hardware changes.

D. HDS and ECC for Dynamic DRAM PUFs

The PUF responses of both the decay-based and the latency-based DRAM PUFs consist of bit flips in DRAM regions. For different query times T_x, a different set of bits will flip. Decay-based DRAM PUFs yield more bit flips for longer decay times while latency-based DRAM PUFs yield more bit flips for shorter latencies. In the following, we show a design of a HDS for decay-based DRAM PUF. For latency-based DRAM PUFs, we simply reverse the logical 0 and 1, and the HDS will work in the same way.

The HDS for decay-based Dynamic DRAM PUFs consists of multiple sets of pointers to cells in the DRAM PUF region. Each set is used for obtaining the PUF response for one specific query time T_x, and consists of pointers to cells that are expected to flip in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ if a logical 1 is to be stored in the PUF response, and pointers to cells in the time interval $[T_x + \delta_t, T_x + 2\delta_t]$ if a logical 0 is to be stored in the PUF response. Considering noise, most of the cells that flip in $[T_x - 2\delta_t, T_x - \delta_t]$ during enrollment

[2]If the initial value does not put a cell into a charged state, then in the enrollment phase, the cell will not decay and will not be chosen to be in the HDS. These cells will not be used later for the final PUF response.

[3]The refresh can be controlled by the kernel module and done at the same rate as in the DRAM standard, but this may use up significant CPU resources. If the system and application memory is refreshed at a reduced rate to save CPU resources, the system will be more vulnerable to the Rowhammer attacks [31].

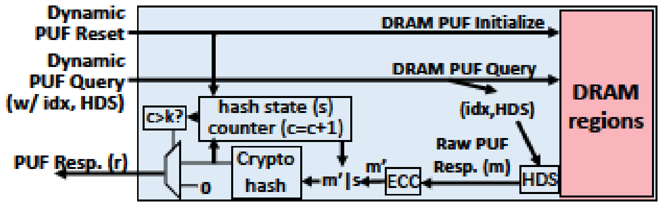

Fig. 6: Controlled PUF interface to the Dynamic PUF. As shown by the multiplexer, The first k queries will not return a hashed result, but give a zero output.

will flip in response at T_x and most of the cells that flip in $[T_x + \delta_t, T_x + 2\delta_t]$ during enrollment will not yet flip in the response at T_x. If the measurement time deviates from T_x by more than δ_t, the decay result will be different.

Since the retention times of cells are independent of each other, with this HDS structure, responses at T_x and T_y are generated from different cells, and thus, are independent.

To correct any noise present in the PUF responses, an Error Correction Code (ECC) should be used over the raw PUF response m to generate a stable response m'. In our implementation, we use 3-repetition codes for error correction, where 3 bits encode 1 bit. The number of bits of the PUF response m' used in HESP is a security parameter. If we choose a 32-bit security level in the HESP scheme in the tamper response code to bind the 32-bit address, $3 \times 32 = 96$ bits are needed in raw PUF response m. For 3-repetition code, with BER of 1.5%, the probability of an error in a 32-bit response will be 2%. For greater BER, more bits are required for error correction. As shown in [32], using 32 bits to correct 1 bit, up to 25% error in the response can be corrected reliably with a failure rate of less than 10^{-9}. Other ECCs (e.g., BCH codes [33], [34]) can be used instead of the repetition code for more efficient error correction.

During the enrollments, for each T_x, DRAM PUF responses at decay times $T_x{-}2\delta_t$, $T_x{-}\delta_t$, $T_x{+}\delta_t$, and $T_x{+}2\delta_t$ are measured. To generate the HDS for T_x, first a random 32-bit binary string is generated and encoded by the ECC (e.g., 3-repetition codes). With the encoded 96-bit binary string, cells that flip in $[T_x{-}2\delta_t, T_x{-}\delta_t]$ for 1 or $[T_x{+}\delta_t, T_x{+}2\delta_t]$ for 0 are chosen to generate the helper data for time T_x. The cells used for T_x in HDS should not be in the sub-regions that are already used for other query times. During the reconstruction, the cells pointed to by the HDS are read, the results are concatenated into a binary string m, and the binary string is decoded by the ECC to obtain the final 32-bit response m', as shown in Figure 6.

E. A Controlled PUF Interface to Dynamic DRAM PUF

A Dynamic DRAM PUF is built using a DRAM region. Because a DRAM PUF (both decay-based and latency-based) is a weak PUF, it can only provide a limited number of CRPs. If the attacker can directly access the PUF, he or she is then able to brute force all PUF challenge-response pairs for each decay time. This allows the attacker to obtain a logical clone of the PUF. To counter this, a controlled PUF environment is needed to prevent the attacker from easily brute-forcing all the challenge-response pairs [35], as discussed in the threat model in Section VII-A.

A controlled PUF interface to the Dynamic PUF is shown in Figure 6. Here, the PUF is queried by a trusted component, and only the cryptographically hashed result is passed to the outside. The controlled PUF environment supports two types of PUF queries: a *Dynamic PUF Reset* and a *Dynamic PUF Query*. A *Dynamic PUF Reset* will initialize the PUF, restart the decay process of all DRAM cells in the DRAM region, and reset the cumulative hash state s and the counter c of the controlled PUF environment. During a *Dynamic PUF Query*, the trusted module reads out the raw response m according to helper data H_{idx}, corrects any errors in the raw PUF response as discussed in Section VII-D, and obtains a noise-free PUF response m'. Then, the PUF response m' is concatenated with the cumulative hash state s and is hashed to get the final response r. The resulting r is subsequently stored as the cumulative hash state s for the next query. Consequently, no raw PUF response can be directly accessed.

Now consider an attacker who has access to the PUF interface. To hide the mapping between T_x and idx from the attacker, a random permutation of the entries of the HDS is applied when generating the HDS. In this way, even if the attacker knows idx of a PUF query, he or she still needs to find out the time since the Dynamic PUF reset to obtain the correct PUF response. Since querying the PUF with idx at a wrong time might result in all zero (if queried before $T_x{-}2\delta_t$) or all one results (if queried after $T_x + 2\delta_t$), which will result in two fixed hash results, the attacker can try to reset and query the PUF once at different times with each of the idx to obtain the correct timing of each H_{idx}. To prevent this, the first k queries will not return a hashed result, but give a null output (shown by the multiplexer in Figure 6). The attacker needs at least k PUF queries to get the first output from the controlled PUF. In this way, the attacker needs to brute force k PUF query times to get one response.

VIII. Evaluation

This section evaluates the performance of HESP. We first discuss the implementation of AutoPatcher and show the sample programs tested. Next, we present the overhead of the protection scheme. We end with an evaluation of the characteristics of Dynamic DRAM PUF.

A. AutoPatcher

To evaluate the protection system, AutoPatcher is implemented in Python. Currently, AutoPatcher inserts one self-checksumming code instance into each function of the program, and thus, S equals the total number of functions in the program. For each function call, a call graph scrambling code instance will be inserted, thus, R_C equals the total number of function calls. Otherwise, if a function does not call any other function, a register value scrambling code instance will be inserted, and thus, R_R equals the total number of functions that do not call any other function. In this way, AutoPatcher analyzes the call graph, divides the program into S segments and pre-inserts the protection code instances. To

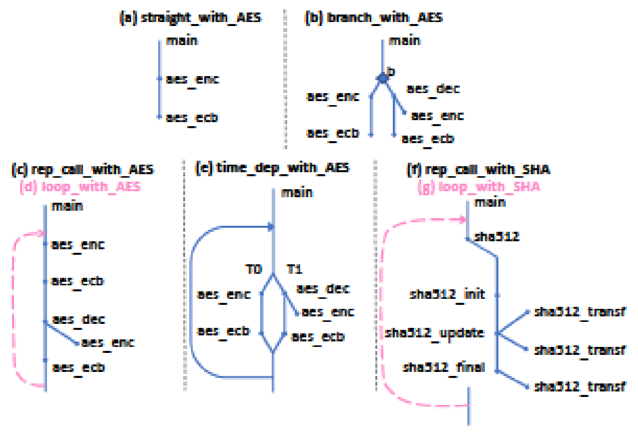

Fig. 7: (a) - (f) show the control flow graphs of the different programs evaluated. For (d) and (g), the loop is shown with the pink dashed lines in the figure.

implement the time-dependent control flow, AutoPatcher takes the original program and an extra file indicating the time-dependent call graph to patch the tamper response functions. We also use a separate Python script to generate the HDS and to determine the expected PUF responses from PUF enrollment measurements. With the expected PUF responses and the LLVM IR code of the software program, AutoPatcher can resolve the reference values as mentioned in Step 2 and Step 3 of Section VI-B.

The core part of AutoPatcher consists of about 1400 lines of Python code. The rest of the code contains the template protection code in LLVM IR (shown in the gray boxes (ii) in Figure 1). To patch and compile the program *rep_call_with_AES* (Figure 7) on an Intel i5-4570 CPU and 8 GiB memory takes in total about 2.6 seconds (including about 1.3 seconds to solve the equations due to circular checksumming). Compiling the plain unprotected software takes about 0.2 seconds. The main patching time is from solving the equations on the prime field, which depends on the security parameter R.

B. Sample Programs Used in Evaluation

For the sample programs we developed to test the HESP, we consider a case where an IoT device measures a sensor and sends out the data periodically. To protect the obtained data, the program performs encryption, or compute a digital signature at fixed time intervals. HESP ensures that the cryptographic operations are executed at the desired time, otherwise, any deviation will be detected by the Dynamic PUF response. To demonstrate that our AutoPatcher can patch different programs, we use open-source AES[4] and SHA[5] C implementations in our sample programs with different control flow structures introduced in Section V. The programs are shown in Figure 7 and listed below. For each program, we also list the security parameters S, R and P used for that program.

[4]FIPS-197 compliant AES implementation: https://github.com/particle-iot/core-communication-lib/tree/master/lib/tropicssl/library

[5]FIPS 180-2 SHA implementation: https://github.com/ogay/sha2

(a) *straight_with_AES* ($S = 3$, $R = 4$, $P = 4$): The program encrypts a data block. The main function calls two functions (*aes_enc* and *aes_ecb*) sequentially to perform AES encryption, similar to the straight-line example in Section V-A. As discussed in Section VIII-A, AutoPatcher assigns parameters depending on the structure of the program. Because there are three functions (*main*, *aes_enc*, and *aes_ecb*), $S = 3$. There are two function calls in the main functions, which are patched by the call graph scrambling code, thus, $R_C = 2$. Inside *aes_enc* and *aes_ecb*, there is no function call, thus, the register value scrambling code instances are inserted in the two functions, with $R_R = 2$. $R = R_R + R_C = 4$. A PUF query is made for each of the tamper response code, thus, $P = R = 4$.

(b) *branch_with_AES* ($S = 4$, $R = 7$, $P = 9$): Depending on the input command, the program either encrypts or decrypts a data block. As shown in Figure 7 (b), in this implementation, the *aes_dec* function calls the *aes_enc* function due to the shared operations in both encryption and decryption process. There are four functions (*main*, *aes_enc*, *aes_dec*, and *aes_ecb*), thus, $S = 4$. Since there are four function calls in the main functions, and one *aes_dec*, which are patched by the call graph scrambling code, thus, $R_C = 5$. Inside *aes_enc* and *aes_ecb*, there are no other function calls, thus, register value scrambling code instances are inserted in the two functions, with $R_R = 2$. $R = R_R + R_C = 7$. A PUF query is made for each of the tamper response code in each branch, because different branches have different timing. For the left encryption branch, there are four tamper response code instances (including two call graph scrambling instances and two register value scrambling instances); for the right decryption branch, there are five tamper response code instances (including three call graph scrambling instances and two register value scrambling instances). Thus, $P = 9$.

(c) *rep_call_with_AES* ($S = 4$, $R = 7$, $P = 9$): The program first encrypts and then decrypts a data block. The *aes_enc* function is called twice repeatedly in total, once by *main* and once by *aes_dec*. The *aes_ecb* function is called twice by *main*. This is similar to the example in Section V-B, where a function is called repeatedly. Similar to the test program branch_with_AES, there are four functions, thus, $S = 4$. There are four function calls in the main functions, and one *aes_dec*, which are patched by the call graph scrambling code, thus, $R_C = 5$. Inside *aes_enc* and *aes_ecb*, there is no other function calls, thus, register value scrambling code instances are inserted in the two functions, with $R_R = 2$. $R = R_R + R_C = 7$. A PUF query is made for each execution of the tamper response code instances as discussed in Section V-B. *aes_enc* and *aes_ecb* execute twice, and thus, two PUF queries are required for the register value scrambling instance inside each of them. Thus, $P = 9$.

(d) *loop_with_AES* ($S = 4$, $R = 7$, $P = 9$): There is a loop with 5 iterations outside *rep_call_with_AES*. At the beginning of the loop, the PUF and all the counters are reset. This is similar to the example in Section V-C. Here, we reset the dynamic PUF at the beginning of each

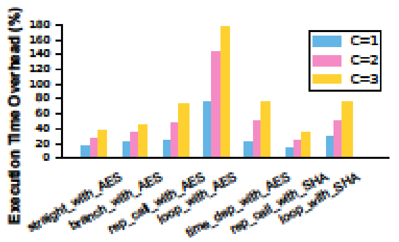

Fig. 8: Runtime overhead for different overlap factors C of the self-checksuming code.

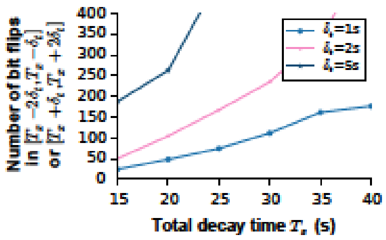

Fig. 9: Number of bit flips in Dynamic DRAM PUFs with DRAM PUF size of 8 MiB.

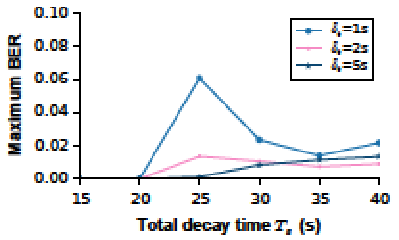

Fig. 10: Bit Error Rate of Dynamic DRAM PUFs with DRAM PUF size of 8 MiB.

loop iteration. The other protection code is identical to *rep_call_with_AES* without loop. (The S, R, P are the same as that for *rep_call_with_AES*.)

(e) *time_dep_with_AES* ($S = 4$, $R = 5$, $P = 9$): The program contains 2 loop iterations. In the first iteration, a data block is encrypted, and in the second iteration, a data block is decrypted. The functionality is the same as rep_call_with_AES. In the loop, whether to encrypt or to decrypt depends on the PUF response, which depends on time. This is an example of time-dependent control flow in Section V-D. Similarly, there are four functions in the program, thus, $S = 4$. In this program with time-dependent control flow, there are two function calls in the main function, the first one calls *aes_enc* or *aes_dec* depends on the PUF response, and the second calls *aes_ecb*. Therefore, within *aes_dec*, *aes_enc* is called. Thus, $R_C = 3$. Inside *aes_enc* and *aes_ecb*, the register value scrambling code is inserted, with $R_R = 2$, resulting in $R = R_R + R_C = 5$. For each execution of tamper response code, a PUF response is required. In the first loop iteration (at T0), the left branch with four tamper response functions will be executed, and then (at T1), the right branch with five tamper response functions will be executed. Thus, $P = 9$ PUF queries will be used.

(f) *rep_call_with_SHA* ($S = 6$, $R = 9$, $P = 11$): a SHA 512 digest of a data block is computed. In the main function, only *sha512* is called. *sha512* calls three other functions. The *shar512_transf* function is called three times repeatedly from different functions. The method of choosing parameter S, R, P is similar to the AES application.

(g) *loop_with_SHA* ($S = 6$, $R = 9$, $P = 11$): There is a loop outside *rep_call_with_SHA*. At the beginning of the loop, the PUF and all the counters are reset.

C. Running Software Protected by HESP

We patched the above sample software with AutoPatcher to bind the execution to three different Intel Galileo Gen 2 boards [36]. The protected software execute successfully, getting the excepted output on the correct device.

We observe mis-behavior of the execution if we changed the executable binary or run the software on a difference device than it was compiled for. In such case, the program gave wrong (or missing) outputs or crashed at some point.

D. Overhead of HESP Framework

HESP will increase the size of the executable binaries and also increase the execution time.

1) Execution Time Overhead:

Controlled PUF Interface: In the setup, the controlled PUF is implemented in a Linux kernel module (Section VI-D). If the platform supports parallel computation (e.g., between CPU and co-processor or hyperthreading), then the computation of the PUF interface can be done in parallel with the software. If the software has some idle time, such as in our implementation, the computation of the PUF interface can be conducted when the software is idle. Otherwise, the interface increases the software runtime.

Protected Software: The overhead for different overlap factors of the self-checksumming code is shown in Figure 8. Here, to make a comparison with the original program, we separate the time waiting for the DRAM PUF and the program execution time. Due to the randomness in assigning code segments to self-checksumming functions, we patch each program by running AutoPatcher six times to get six versions of protected binary. Each protected and unprotected binary is executed and timed ten times on the Intel Galileo Gen 2 to measure the CPU cycles. The overhead is computed by the average execution time of all the protected binaries minus the average execution time of the original binaries. Thus, the overhead contains the overhead due to self-checksumming code instances, PUF queries, and tamper response functions. The latter two have small overhead compared to the checksumming function. As shown in Figure 8, bigger overlap factors C will result in bigger overhead. When $C = 2$, the protected code of *rep_call_with_AES* has 48% runtime overhead. The test programs with loops show higher overhead, due to more self-checksum instances at runtime. The overhead also depends on the code size, as the self-checksum code compute hashes over the whole code. The unprotected *rep_call_with_AES* and *rep_call_with_SHA* have similar runtime, but due to the smaller code size of *rep_call_with_SHA*, it has smaller runtime overhead in the self-checksum function, as shown in Figure 8.

2) Executable Binary Size Overhead:

Helper Data System: To extract unique response for each query time and correct errors in PUF responses, HDS is needed (see Section VII-D). For each PUF query, the HDS requires 384 Bytes (4 Bytes for each pointer, and with 3-repetition code, 3×32 cells are needed). For *rep_call_with_AES* where

$P = 9$, the HDS takes up 3.4 KiB. For *rep_call_with_SHA* application where $P = 11$, the HDS takes up 4.1 KiB.

Patched Protection Code: The protection code instances are inserted into the executable binary. The size of the unprotected *rep_call_with_AES* executable with $S = 4$ is 16.6 KiB, while the protected binary is 2 KiB larger. With HDS ($P = 9$) of 3.4 KiB, the total overhead is 32.5%. The size of the unprotected *rep_call_with_SHA* executable with $S = 6$ is 12.8 KiB, and the protected binary is 2.8 KiB larger. With HDS ($P = 11$) of 4.1 KiB, the total overhead is 54%. Other protection schemes also shown similar overhead [19].

E. Dynamic DRAM PUF Characteristics

Prior works have shown the uniqueness of DRAM PUFs [17], [28]. Because for each query time the HDS uses different cells for the PUF response, the uniqueness of DRAM PUF in prior work implies the uniqueness of the Dynamic DRAM PUF responses for different query times. Possible correlations between different DRAM cells in the DRAM PUF responses remain to be further investigated in the future work.

To evaluate the robustness of Dynamic DRAM PUFs, measurements were conducted on Intel Galileo Gen 2 platforms [36] with two 128 MiB DDR3 and a temperature feedback loop such that the temperature of DRAM chips are stabilized at 40°C. We evaluated total decay times $\mathcal{T}_x = \{15, 20, 25, 30, 35, 40\}$ seconds and time resolutions $\delta_t = \{1, 2, 5\}$ seconds. Here, T_x indicates the total decay time, i.e., the time elapsed between the last PUF reset and the PUF query. Two enrollment measurements were taken at $T_x - 2\delta_t$, $T_x - \delta_t$, $T_x + \delta_t$ and $T_x + 2\delta_t$ separately, and cells that flip in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ and $[T_x + \delta_t, T_x + 2\delta_t]$ are enrolled in HDS (see Section VII-D).

Figure 9 shows the average number of bit flips in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ or $[T_x + \delta_t, T_x + 2\delta_t]$. Figure 9 shows that longer T_x or larger δ_t give more bits that can be used in the DRAM PUF region. To generate the HDS for time T_x, 96 bit flips are needed in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ and $[T_x + \delta_t, T_x + 2\delta_t]$, as discussed in Section VII-D. With 8 MiB DRAM PUF size, when the total decay time is larger than 25 seconds for time resolution $\delta_t = 2$ seconds, it can provide more than 96 bits. This means that T_{min} for 8 MiB DRAM PUF on Galileo Gen 2 is 25 seconds. For T_{max}, in the current DRAM PUF construction, the PUF responses come from the new bit flips in a time interval of δ_t (in $[T_x - 2\delta_t, T_x - \delta_t]$ or $[T_x + \delta_t, T_x + 2\delta_t]$), and enough bit flips in the intervals are always observed in the decay time tested. Thus, T_{max} is beyond the decay times we tested. To achieve a better time resolution (smaller δ_t) or a smaller T_{min}, a larger DRAM size should be used. Here, we reserved a big chunk of DRAM PUF region (e.g., 8 MiB), which can provide enough PUF query times. For different PUF query times, different rows in the same DRAM PUF region are used.

To evaluate the error rate of DRAM PUF responses, we took 100 measurements at each decay time T_x. To compute the average bit error rate (BER), with the enrollment, we divide the number of cells that do not flip to their desired logical value (logical one for $[T_x - 2\delta_t, T_x - \delta_t]$ and logical zero for $[T_x + \delta_t, T_x + 2\delta_t]$) by the total number of cells enrolled in the time interval. Figure 10 shows the maximum BER in all reconstructions to show the worst case. When the time resolution δ_t is larger than 2 seconds, the BER is smaller than 1.5%. Figure 10 also shows that the resolution of decay-based DRAM PUF is better than 1 second, but with greater BER, where more efficient ECC or more raw PUF bits are required.

Temperature Effects: The DRAM decay depends on temperature. We tested the time resolution when the temperature of the DRAM chip is stabilized at 40°C. Our setup controls the temperature variation to be within ±0.5°C, which gives the above time resolution. Without temperature control, there will be more noise due to temperature fluctuations and the time resolution will be degraded.

IX. Security Analysis

The presented scheme aims to prevent an attacker from modifying the executable and un-binding the execution of the code from the specific hardware device instance it was compiled for. The attacker aims to defeat one or both of these protections. The objective of HESP is to make the attacks as difficult as possible, given the limitations of low-end IoT-type devices. Especially, an attacker with enough money or time resources can reverse-engineer any software if the software is not protected by cryptographically strong obfuscation [4], [5], [37], [38]. As we discuss below, this work raises the bar for the attacker significantly over the existing software-only solutions, e.g., the use of self-checking codes or existing hardware-software solutions that use SRAM PUFs [19] to bind software execution to a device instance.

A. Attackers without Access to the Correct Dynamic PUF

Without access to the authorized device and the correct Dynamic PUF, the attacker cannot successfully run the program, as correct Dynamic PUF responses are needed to continue the correct execution. An attacker can only statically analyze the code or can run code on a different device or a emulator and guess the Dynamic PUF responses. Without access to correct Dynamic PUF responses, the execution call graph is not known. Consequently, an attacker without access to the authorized device and correct Dynamic PUF cannot run the original or a modified version of the software successfully.

B. Attackers with Access to the Correct Dynamic PUF

A more powerful attacker has access to the authorized device and can query the controlled Dynamic PUF. The attacker may also run the protected software on the device it was originally compiled for, or run any other code that can query the Dynamic PUF at times chosen by the attacker. The attacker can also read and modify the software binary, including the hard-coded data such as the HDS. In the following, we consider two different attackers, namely (1) static attackers who can only statically inspect the binary, and (2) dynamic attackers employing tools like emulators and debuggers.

1) Static Attackers:

A static attacker cannot run the program or observe the behavior of the program. The attacker can only statically inspect the code and then use a separate program to query the Dynamic PUF. However, the attacker needs to know the timing of each PUF query, whereas the attacker does not know the call graph of the software to estimate the execution time. The attacker can only brute force the P Dynamic PUF query times one by one. Moreover, due to the obfuscation of the self-checksumming code instances, the static attacker cannot circumvent the self-checksumming protection (in Section IV-B).

2) Dynamic Attackers:

Emulators: Emulators can simulate the behavior of hardware platforms and "execute" the binary. More importantly, while most emulators do not model the timing of software, a cycle-accurate emulator can give an estimation of the timing of PUF queries. Recall that as the PUF responses depend on time, the attacker needs to get PUF query timing from the emulator and then query the PUF. The attacker has to do this for each query back and forth until the end of the program. Increasing the number of PUF queries P makes the attack more difficult.

Debuggers: One powerful tool an dynamic attacker will use is a debugger. However, there are several limitations: First, we only distribute the final binaries. So the attacker does not have binaries compiled with debug flags. Second, the debugger sometimes changes the binary, for example, when a breakpoint is inserted. This will be detected by the self-checksum code and will trigger the response function leading to a wrong execution. Third, a debugger will change the timing of execution (e.g., when a breakpoint is inserted). Since the Dynamic PUF responses depend on time, this will lead to incorrect PUF responses and thus incorrect execution. Even with this limitation, the attacker may be able to use watchpoints to figure out each of the P PUF queries and all the S self-checksumming code instances, and then all the R response functions. Increasing S, C, P, and R will make the attack more difficult. Furthermore, a number of anti-debugging techniques have been proposed [39], [40] and could be additionally used by our protection scheme.

Tracing: Tracing involves logging the execution traces of software, such as the memory contents, the register values, and the execution path. The attacker then can trace back the execution. With tracing and taint analysis based on the execution trace, it is reported that it is possible to identify the self-checksumming code or reverse engineer the call graph of an application [37], [38]. The complexity of the attack is similar to the case of using a debugger.

Profiling: Profiling is another tool to analyze the software execution involving performance counters. The protection code inserted does not consume significant time nor does it make excessive memory accesses, so it is hard to use profiling to locate the protection code. Even if profiling infers the locations of the protection code, it does not reveal the PUF responses. Thus, profiling is not as powerful as watchpoints in debuggers, whose attack complexity depends on S, C, P, and R.

3) Attackers Accessing More than One Device:

In this scenario, an attacker has access to more than one device and the protected software instances for each of the device. The attacker can do everything mentioned above. However, because every device has a different PUF response, accessing more instances does not help the attacker to attack the Dynamic PUF.

Moreover, the attacker can compare different versions of protected binaries or protected binary compiled for different devices of the same piece of software. Due to the randomness in generating the protection binary code, the attacker can locate the inserted protection code, and try to remove the protections. However, the attacker still needs to brute force all the PUF responses to obtain the call graph. Further, the HESP can be applied with the same deployment preferences, such that for the same program patched for different devices, the location of the protection codes and the type of the obfuscated protection code inserted are the same, and the only difference is the reference values, which depend on the PUF responses. In this case, the attacker still cannot locate the self-checksumming code to break the integrity.

C. Comparison to other Protection Schemes

1) Alternative Approaches:

Compared to software-only schemes, such as [3], by binding the execution with a PUF in hardware, the software binary cannot be run on an unauthorized device directly. Compared to binding the execution with an SRAM PUF [19], attacking a Dynamic PUF requires more effort, and this effort can be increased by choosing higher values for the S, C, P, and R parameters. The SRAM PUF generates all the responses at once based on the PUF readout from SRAM at system startup, and stores all the response in memory for latter runtime use. In this setup, a dynamic attacker, who can access the memory space of the protected software, can directly dump the memory to obtain all the static PUF response at once. With Dynamic PUFs, only the current PUF response is in memory and used on the fly. The attacker needs to obtain all the PUF responses one by one at runtime for the target device. Meanwhile, the runtime overhead of HESP is mainly due to the self-checksumming function, so the runtime overhead is similar to binding the software execution to SRAM PUF [19].

2) Related Approaches:

Software diversity: Software diversity introduces randomness in the target software, so that an attack targeting at one executable will only apply to a small number of other targets [11]. Software diversity can be implemented in different phases of the life-cycle of the software, e.g., implementation time, compilation time, installation time, load time or execution time. Here, since HESP only requires changes in the compilation step, we compare HESP with software diversity achieved at compile time. With compile-time software diversity, many different versions need to be generated at compile time, the cost of producing program variants is large. Also, each client must download a different program variant, so the cost to distribute and update the executable is proportional to the number of targets. The runtime overhead of compile-time software diversity schemes is reported to range from 1% to 11% [11].

TABLE I: Comparison of Overhead and Security Goals of Existing Schemes

	Compilation Time	**Distribution Time**	**Execution Time**	**Software update**	**Security Goals**			
					Defend "attack once apply everywhere"	**Anti-reverse Engineering**	**Device Authentication**	**Software Integrity**
Plain Software	compile once for all distributions	distribute the same binary	1	Same patch for all distributions				
Software Diversity – Compile Time	compile once for each distribution	distribute one unique binary for each device	1.01–1.11× [11]	a unique patch for each distribution	✓			
Software Obfuscation	compile once for all distributions	distribute the same binary	2×–30×* [10]	Same patch for all distributions		✓		
This Work	compile once for each distribution	distribute one unique binary for each device	up to 2×**	a unique patch for each distribution	✓	✓	✓	✓

* When the whole program is obfuscated.

** In this work, only small part of the software is required to be obfuscated.

Like all other security mechanisms using PUF, HESP needs to enroll the Dynamic PUF for each device, to generate the HDS and subsequently to use the PUF responses in the protection scheme. Because each device will have a different executable, the cost to compile, distribute and update the software is similar to software diversity schemes deployed at compile time. As discussed in [41], nowadays software for certain hardware platforms are only available through a single app store. It is not impractical that for each download an individual instance of software that is generated for the targeted device.

Software obfuscation: Software obfuscation is a mechanism to hide a program from analysis [4], [5]. There are a number of classes of software obfuscation transformations that protect against reverse-engineering, such as data obfuscation [6], reordering instructions [7], or control flow flattening [8], [9]. In addition, the control flow of a software program can be flattened and diversified so that there are many more control flow paths than in the un-obfuscated version, and thus the reverse engineering process is more complicated [8], [9]. Software obfuscation is usually achieved by rewriting or recompiling the executable from the source code automatically. This comes with compilation time overhead. In most cases, obfuscation increases the size of the executable, and the execution time of the obfuscated code increases as well. In [10], several different obfuscation schemes are implemented and evaluated. Depending on the scheme used, the runtime overhead ranges from 2× to 30×. If a software update is needed, a patch will be generated for all the distributions, and then the patch is distributed and applied. Since the same binary will be distributed to all users, the same patch will apply to all the distributions of the code. Hence, the complexity of software update of software obfuscation is the same as that of distribution time of software obfuscation.

HESP can be seen as an instance of software diversity and software obfuscation. The PUF response diversifies the binary, which prevents the "attack once apply everywhere", and the call graph scrambling code obfuscates the control flow, which defends reverse-engineering the call graph, although the initial goal of HESP is to protect software integrity and authorize the hardware. Other software diversity and software obfuscation mechanisms can also be applied over or combined with this work to achieve more security features. For example, the layout of the binary can be randomized as in software diversity, so that the destination addresses of function calls for each target executable are randomized.

Table I shows a comparison of the overhead and the security goals between different protection schemes. Different schemes have different security goals. Similar to compile-time software diversity, HESP generates different binaries for different targets, and thus, has the similar compilation, distribution and update overhead. The execution time overhead of HESP is up to 2× when C=2, mainly due to the self-checksumming functions. Compared to software obfuscation, HESP has a smaller program runtime overhead.

X. Conclusion

This paper presented a hardware-entangled software protection scheme which leverages Dynamic PUFs. Different from the usual static PUFs, a Dynamic PUF has time-dependent responses. The proposed Dynamic PUFs and linear self-checksum functions over the prime finite field can be used to detect if a software is running on an authorized device and if the software was modified. The tamper response code instances further scramble the call graph of the software and scramble the register values if the checksum or Dynamic PUF response is incorrect. An AutoPatcher can take the to-be-protected program and Dynamic PUF enrollments, and can automatically deploy our scheme. The protection scheme was implemented and tested on a commercial off-the-shelf platform (Intel Galileo Gen 2) and was shown to have a moderate performance overhead. The HESP framework and the Dynamic PUF code developed in this project will be made available under an open-source license at https://caslab.csl.yale.edu/code/puf-software-protection/.

References

[1] D. Aucsmith, "Tamper resistant software: An implementation," in *International Workshop on Information Hiding*. Springer, 1996, pp. 317–333.

[2] H. Chang and M. J. Atallah, "Protecting software code by guards," in *ACM Workshop on Digital Rights Management*. Springer, 2001, pp. 160–175.

[3] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *ACM Workshop on Digital Rights Management*. Springer, 2001, pp. 141–159.

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, p. 4, 2016.

[5] S. Banescu and A. Pretschner, "A tutorial on software obfuscation," *Advances in Computers. Elsevier*, 2018.

[6] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[7] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[8] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," University of Virginia, Tech. Rep., 2000.

[9] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *International Workshop on Information Hiding*. Springer, 2011, pp. 270–284.

[10] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *the International Workshop on Software Protection (SPRO)*. IEEE, 2015, pp. 3–9.

[11] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[12] K. Lofstrom, W. R. Daasch, and D. Taylor, "IC identification circuit using device mismatch," in *IEEE International Solid-State Circuits Conference*. IEEE, 2000, pp. 372–373.

[13] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 148–160.

[14] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proceedings of the Design Automation Conference (DAC)*. IEEE, 2007, pp. 9–14.

[15] S. Katzenbeisser, Ü. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2012, pp. 283–301.

[16] J. Sepulveda, F. Willgerodt, and M. Pehl, "SEPUFSoC: Using PUFs for memory integrity and authentication in multi-processors system-on-chip," in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2018, pp. 39–44.

[17] W. Xiong, A. Schaller, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer, "Run-time accessible DRAM PUFs in commodity devices," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2016, pp. 432–453.

[18] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM latency PUF: Quickly evaluating physical unclonable functions by exploiting the latency-reliability tradeoff in modern commodity dram devices," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 194–207.

[19] F. Kohnhäuser, A. Schaller, and S. Katzenbeisser, "PUF-based software protection for low-end embedded devices," in *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer, 2015, pp. 3–21.

[20] W. Xiong, A. Schaller, S. Katzenbeisser, and J. Szefer, "Dynamic Physically Unclonable Functions," in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2019.

[21] M.-D. Yu and S. Devadas, "Secure and robust error correction for physical unclonable functions," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 48–65, 2010.

[22] A. Schaller, W. Xiong, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, B. Skoric, S. Katzenbeisser, and J. Szefer, "Decay-Based DRAM PUFs in Commodity Devices," *Transactions on Dependable and Secure Computing*, 2018.

[23] G. Wurster, P. C. Van Oorschot, and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2005, pp. 127–138.

[24] R. Nithyanand and J. Solis, "A theoretical analysis: Physical unclonable functions and the software protection problem," in *IEEE Symposium on Security and Privacy Workshops (SPW)*. IEEE, 2012, pp. 1–11.

[25] M. V. Pedersen, J. Heide, P. Vingelmann, and F. H. Fitzek, "Network coding over the $2^{32} - 5$ prime field," in *IEEE International Conference on Communications (ICC)*. IEEE, 2013, pp. 2922–2927.

[26] S. Rosenblatt, S. Chellappa, A. Cestero, N. Robson, T. Kirihata, and S. S. Iyer, "A Self-Authenticating Chip Architecture Using an Intrinsic Fingerprint of Embedded DRAM," *IEEE Journal of Solid-State Circuits*, pp. 2934–2943, 2013.

[27] A. Rahmati, M. Hicks, D. E. Holcomb, and K. Fu, "Probable Cause: The Deanonymizing Effects of Approximate DRAM," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 604–615.

[28] S. Sutar, A. Raha, and V. Raghunathan, "D-PUF: An intrinsically reconfigurable DRAM PUF for device authentication in embedded systems," in *Proceedings of the International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 2016, pp. 1–10.

[29] A. Schaller, W. Xiong, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer, "Intrinsic rowhammer PUFs: Leveraging the rowhammer effect for improved security," in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017, pp. 1–7.

[30] F. Tehranipoor, N. Karimian, K. Xiao, and J. Chandy, "DRAM based intrinsic physical unclonable functions for system level security," in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2015, pp. 15–20.

[31] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceedings of Annual International Symposium on Computer Architecuture (ISCA)*. IEEE, 2014, pp. 361–372.

[32] "The Reliability of SRAM PUF," https://www.intrinsic-id.com/wp-content/uploads/2017/08/White-Paper-The-reliability-of-SRAM-PUF.pdf.

[33] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres*, vol. 2, no. 2, pp. 147–56, 1959.

[34] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and control*, vol. 3, no. 1, pp. 68–79, 1960.

[35] B. Gassend, M. V. Dijk, D. Clarke, E. Torlak, S. Devadas, and P. Tuyls, "Controlled physical random functions and applications," *Transactions on Information and System Security (TISSEC)*, vol. 10, no. 4, p. 3, 2008.

[36] "Intel® galileo gen 2 development board," https://ark.intel.com/products/83137/Intel-Galileo-Gen-2-Board.

[37] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

[38] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su, "Identifying and understanding self-checksumming defenses in software," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 207–218.

[39] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security & Privacy*, vol. 5, no. 3, 2007.

[40] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, 2012.

[41] T. Wolf, "Sacrificing interoperability for information security: Containing data loss and malware propagation," https://www.sigarch.org/sacrificing-interoperability-for-information-security-containing-data-loss-and-malware-propagation/.

Wenjie Xiong received her B.Sc. in Microelectronics and Psychology from Peking University in 2014. She is currently a Ph.D. candidate at the department of electrical engineering at Yale University, working with Prof. Jakub Szefer. Her research interests comprise Physically Unclonable Functions, and cache side channel attacks and defenses.

André Schaller received his M.Sc. degree from the Technical University of Darmstadt. He then joined the Security Engineering Research Group, led by Prof. Dr. Stefan Katzenbeisser. His main research interest comprised hardware-based cryptography, security of mobile and embedded systems and Physically Unclonable Funtions in particular. In 2017 he received his Ph.D. from the Technical University Darmstadt. He has been working as a security engineering consultant in the aerospace sector since then.

Stefan Katzenbeisser (S'98–A'01–M'07–SM'12) received the Ph.D. degree from the Vienna University of Technology, Austria. After working as a Research Scientist with the Technical University of Munich, Germany, he joined Philips Research as a Senior Scientist in 2006. After holding a professorship for Security Engineering at the Technical University of Darmstadt, he joined University of Passau in 2019, heading the Chair of Computer Engineering. His current research interests include embedded security, data privacy and cryptographic protocol design. He has authored more than 200 scientific publications and served on the program committees of several workshops and conferences devoted to information security. He is currently serving on the Information Forensics and Security Technical Committee of the IEEE Signal Processing Society.

Jakub Szefer (M'08–SM'19) received B.S. with highest honors in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign, and M.A. and Ph.D. degrees in Electrical Engineering from Princeton University where he worked with Prof. Ruby B. Lee on secure hardware architectures. He joined Yale University in summer 2013 as an Assistant Professor of Electrical Engineering, where he started the Computer Architecture and Security Laboratory (CASLAB). His research interests are at the intersection of computer architecture, hardware security, and FPGA security. His research focuses on secure hardware-software processor architectures, side and covert channel attacks and defenses, speculative execution attacks and defenses, hardware security verification, physically unclonable functions, hardware FPGA implementation of cryptographic algorithms, and FPGA security. His research is supported through National Science Foundation and industry donations.