universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

DRAM PUF-based Hardware-Software Binding through LLVM passes

verfasst von | submitted by

Bc. Matúš Mrekaj

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl

Acknowledgements

I would like to thank Sebastian for his advice and quick responses whenever I was stuck and had questions about how to proceed. I would also like to thank Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer for their previous work on PUFs, DRAM PUFs, and dynamic PUFs on commodity hardware, without which this work wouldn't have been possible.

Finally, I would like to thank my family for their support while I was working on this thesis and for their support in general during my studies, without which I wouldn't have made it this far.

Abstract

IoT is increasing the demand for embedded devices in many industries. These devices may not have the necessary security protection due to cost savings. Recently, Physically Unclonable Functions (PUFs) have been explored to leverage the hardware already present in these devices to implement missing security mechanisms that provide device authentication for software distributed for specific hardware. By using a PUF, the deployed software is essentially made to run only on the specific hardware for which it was enrolled.

A new implementation of an intrinsic PUF based on a decay-based DRAM PUF that can be queried at runtime during normal device operation using Linux kernel modules is described on a commercial off-the-shelf IoT device, the Beaglebone Black rev 3C. The implementation uses a LLVM out-of-tree pass to patch LLVM IR to use the PUF implementation available on the device, combined with self-checksum schemes. The LLVM pass destroys the call graph of the program to be patched and embeds instructions in the code that incrementally rebuild the call graph at runtime when the program is executed on a device running the PUF. Each PUF response at user enrolled query times is used to rebuild the call graph. If tampering is detected, either by the selfchecksumming schemes or by running the patched program on an unauthorized device, the tamper response mechanism is triggered, either by introducing undefined behavior in the former case or by reconstructing the wrong call graph in the latter case.

The scheme was applied to a Rust TCP server performing cryptographic operations. Stable 32-bit PUF responses could be generated at room temperature with error correction up to 8 bits, or 25%. The impact of the code inserted by the LLVM pass on the sample program results in a larger binary size by $\approx 30\%$ and longer compilation times of $\approx 20\%$. The custom DRAM refresh in the kernel module takes up 28% of CPU time while the PUF is being evaluated. The stability of the PUF responses generated by the Linux kernel modules was evaluated over a period of ≈ 1 month.

Kurzfassung

Das Internet der Dinge führt in vielen Branchen zu einer steigenden Nachfrage nach eingebetteten Geräten. Aufgrund von Kosteneinsparungen kann die erforderliche Sicherheit dieser Geräte oftmals jedoch nicht gewährleistet werden. In den vergangenen Jahren wurden Physically Unclonable Functions (PUFs) untersucht, um in diesen Geräten bereits vorhandene Hardware zu nutzen, um fehlende Sicherheitsmechanismen zu implementieren, beispielsweise Geräteauthentifizierung für Software. Die Verwendung eines PUF gewährleistet im Wesentlichen, dass die Software nur auf der speziellen Hardware läuft, für die sie registriert wurde.

In dieser Arbeit wird eine neuartige Implementierung eines Decay-DRAM-PUFs auf einem Beaglebone Black beschrieben, der zur Laufzeit während des normalen Gerätebetriebs mit Linux-Kernel-Modulen abgefragt werden kann. Die Implementierung verwendet einen LLVM-Out-of-Tree-Pass, um die LLVM IR so zu patchen, dass die auf dem Gerät verfügbare PUF-Implementierung in Kombination mit Self-Checksumming verwendet wird. Der LLVM-Pass zerstört den Call-Graph des zu patchenden Programms und integriert Anweisungen in den Code, die den Call-Graph zur Laufzeit inkrementell wieder aufbauen, sobald das Programm auf einem Gerät ausgeführt wird, auf dem der PUF läuft. Jede PUF-Antwort wird zur Rekonstruktion des Call-Graphen verwendet. Im Falle einer Manipulation durch Veränderung des Programmcodes oder Ausführung des gepatchten Programms auf einem nicht autorisierten Gerät, wird eine Tamperproofing-Reaktion ausgelöst.

Das Schema wurde auf einen Rust-basierten TCP-Server angewendet, der kryptografische Operationen durchführt. Es konnten stabile 32-Bit PUF-Antworten bei Raumtemperatur mit einer Fehlerkorrektur von bis zu 8 Bit oder 25% erzeugt werden. Die Auswirkungen des durch den LLVM-Pass eingefügten Codes auf das Beispielprogramm resultieren in einer um ca. 30% größeren Binärdatei und einer um ca. 20% längeren Kompilierzeit. Das Kernel-Modul beansprucht während der Auswertung des PUFs 28% der CPU-Zeit. Die Stabilität der PUF-Antworten wurde über einen Zeitraum von ca. einem Monat evaluiert.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction

Embedded devices are used in many industries, and growth is expected to continue in the future. These devices are usually part of a larger system where they may process sensitive information, monitor conditions or issue commands to control other parts of the system. In such scenarios, device authentication can be a critical requirement for the reliability of the overall system. As most organisations are focused on maximising profits, cost savings that are part of their own profit maximisation can result in these devices lacking necessary security features, thus creating a vulnerability or potential attack vector for the system.

The goal is to prevent an attacker from modifying existing binaries on these devices and from running them on their own unauthorised hardware. Already today, some of these vulnerabilities can be mitigated using existing techniques to add a layer of defence. Software deployed on these devices can be protected against reverse engineering by static or dynamic analysis using obfuscation techniques such as control flow flattening, replacing instructions with mixed boolean arithmetic (MBA) and various static and dynamic obfuscation methods in general [8]. From another perspective, binaries can also be protected by using tamper detection, where the program runs as intended even if the adversary alters the execution of the control flow. The intention may be to make the program misbehave, degrade its performance or even crash if the tampering is detected. Tamperproofing can be implemented through self-checksumming schemes, where a hash is calculated and compared to a reference value. These schemes can be run continuously at runtime to check integrity [9, 29]. The combination of these two methods can make it more difficult to analyse the binary files, but cannot prevent the attacker from reverse engineering them. The intention is to make the process as irritating as possible, in the hope that reverse engineering the binaries will take more time than the adversary is willing to invest.

To give a practical example: The Skype client along with the protocol combines different obfuscations with the use of self-checksumming schemes, but with enough time invested in reverse engineering these techniques, they were eventually broken [7].

On the hardware side, in the late 2000s, some research shifted to Physically Unclonable Functions (PUFs) for low-cost device authentication and the use of some cryptography building blocks, such as secret generation. PUFs are a challenge-response mechanism that extracts unique properties from the physical characteristics of an object. In the context of electronic devices, the unique properties would be the random variations introduced in the manufacturing process of some of the components, such as transistors. Some of these designs required the inclusion of specialised circuitry as part of the devices for the use of the PUF [34]. The inclusion of a specialised circuit as part of a device would most likely result in increased cost spending as the design and manufacture of commonly

mass-produced systems would need to be adapted.

As a result of that, some of the later research has explored intrinsic PUFs. Intrinsic PUFs leverage hardware components that are already present on off-the-shelf devices. They can be implemented without any modification to the hardware. Random Access Memory (RAM) modules, namely Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM), have been explored [40, 33] and have been shown to have PUF characteristics.

The work of Xiong et al. [41] showed that not only can DRAM-based PUFs be queried at runtime without hardware changes, but also that the response of the PUF can depend on time. The same challenge queried at different times would yield different answers. This concept was combined with the self-checksumming and obfuscation techniques mentioned above to create binaries that query the PUF at runtime at different points in time to verify the integrity of the hardware running the binaries.

The device that was used in the work, Intel Galileo Gen 2, was discontinued [19] and the software itself is outdated and hardly executable today and almost impossible to test or further build upon without the device. Therefore, the focus of this work is on a new implementation of hardware-software binding using DRAM PUFs and direct use of LLVM to patch LLVM IR code to utilize the PUF, for a new device that is still supported at the time of writing this thesis, namely the Beaglebone Black rev 3C.

1.1 Contributions

The main contributions of this thesis are:

- A new implementation for an intrinsic PUF based on Decay-based DRAM PUF that can be queried at runtime by making use of Linux kernel modules, on an off-the-shelf IoT device Beaglebone Black rev 3C
- LLVM out of tree pass to patch LLVM IR to make use of the PUF implementation, combined with self-checksumming schemes
- Evaluation of the stability of the PUF implementation, including discussion for enrollment of DRAM cells which directly influence the stability of the PUF responses at runtime.

1.2 Reading Guide

The rest of this thesis is organized as follows: Chapter 2 explores related work for DRAM PUFs. Chapter 3 introduces the background concepts used in the implementation part. Chapter 4 describes the implementation and the choices made. In Chapter 5 the PUF implementation is evaluated and the differences to other implementations are discussed. Finally Chapter 6 summarizes the results of this thesis.

2 Related Work

In this chapter, we will briefly discuss previous work on SRAM-based PUFs and their limitations, and then focus mainly on giving an overview of the different DRAM-based PUFs that have been developed so far.

2.1 SRAM PUFs

Guajardo et al. [15] introduced the notion of an intrinsic PUF that was based on SRAM. SRAM is a volatile memory component, which retains data while it is connected to power. During the manufacturing process, variations are introduced in the transistors that make up the SRAM cells, resulting in slightly different voltage thresholds, meaning that each cell has a bias towards a 1 or a 0 independently of other cells. It has been found that these biases are stable and repeatable at device boot and can be extracted. Other work [14, 25, 33] has evaluated SRAM PUFs while others [38] have investigated their longevity.

SRAM-based PUFs require the extraction of the PUF response at a very early stage in the process, namely when the device is booting. This means that the extracted stable bits must be stored in memory or in another volatile memory component during the runtime of the device, as long as it is connected to power or until the response is no longer needed.

2.2 DRAM PUFs

Similar to SRAM, DRAMs also have PUF characteristics. The main difference to SRAM PUFs is that some of the implementations can be queried at runtime of the device, thus allowing runtime accessible authentication of the hardware. There are several types of DRAM-based PUFs that have emerged, namely decay-based, latency-based, start value-based and write duty cycle-based. Each of these types is described in more detail in a separate subsection.

2.2.1 Write Duty-cycle

Due to the little research in utilizing DRAM modules for authentication purposes at that time, Hashemian et al. [16] developed an authentication of hardware process using a DRAM PUF. The DRAM PUF was based on modifying the write duty cycle [1] of DRAM modules. By changing the period of the write duty cycle, cells become unstable and fail the write operation. The bit error pattern from the failed operation would then

[1] A duty cycle is a period of time during which a system is active [1]. In the context of a write, this would translate to following a series of steps to ensure that data is correctly written to memory

be extracted and used as response of the PUF. The reduction of the duty cycle was implemented by modifications to the memory controller. The PUF was evaluated and showed promising results.

2.2.2 Startup-value Based

Tehranipoor et al. [37] investigated and evaluated a startup-value-based DRAM PUF using an FPGA development board. They observed that DRAMs exhibit similar behavior to SRAMs at startup in that the DRAM cells have random startup values that could allow them to be used as a PUF. The authors described a novel PUF ID generation, that uses a specific enrollment algorithm for the cells, to generate stable PUF responses. The PUF was evaluated on data from multiple DRAMs.

In a short paper, Anagnostopoulos et al. [5] explored the startup value PUF on off-the-shelf devices namely the Intel Galileo Gen 2 and PandaBoard. After conducting experiments on these boards, they concluded that these start-up values cannot be used as a PUF on these boards.

2.2.3 Decay Based

Schaller et al. [32] presented a decay based DRAM PUF that was evaluated at different decay times on two off-the-shelf devices namely, Intel Galileo Gen 2 and PandaBoard. The PUF works by disabling the DRAM refresh for a specific memory region in which the DRAM cells will decay until the refresh is re-enabled again. Depending on how long the timeout for the decaying of the cells is[2] different bit flips will be introduced at different timeouts. The bits from that isolated memory area are then used as the PUF response. The PUF is then combined with a Helper Data System (HDS) scheme to obtain a stable bit pattern. The HDS takes care of the noise in the PUF, as some of the bit flips during the decaying are not stable when repeated. Further, a lightweight authentication protocol is then introduced that uses the PUF itself.

Since the timeout for the decay-based PUF may need large amount of time Kumari et al. [22] proposed methods for run-time rapid extracting of values for the decay-based DRAM PUF which were successfully implemented and tested on the Intel Galileo Gen 2 at room temperatures. The PUF used 1024KB of memory and stable responses could be obtained from 15 seconds.

Sutar et al. [35] similarly evaluate a decay-based DRAM PUF for device authentication using an FGPA development board at different temperatures and also observe the effect of DRAM aging. The main differences are that they use helper data generated by a Hamming encoder/decoder during the enrollment phase to perform error correction, and then use the resulting bit stream as the response. In addition, to avoid DRAM cells with insufficient entropy, a block selection algorithm is used to select blocks that meet the required minimum entropy.

[2]The timeout for a decay-based DRAM PUF must be long enough to obtain a sufficiently stable response. A minimum of 10 seconds was taken into account in the work, which can extend to several minutes.

Xiong et al. [41] further build upon the decay-based PUF, by describing a new Hardware-Entangled Software Protection (HESP) scheme and an AutoPatcher, written in Python, that patches LLVM IR code with the protection code. The HESP scheme uses tamper detection and tamper response to detect malicious behavior and change the outcome of the program itself, including detecting whether the software is running on an authorized device. The scheme was tested successfully on an off-the-shelf device, namely the Intel Galileo Gen 2.

In addition, Schaller et al. [31] explored a decay-based PUF in combination with the Rowhammer effect [3] and evaluated it on off-the-shelf devices with a software-only solution, without any hardware modifications. It was observed that the bit flips introduced by the row hammer effect produced stable patterns, but with insufficient entropy. The paper presents a combination of the Rowhammer effect with a decay-based DRAM PUF to maximise the number of bit flips, which are then used as a stable PUF response.

2.2.4 Latency Based

Due to the long duration needed to obtain a response when using the Decay-based DRAM PUF, Kim et al. [21] investigated another type of runtime accessible DRAM PUF, called DRAM latency PUF. The PUF can be implemented with no additional hardware using a software only solution to modify the DRAM timing parameters specified by the manufacturer, namely the read access latency (t_{RCD}), provided that the given off-the-shelf commodity device supports such modifications via software. By reducing[4] the t_{RCD} parameter below the manufacturer specified recommended value, errors are introduced in DRAM cells after a read operation. To generate a PUF response, the read operation would be repeated multiple times with the reduced parameters to identify repeatable stable cells failures, which would be used as the PUF response. Experiments were performed on state of the art LPDDR4 DRAM devices. The PUF can be evaluated in $\approx$ 88.2ms on average across all devices at all operating temperatures.

A separate group, Talukder et al. [6] experimented with the latency-based DRAM PUF by modifying different timings parameters, namely the precharge time (t_{RP}), using an FPGA development board. Reducing the precharge time value might result in deviations in the precharge of the bitline which might not reach the needed value of $V_{DD}/2$[5]. This in consequence might introduce variations on the charge of the storage capacitor of a DRAM cell, which during a read operations might behave differently. With this approach the DRAM cells may not have sufficient randomness, thus a selection algorithm for suitable cells is used for a robust PUF response. The PUF can be evaluated in $\approx$ 1.59ms

Miskelly et al. [27] build upon the read-access latency-based PUF by further minimizing the query time. The query time was reduced to an average of $\approx$ 3ms and $\approx$ 31ms in the

[3] The Rowhammer effect is an exploit for DRAM in which rapid sequential activation of a single memory row leads to charge leakage in neighbouring rows, resulting in bit flips in those neighbouring rows [3]

[4] As it was the case with Decay-based DRAM PUFs, different settings for the timings parameters results in different patterns [21]

[5] In DRAMs, the bit line voltage needs to be near $V_{DD}/2$ for accurate data reading from the DRAM memory cells. V_{DD} refers to the supply voltage

worst case while still maintaining uniqueness and robustness. The result was achieved with careful selection of bit patterns and with more efficient enrolment and query algorithms. The PUF was evaluated on off-the-shelf devices without any hardware modifications.

Najafi et al. [28] proposed a novel strategy to generate reliable PUF responses. The proposed PUF implementation uses bitmap images of raw DRAM content and their respective entropy features to generate reliable responses. The PUF was implemented and evaluated on an FPGA development board, with modified DRAM timing parameters. The evaluation time of the PUF was even further reduced to $\approx$ 0.93ms. Finally, the authors described a state-of-the-art authentication protocol in terms of computational cost that uses the proposed PUF.

3 Background

In this thesis a scheme for tying an instance of software to a single IoT board by making use of a Decay-based DRAM PUF is implemented. A program written in a programming language, such as Rust or other language with a LLVM frontend, will be patched by using LLVM passes that embed code into the control flow. When the patched software is executed on the target IoT device, the embedded instructions prompts the running PUF on that IoT device for authentication. Depending on the PUF response, the software will either continue executing if it is executing on an authorized IoT device, or abort as the PUF returns different responses on different IoT boards, essentially tying a given instance of the software to a single IoT board. Before we dive into the implementation part, related concepts need to be introduced in this chapter to understand the remainder of this thesis.

3.1 Dynamic Random Access Memory

DRAM is a volatile memory component that only retains data when connected to a power source. A DRAM consists of many individual cells that are grouped in banks. The number of banks that make up a DRAM can vary between 1, 2, 4 or 8 (which is the most common). Each DRAM cell typically comprises a transistor that is linked to the word line and a storage capacitor connected to the bit line via the transistor. A High-level schematic of a DRAM is depicted in figure 3.1.

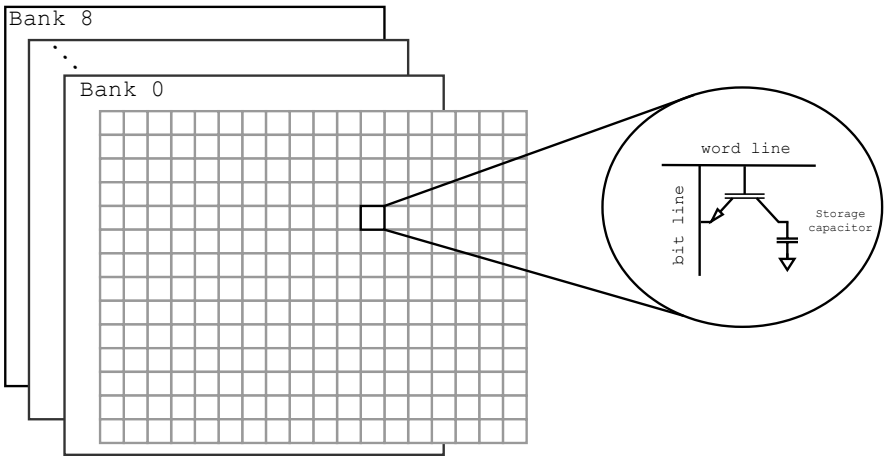


Figure 3.1: High-level DRAM overview

Cells that are connected to the same word line form a row. Bitlines are connected to sense-amplifiers which are used when converting the charge on the capacitor into its digital representation. A single bit of data is stored in a single DRAM cell. The storage capacitor in a cell can only have two states, either charged or discharged. The former represents the bit value of 1 and the latter a value of 0. The charge that is stored in the capacitor leaks over time, and when enough time has passed, the data stored in the cell is lost. The time that an individual cell can retain the stored data is also referred to as retention time, and this time is different for each cell of the DRAM. This is due to the variations introduced during the manufacturing process. Therefore, each DRAM cell has a bias towards one of the values. In order to retain the data stored in the DRAM cells over long periods of time, the cells must be refreshed at regular intervals, which is usually 64ms, but can vary [24].

3.2 Physically Unclonable Functions

Physically Unclonable Functions are objects that when given a challenge return a response. A PUF can be imagined as composed of two parts the physical part, which is extremely difficult to clone and the interactive part that operates upon the given challenge and the unique physical properties to generate a response, sometimes also referred to as a fingerprint in the context of semiconductor devices[15].

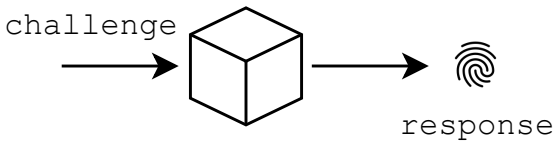


Figure 3.2: PUF concept

A PUF can have the following properties [15, 26, 39]:

Unclonable. The PUF responses depend on underlying physical properties. The same request to multiple instances of the same PUF should not produce the same response.

$$PUF_x(challenge_i) \neq PUF_y(challenge_i)$$

Unpredictable. In the absence of the PUF, the responses to the challenges should only be reconstructable with negligible probability and vice versa. A challenge-response pair should not reveal any useful information about another challenge-response pair.

Stability. Passing a challenge to the PUF should result in a stable response. Repeating the same challenge should always return the same response.

Uniqueness. When the PUF is prompted with different challenges, it should always return different responses that depend on the current challenge passed. No two challenges

should have the same response.

Strong/Weak. A PUF can be classified as weak or strong. This categorization is based on the number of challenge-response pairs (CRPs) that a PUF can generate. If a PUF has a low number of CRPs, we consider it a weak PUF, as an exhaustive attack has a high chance of success. If the PUF has many CRPs, it is a strong PUF, as the probability of an exhaustive attack being successful is negligible.

Static/Dynamic. A PUF can also be classified as static or dynamic. When a challenge is given to a PUF and the PUF does not take into account the time elapsed since the challenge, it is considered a static PUF. PUFs that consider time as part of the response are classified as dynamic.

3.2.1 Dynamic PUFs

Xiong et al. [39] introduced the concept of a dynamic PUF. Dynamic PUFs have the additional property that the response to a challenge depends on the time elapsed since the challenge was given to the PUF. A PUF that was initialized with a challenge where responses are queried at different times should provide different stable and unique responses. This implicitly means that a response is obtainable only within a certain time period until it changes. For two points in time T_1, T_2, a response should be obtainable for the time period of $|T_1 - T_2| = \alpha$, where the value of α depends on the implementation. The response should not be obtainable before T_1 or after T_2, as this would be considered as time intervals for other responses.

In the paper, DRAM based PUFs namely the Decay-based and Latency-based, have been shown to have the Dynamic PUF property, however. The latency-based version was not further worked on, as it requires software controlled access to modification of the DRAM timing parameters, as it was discussed in Chapter 2, which is not common on commodity off-the-shelf devices.

3.3 Decay-based DRAM PUF

The intrinsic property of a DRAM that the storage capacitors lose charge over time, and that this retention time is different and independent of other cells, is the key building block for the dynamic PUF.

A Decay-based DRAM PUF is considered as a weak PUF[40], which can achieve the dynamic property as follows. Within the DRAM, reserve a memory region for the PUF so that this region will not be used by other processes. Disable the refresh operation of the DRAM cells for the PUF memory region and initialize the region with the challenge. Due to the varying retention time of the individual DRAM cells and each cell having a bias towards a 1 or a 0, at different points in time T_1, T_2, when reading the PUF memory region, different bit patterns will be present, which we can treat as the PUF response for that respective point in time.

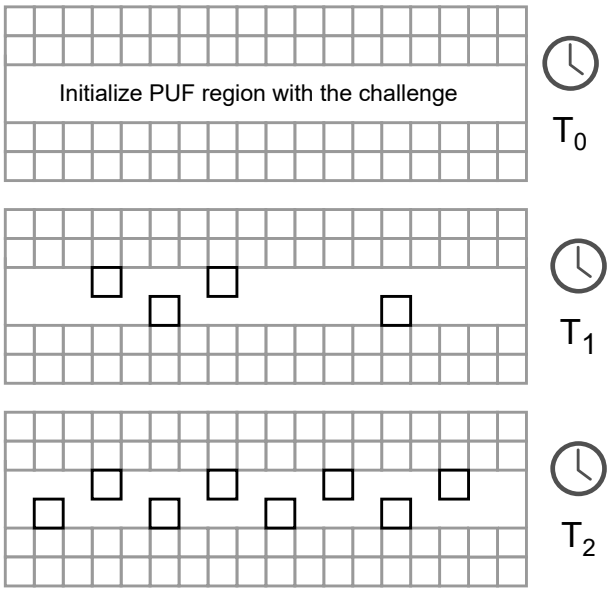


Figure 3.3: Decaying of DRAM cells

This process is depicted in figure 3.3. From the image one can also observe a subtle property. The decaying cells at a later time are mostly a superset, excluding the unstable cells that occur, of the cells decayed at an earlier time.

$$D_{T_1} \subseteq D_{T_2} - \epsilon$$

where:

- D_{T_1} is the set of cells decayed at time T_1, D_{T_2} is the set of cells decayed at time T_2.
- ϵ is the set of unstable cells.

3.4 Error Correction

DRAM cells are inherently unstable, and without a regular refresh, DRAM cells will lean towards a 1 or a 0 over time. If we read the response from the raw DRAM content to construct a PUF response, that response will be subject to noise. Some cells have a stable retention time, where repeating the same decay timeout will yield the same result, while others are more unstable on repeated readings of the same decay timeout. The reading of the bit pattern generated by the decay of the cells can essentially be considered as a noisy channel from which the noise must be filtered out. Further, the retention time for some of the cells can change due to aging of the components or being subject to temperature variations [37, 42, 36], thus error correction is required to create a stable PUF response. There are many possibilities for an error correction algorithm such as Hamming codes, BCH codes, repetition codes, etc. For this implementation, Reed-Solomon codes are used.

3.4.1 Reed-Solomon Codes

Reed-Solomon codes [30] is an error-correction algorithm based on finite fields arithmetic. The algorithm consists of an Encoder and a Decoder. The encoder takes the input and treats it as blocks of data which are in turn processed as elements of a polynomial of a finite field. The encoder adds extra blocks of redundant data to the original message, which are then in turn used by the decoder when reconstructing the received blocks of data. This redundancy enables the decoder to reconstruct the original message even in the presence of errors. Reed-Solomon codes are configurable and allow for encoding of vasts amounts of different data. When encoding the following properties apply[2]:

$$RS(n, k, s, t)$$

where:

- s is the width of the symbols in the message.
- k is the number of symbols with width of s
- t is the number of symbols that can be corrected.
- n is the result of the encoder that comprises of k and $2t$ parity symbols. For a given symbol size s, n can have a max value of 2^s-1. To correct up to t symbols, $2t$ of parity symbols are required to be part of n.

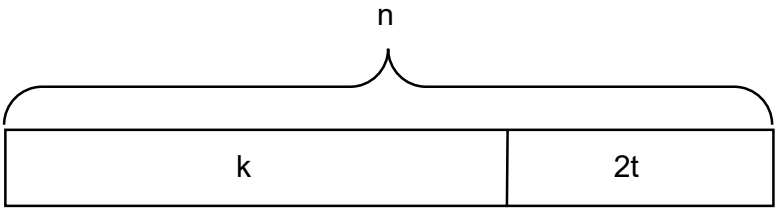


Figure 3.4: Reed Solomon encoded message

3.5 LLVM

LLVM is a comprehensive compiler infrastructure. The modular architecture of LLVM enables development of custom frontends and backends targeting different architectures, enabling faster development of new programming languages by allowing re-use of existing optimizations, transformation and backends, while also helping to improve existing programming languages [23].

A frontend is a program that parses the syntax of a programming language and outputs the program logic in the LLVM Intermediate Representation (LLVM IR). This

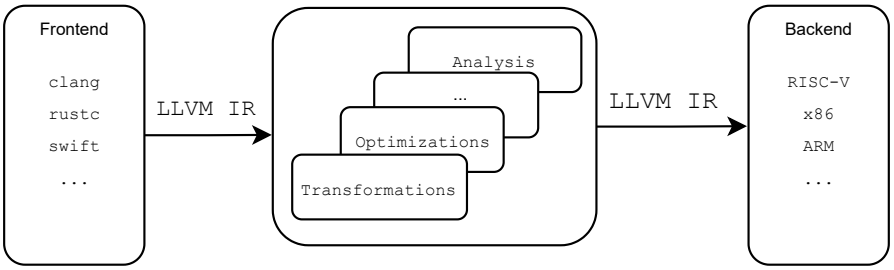


Figure 3.5: High-level overview of LLVM

intermediate representation is then passed as input to one of the available backends for compilation to a native architecture.

LLVM IR is an important feature of LLVM, as it enables a language-independent representation within the LLVM infrastructure. LLVM IR follows the Static Single Assignment form (SSA). This form restricts code by following two strict rules. Only a single assignment to a variable is allowed and the variable needs to be defined before use [4]. In LLVM most of the transformations, optimizations, analyses, etc. are implemented as LLVM passes that are executed on this intermediate representation that results in a new IR that eventually is passed to the backend. The LLVM infrastructure allows for creation of arbitrary new, out-of-tree[1] passes that can be used on the LLVM IR obtained from a frontend, alongside existing LLVM passes. This feature is the core of this work and will be described in more detail in Chapter 4

3.6 Binary Protection

The software distribution model consists of compiling a program into the appropriate binary representation for the target platforms on which the users of that software can use it. Distribution can be an attack vector that the developers of the software have not considered in advance. Imagine an attacker whose only goal is to obtain this distributed binary. There can be several reasons for this, just to name a few:

- Using the binary on his own hardware without paying for the software
- Removing security mechanisms so that he can resell the software at a lower price
- learn trade secrets from a competing company

One such great example is the distribution of video games. Developers spend years on building a product which then on release gets obtained by an adversary that modifies the distributed binary such that the security mechanisms are removed and may further distribute it free or at a lower price than the software itself.

[1] An out-of-tree pass is a LLVM pass that is not part of the official source code of the LLVM infrastructure. It is developed separately enabling custom modifications to the LLVM IR

At the time of writing, there is no way to prevent an attacker from reverse-engineering the binary. Once the attacker has obtained the binary, there is no limit as to what he/she can do with it as long as they have a motive to do so. Mechanisms exists to address this problem to add an extra layer of defense to the binary, namely obfuscation, watermarking or tamper detection, also known as Surreptitious Software [8, 9, 10]. By employing these techniques, the resulting code in the binary is not designed to completely prevent reverse-engineering. Rather, the intent here is to waste resources available to the attacker in the hope that they will not spend a large portion of them, one of those resources is time. The intention is to slow down the attacker so that reverse-engineering the binary takes more time than the attacker is willing to invest.

In this implementation only a few of the techniques are utilized which are described in the following subsections.

3.6.1 Opaque Predicates

Opaque predicates are expressions that evaluate as either true or false. Compilers always try to optimize the code as much as possible. Due to the optimizations, code with side effects that the compiler thinks will not be used can be optimized away. The use of opaque predicates can be useful in this situation as protection against the optimizations. An opaque predicate must be complicated enough so that the compiler does not infer the result and optimize it together with the side effect code.

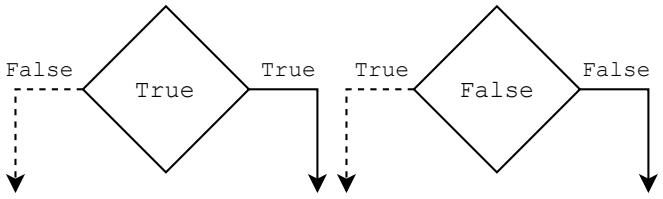


Figure 3.6: Opaque Predicates

3.6.2 Operation Substitution

Operation substitution, also known as Mixed Boolean Arithmetic (MBA), is a simple transformation of source code in which the instructions are replaced by a sequence of carefully crafted instructions that obfuscate the original instruction and ultimately produce the same result as the original instruction.

3.6.3 Control Flow Flattening

Loops or conditional statements in a program create branches in the control flow graph, which can usually be decompiled into the original representation with little effort using a tool such as Ghidra. The intent of control flow flattening is to destroy the control flow of

a program such that it cannot be inferred easily. The effect of this technique can be seen in figure 3.8 where it was applied on the function in figure 3.7.

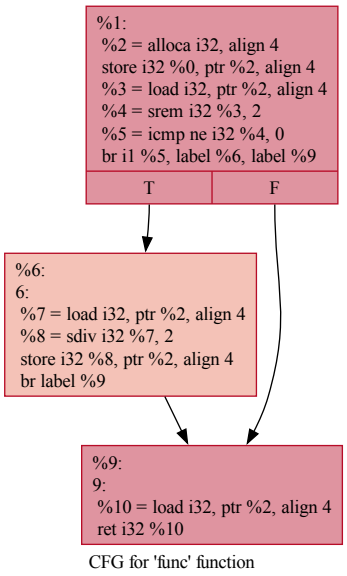


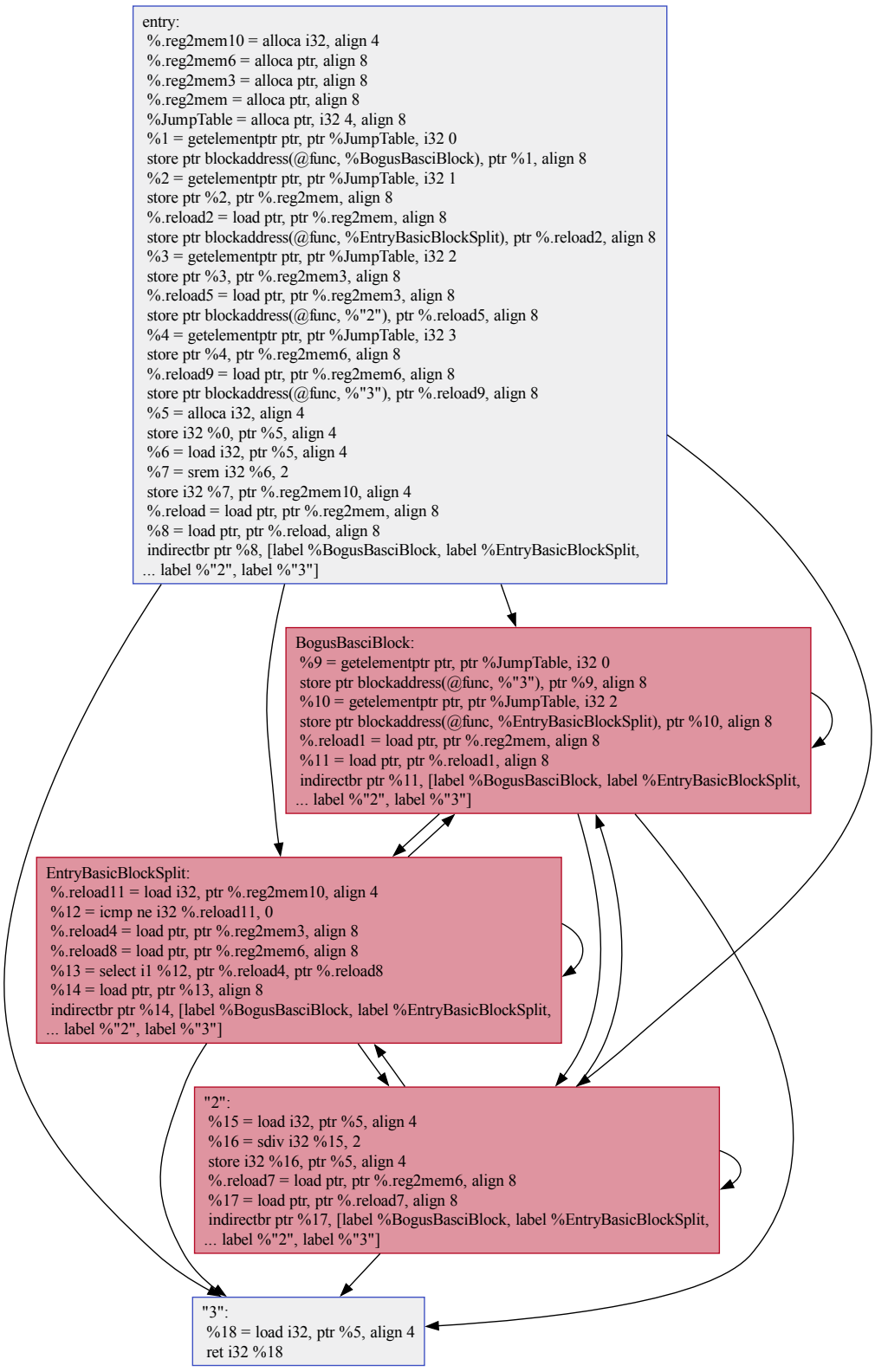
Figure 3.7: Original Function

3.6.4 Call Graph Scrambling

By tracing the call graph of a program, it is possible to identify sections within the program in which security mechanisms such as authentication or the processing of sensitive data are carried out. Call graph scrambling destroys the call graph by replacing function calls with indirect calls based on a runtime calculation of the function address [41]. This is intended to hide which function is actually being called. On small codebases, this may not be as effective as for larger codebases where functions with similar signatures exist. To reconstruct the call-graph without knowing the correct values, the program would need to be executed with many inputs to restore the original call graph.

3.6.5 Checksumming

Checksumming code is a form of tamper detection in which the integrity of the binary is checked at runtime by checksumming over a range of instructions and comparing it with a reference value. Depending on the result of the checksum, the program either continues execution without error or the checksum triggers built-in mechanisms, also referred to as tamper response, that can result in abort of the execution or further modify the binary, resulting in undesired behavior.



CFG for 'func' function

Figure 3.8: Control Flow Flattening with Aliasing

4 Implementation

The implementation consists of several parts written in different programming languages, C++ which is used for the LLVM out-of-tree pass, C for the kernel modules and Rust which is used on the rest of the utilities. The entire project consists of a total of 6k lines of code, excluding single header files dependencies used in C++. The source code is available at `https://github.com/Despire/puf-software-protection`

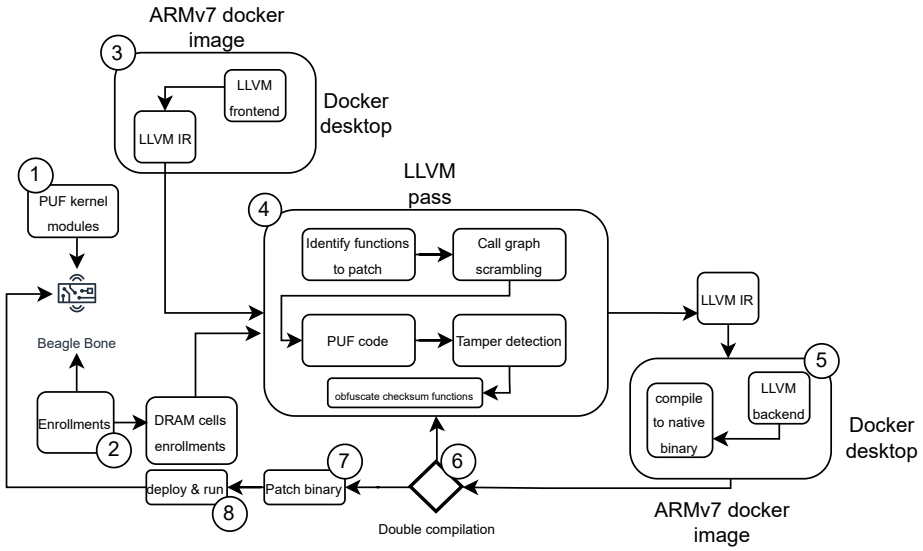


Figure 4.1: Process of patching a binary

Figure 4.1 depicts the process from the start by collecting enrollments on the Beaglebone Black rev 3C, for the decay-based DRAM PUF to the patching of the LLVM-IR to utilize the PUF at runtime, which in turn is compiled into a binary by passing it to an LLVM backend that is then executed on the device. The implementation uses Docker Desktop[1] with a specific image that emulates the CPU architecture used on the Beaglebone Black, namely ARMv7. The board does not need to be present during the patching and compiling of the binary. With this decision we hope that in the future, when the boards are no longer manufactured, the implementation can be used as a reference where the binary can still be produced, as it uses emulation for producing the binary. The rest of this chapter

[1] On Windows and Macos this should be the default, on Linux running bare Docker is not sufficient as the docker-image emulates a different architecture, thus running Docker Desktop on Linux is also necessary

will be devoted to describe all of the parts depicted in figure 4.1, including the device used.

4.1 Beaglebone Black rev 3C

Finding a suitable device for this implementation was a non-trivial task, mainly due to the lack of public information or documentation about the inner workings, especially the registers, of various IoT devices. Before deciding on the Beaglebone Black, other devices such as Raspberry PI, Banana PI or the Arduino boards were investigated, just to name a few. After spending a lot of time researching these boards in more detail, including the official forums, no progress was made towards a potential DRAM PUF implementation.

What makes the Beaglebone Black stand out, is that it is an open-source community-driven IoT board, while also having an affordable price range. The design and schema of the board are freely available on a publicly hosted git repository [13]. The board uses a Arm Cortex-A8 (AM3358) from Texas Instruments, which is documented in detail, including all the registers and memory addresses, on the CPU's official website [18, 17].

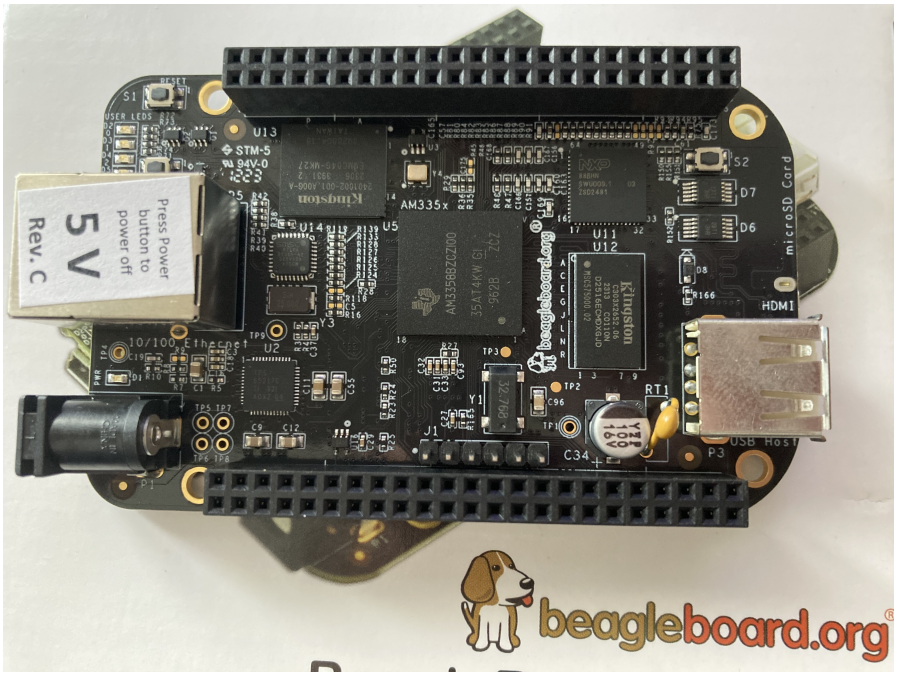


Figure 4.2: Beaglebone Black rev 3C

At the time of writing, the implementation was made on the latest available Operating System image running Debian version 11.7. It is important to take this into account, as newer updates may break the implementation due to the possibility of new optimizations being used when compiling the PUF kernel modules on the device. Originally the implementation was made for the Debian version 9.5, which is the version that the device

shipped with, and after upgrading to the latest version, the implementation for the kernel modules broke and needed to be adjusted.

4.2 Linux Kernel Modules

The Decay-based DRAM PUF is implemented as a set of Linux kernel modules. With this approach the kernel modules can be loaded and unloaded at any point during the runtime of the device to utilize the PUF for authentication purposes.

4.2.1 PUF Memory Reservation

The first kernel module is more straightforward, as the only task is to allocate a contiguous block of memory of the desired size, where the size to allocate is configurable, that will be used by the PUF kernel module, so that no other process will access the PUF memory region. On the BeagleBone Black it is possible to allocate contiguous memory of up to 8MB, which is plenty of DRAM cells that can be leveraged by the PUF. The kernel module was created for rapid prototyping, as some of the other options would be memory ballooning used by Xiong et al. [40, 41] or a custom configuration of the Linux kernel.

4.2.2 Decay-based DRAM PUF

When loading the PUF kernel module, the start address and size of the allocated PUF memory region are passed as inputs. The PUF kernel module exposes its interface by registering it as a miscellaneous device linked to the Linux kernel, making it accessible to any user space program running on the device until the kernel module is unloaded.

```
static const struct file_operations puf_fops = {
    .owner   = THIS_MODULE,
    .open    = puf_open,
    .read    = puf_read,
    .write   = puf_write,
    .release = puf_release,
};

static struct miscdevice puf_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DEVICE_NAME,
    .fops  = &puf_fops,
};
```

Listing 4.1: PUF device operations

The PUF interface consists of four functions:

Open. When a user space program opens the PUF device, two conditions are checked:

1. Is the PUF in warm up phase ?
2. Is the PUF already being used by another user space program?

Only after both of these conditions are false the user space program can proceed with using the PUF, otherwise it will be rejected.

Write. After the Open operation succeeds, it is expected from the user space program to write the enrolled DRAM cells. The first write is treated as a special case that stores any incoming data into the runtime state of the kernel module, while also disabling the refresh of the DRAM cells for the PUF memory region. Any write operations from a user space program, initializes the PUF memory region with the challenge, this includes also the first write.

Read. Read operations by default return a zero response. It is only after the first write was made by the user space program that the read operation will return a non zero response. The passed data from the first write operation is consumed to figure out which DRAM cells to read, that are then in turn passed to the error correction step, after which the reconstructed bits are returned as the response.

Release. When the user space program is done with the PUF the close operation is executed. This clears the internal state of the kernel module from the data passed by the current user space program and enables the DRAM refresh again, so that subsequent utilization of the PUF starts with a clean state.

The internals of the PUF kernel module are also illustrated in figure 4.3

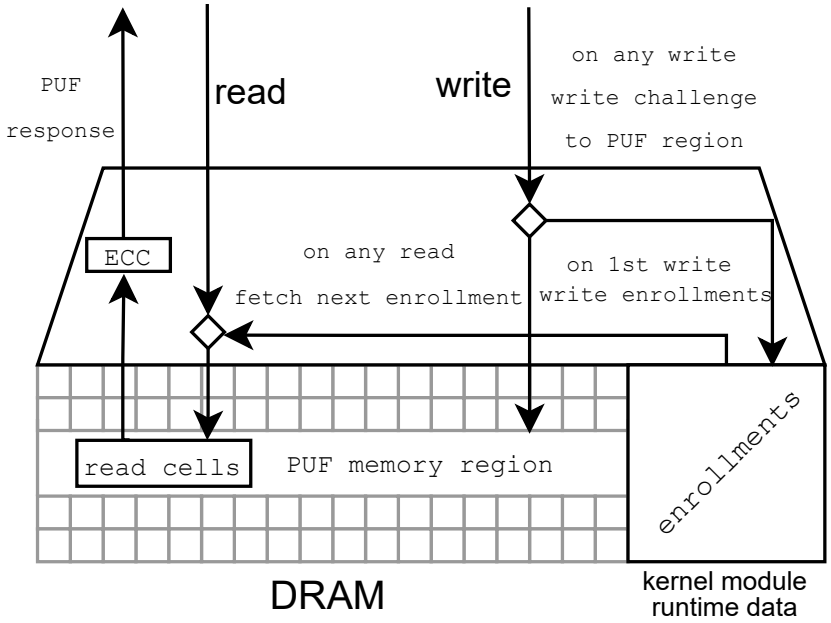


Figure 4.3: Overview of the DRAM PUF

4.2.3 Disabling DRAM Refresh

DRAM Refresh is disabled for the memory region reserved for the PUF by writing to special registers of the device, namely the $SDRAM\_REF\_CTRL$ register. By writing a specific value to the register, DRAM refresh is turned off, causing the DRAM cells to decay. Since the BeagleBone Black has only one memory interface controller, namely $EMIF0$, refresh is disabled for the entire connected DRAM. It only takes about ≈ 2 seconds for the runtime memory to enter a corrupted state, at which point the only option is to reboot the device.

A custom refresh of the entire DRAM is part of the implementation of the Linux kernel module. Xiong et al. [40] described how a custom refresh of the cells can be implemented. It is sufficient to access a single cell, either with a write or a read operation, and all other DRAM cells that are connected to the same word line and form a row are refreshed.

The implementation of such a custom refresh is not a trivial task, as the row size must be calculated correctly. Depending on how large the error in the calculation is, some DRAM cells could otherwise not be refreshed, which in the worst case could lead to proper operation of the device or in the best case to random runtime crashes. The $SDRAM\_CONFIG$ register provides all the necessary values for calculating the correct row size.

SDRAM connected to the BeagleBone Black has the following parameters:

1. 1024-word page (10 column bits)
2. 2^{16} rows (16 row bits)
3. 8 banks
4. 16 bit bus width (word size)

The row size is simply calculated with:

$$1024 * word\_size = 2048$$

The refresh operation is carried out every $\approx$ 100ms by performing a single read per each row of the connected SDRAM, except the reserved PUF memory region.

```
1  void sdram_refresh(struct work_struct *work) {
2      uint32_t discard, offset;
3      uint32_t puf_beg, puf_end;
4
5      struct delayed_work *dwork;
6
7      dwork = to_delayed_work(work);
8      schedule_delayed_work(dwork, msecs_to_jiffies(REFRESH_PERIOD));
9
10     puf_beg = puf_phys_addr;
11     puf_end = puf_phys_addr + puf_size;
12
13     for (offset = DDR_BASE; offset <= DDR_END; offset += ROW_SIZE) {
```

```
14          if (offset >= puf_beg && offset < puf_end) {
15              offset = puf_end - ROW_SIZE;
16              continue;
17          }
18          phys_r32(offset, &discard);
19      }
20  }
```

Listing 4.2: Custom SDRAM refresh

4.2.4 Creating a Response

A single decay timeout enrollment consists of 3 components. Pointers to the DRAM cells from which a single bit is to be read. Each pointer is an integer that identifies the row and column of the PUF DRAM region of a particular cell to be read. This is followed by parity data generated by the Reed-Solomon encoder when the enrollment was created. Finally, a single byte indicating how many parity bytes will be used to reconstruct the message. This implementation uses a configurable error correction, so this single byte is needed to know how many parity bytes to expect, otherwise it could be omitted if the error correction was fixed at a certain number.

The process of constructing a response when a user space program initiates a read to the PUF interface is illustrated in detail in figure 4.4. The enrollment data is consumed from the data passed in during the first write of the user space program. DRAM cells are read and the resulting values are stored in an array. Parity bytes are appended to the array and the resulting sequence of bytes is then passed to the Reed-Solomon decoder which corrects any unstable bit flips and returns the result as the PUF response. Subsequent reads repeat the process for the remaining enrollments.

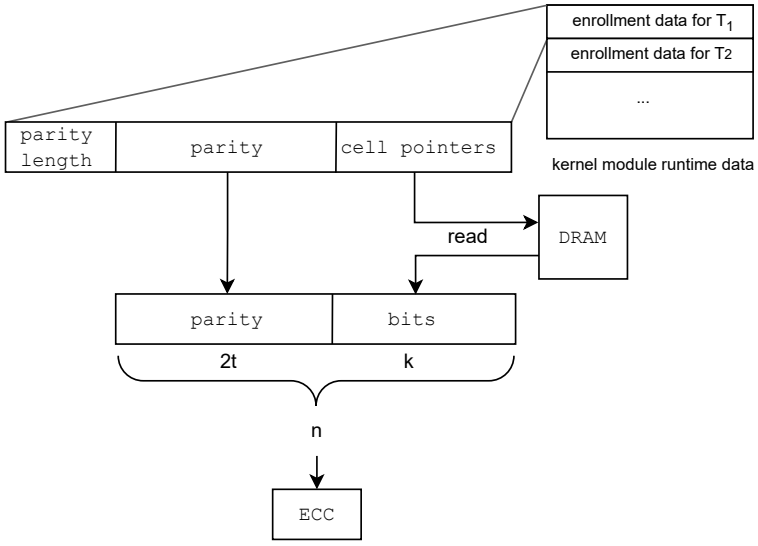


Figure 4.4: Enrollment for a single decay timeout

4.2.5 Warm Up Phase

When the PUF kernel module is loaded for the first time, it enters a warm-up phase that lasts $\approx 10\text{min}$[2]. If the device is not connected to power for a longer period of time and is then booted, it is not possible to obtain a stable response immediately after booting. It is not known exactly why this is the case, but after the selected timeout the PUF can be used again. We do not consider this a limitation, as this period is relatively short when using the kernel module over long periods of time.

4.3 Enrollments

Once the PUF kernel module is running on the BeagleBone Black, the DRAM cells of the allocated memory area must be enrolled. Enrollment here refers to the process of determining the cells against which a patched user space program will authenticate. The cells are enrolled against a chosen challenge that is built into the PUF kernel module. The DRAM cells are initialized with all 0s, which then decay and the bits that flip are collected. It is possible to write other challenges such as all 1s, a checker pattern or other challenges into the PUF memory area before letting the cells decay.

It must be known at what times the program queries the PUF for authentication in order to enroll the DRAM cells. Consider the following decay times $[10s, 20s, 30s]$, which indicates when the program will authenticate itself to the PUF. To enroll the DRAM cells, the PUF kernel module can be used to start the decay of the cells in the PUF memory region and after the first timeout of 10s, immediately abort the decay and read out the DRAM contents of the PUF memory region into a binary file. To enroll the cells for the next decay time, the PUF must be reset, which means that the decay of the cells must start again from 0 seconds, since reading out the DRAM contents implicitly refreshes the cells, otherwise reading the DRAM contents at 20s without a reset would result in incorrect cell enrollment. This process is repeated until all decay times have been recorded.

Having collected the DRAM contents for all decay times, the binary files need to be processed to extract the cells that will be used when the patched user space program authenticates to the hardware. To achieve a higher level of confidence, the process just described should ideally be repeated several times so that we can determine which cells have the same bit flips across multiple measurements. In this implementation, we use a simple method to extract the bit flips, which are then used to create the enrollments. The method is described in algorithms 1 and 2, it was used by Xiong et al. [41] and is based on and adapted from their work.

[2]This number was obtained by trial and error by examining the responses of the PUF

```
Data: Timeouts
Result: DRAM cell pointers for each Timeout
for each pair in Timeouts do
    current ← collect_measurements(current timeout);
    previous ← collect_measurements(previous timeout);
    for each byte in (PUF size bytes) do
        b1 ← common bit flips across all current measurements;
        b2 ← common bit flips across all previous measurements;
        b3 ← extract bit flips in b1 but not in b2;
        for each non-zero bit in b3 do
            row, column ← position of the bit within the PUF region;
            add (row,column) for current timeout;
        end
    end
end
```

Algorithm 1: DRAM cell extraction

To extract DRAM cells for each decay time, we essentially look for bit flips that only occur in that particular timeout, and ignore the bits that also occurred in the preceding timeout. To create enrollments, these extracted cells are then used in the enrollment phase described in Algorithm 2

```
Data: Timeouts, DRAM cell pointers for each Timeout, t
Result: Enrollments
for each timeout in Timeouts do
    current ← current bit flips;
    next ← future bit flips;
    r ← generate random value;
    ECC ← Reed Solomon(r, 2t);
    cells ← ∅;
    for each bit in r do
        if bit ≠ 0 then
            random cell from current → cells;
        end
        else
            random cell from next → cells;
        end
    end
    (timeout, r, ECC, cells) → Enrollments;
end
```

Algorithm 2: DRAM cell enrollments

In the enrollment phase, for each timeout a random number is generated which bits

are then encoded using the obtained pointers to cells at that specific decay timeout. The generated random number is passed to the Reed-Solomon encoder to generate the requested amount of parity bytes for error correction. When these cell pointers along with the parity bytes are then given to the PUF kernel module, it will use them to reconstruct the generated number. To make the response valid only during a specific time interval and satisfy the dynamic PUF property described in Chapter 3, we also use pointers for DRAM cells that will decay in the future.

Example: Given a program that will query the PUF at decay timeouts $[10s, 20s, 30s, 40s]$. If the bits for the response at e.g. 10s were partly enrolled with cells that will decay at 30s, then when the PUF is queried with these enrolled cells before or after this time interval $[10s, 30s]$, it will return an invalid response.

The collected enrollments are then used in the LLVM out-of-tree pass to patch a program to utilize the PUF.

4.4 LLVM pass

The LLVM out-of-tree pass takes an arbitrary LLVM-IR input and transforms it to use the collected enrollments to query the PUF kernel module at runtime for authentication. The patched LLVM IR will have tamper detection and authentication against a Decay-based DRAM PUF. Double compilation process is used to catch functions that are either optimized away because they are not used or that are inlined after all optimizations. The same transformations are repeated in both runs; only in the second run feedback is given from the compiled binary as to which functions are still present as well defined and will be processed on the second run of the LLVM pass. The transformation of the LLVM IR is done in the following steps:

Collect Functions. A single LLVM-IR module is parsed where functions that are well defined within that module are collected. Functions that are only declared in this module are ignored. These are functions that are linked dynamically or possibly via a library in the compilation process, etc. We do not have the definitions for these functions and therefore we cannot patch them. The LLVM pass allows for a more narrow collection of functions that contains a certain prefix by setting a flag when running the LLVM pass, which may be useful if only a specific path within the LLVM-IR module will utilize the PUF.

Replace Function Calls. A lookup table is created where each collected function has a single entry. All of the functions calls within the LLVM-IR module are replaced by indirect calls via this lookup table. The table holds functions addresses that are initially all zero and that will be computed at runtime of the program.

Function address Calculation. Within the LLVM-IR module, we need to correctly identify at which location a particular function address needs to be calculated. These identified locations are patched with the necessary calculations to rebuild the lookup

table. Let's look at a simple example from Figure 4.5, where we try to determine at which location in the call graph the address for function C must be calculated. First, we need to determine all possible entry points[3] into the IR module. From each of the entry points, we find a path that leads to function C. Since there can be multiple paths from a single entry point to the target function, we consider only those nodes that all paths have in common, in which the function address computation can be inserted so that they do not have to be inserted individually into the sub-paths. At any point along this path, we can perform the calculation for the address of C, e.g. from the entry point *main* at any location along the path main → A → B. For *E* the path would be E → D. It is also possible that the paths from several external points cross at the same location. Therefore, when selecting a location, we must also perform a simple check to see whether a calculation has already been inserted at the chosen location in the call graph. However, there is a catch. The catch is that each of the functions in the call graph can have its address taken and can be called indirectly from a library or within the module itself. Therefore, we must also consider functions whose address is taken as a potential entry point into the LLVM-IR module itself. The function B can in this case also be an entry point and the path for calculating C would be simply B as it is the direct predecessor. As can be deducted from this example an entry point can be part of a path of another entry point to the target function, if it is an indirect function call. This means that the calculations for the same function C, may be inserted at multiple locations in a single path, depending on how many functions in the path have their address taken.

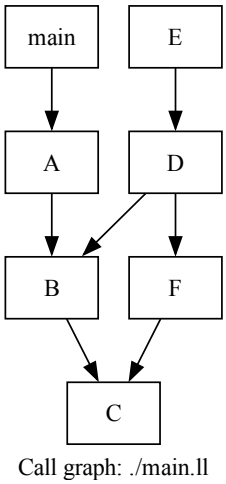


Figure 4.5: Example Call Graph

[3]In this context, an entry point is any function that is at the root of a call graph. No other function calls this function, in the example in Figure 4.5 this would be function *main*, *E*

The code that is inserted into a function to reconstruct the lookup table by calculating the addresses of functions is shown in Listing 4.3. Blocking code is inserted that waits for a specific PUF response and continuously calculates the checksum of a randomly selected function until the PUF response is obtained. The inserted code may also wait on multiple PUF responses and calculate addresses for multiple functions, if it was determined that the calculations need to be inserted at the same location in the call graph. The address is calculated as follows:

$$PUF\_response + Reference\_Value + Checksum$$

The reference value, described more in detail in section 4.5.2, is calculated as:

$$Reference\_value = Function\_Address - PUF\_response$$

```
1    ...
2    sum = 0x0;
3    while (puf_response == 0) {
4     sum += checksum(...);
5    }
6    lookup_table[function] = puf_response + reference_value + sum;
7    ...
```

Listing 4.3: Function Address Calculation

In the LLVM pass, only a marker[4] is left in the code for the reference value, via inline assembly. This marker is replaced with the correct value in the patching of the compiled binary. This is because it is hard to know at the LLVM IR level which address the function will have. A feedback loop would need to be created with the LLVM pass and the compiled binary, but the generation of the instruction in the LLVM IR would also need to be stable enough to produce the same exact code. Therefore, the simpler approach of a marker was chosen.

The inserted address calculations do not wait for randomly selected PUF responses. Instead, within the LLVM pass they are assigned specific PUF responses which they wait for. The exact PUF response used depends on the location of the inserted calculations in the call graph. The computations that are inserted closer to the entry point use PUF responses that are received earlier. The further away the inserted code is from the entry points, the later the PUF response will be used to reconstruct that part of the lookup table. For example, if the program in Figure 4.5 was patched to authenticate using the PUF responses at timeouts $[10s, 20s, 30s]$, code would be inserted in *main* and E that calculates the function addresses in the lookup table for functions A and D using the PUF response at 10s. In functions A, D, the PUF response at 20s is used to reconstruct the address for functions B, F. And within B, F, the PUF response at 30s would be used to calculate address of C. In a larger call graph, a single PUF response would be used to reconstruct a far larger number of functions instead of just the next one as, in this small example.

[4] A marker is simply a placeholder with a specific value that is searched for and replaced with the desired value. The replacing of the markers is done on the compiled binary

Collect PUF responses. A global array is created in which the responses from the PUF queries are stored, which are then used in the function address calculations. Together with this global array, a global 32-bit integer is created that is used to calculate the index of where the next PUF response should be placed in the global array. Querying the PUF at runtime at the desired time is executed in a separate thread that is added in the LLVM pass. The global variable for index calculation is only used within the spawned thread and the inserted checksumming code, it is not used within the inserted function address calculations.

Embed Enrollments. LLVM-IR instructions are inserted to pass the collected enrollments for the DRAM cells to the PUF interface. This write operation is the first to be performed when the program is executed so that the decay of the cells begins as early as possible.

Checksumming. The functions collected in the first step are then further modified to include checksum instructions. In this step, the filtering of the functions via the available flag in the LLVM pass is ignored, so that all well-defined functions are considered. Similar to how the reference values for the function addresses are handled, the checksum code inserted uses markers for the start address and the size of the byte sequence to be checked. The how is explained in more detail in listing 4.6 in section 4.5.2. The number of checksum functions generated is configurable and scales linearly with the number of collected functions to patch.

4.5 Tamper Detection

The LLVM pass uses two methods to detect tampering, namely call graph scrambling and checksumming. As soon as a tampering is detected, these two methods react in different ways to prevent the attacker from continuing to execute the program.

4.5.1 Checksumming

If the checksum detects that newly inserted instructions have been inserted or existing instructions have been changed inside a function, it will simply shift the positions of the PUF responses so that the inserted code for the function address calculations will calculate the wrong addresses. To determine the position of the next PUF response into the global array of PUF responses a global variable is used by the spawned PUF thread. The inserted checksumming code, simply calculates the checksum and adds the result to this global variable. The checksumming is implemented based on the idea described by Collberg and Nagra [9], where the result of the checksum is zero, by interpreting the instructions as a linear modular equation mod 2^{32}. Thus, adding the result of the checksum will only result in undefined behavior if tampering is detected as the hash will not equal zero.

```
1 uint32 puf_responses[...];
```

```
uint32 offset = 1;
    ...
function() {
    offset += checksum(start, size);
    ...
}
```

Listing 4.4: Checksumming

To make the checksum of instructions of a function equal to zero, a marker is added by wrapping it in an opaque predicate. As explained in chapter 3, this is used to insert side-effect code into a function that is not optimized away. The marker will be replaced when patching the binary with the solution of a constructed modular linear equation of the to be checksummed bytes.

```
    ...
    if (opaquely-false predicate) {
      asm volatile("marker:.word 0x00010203");
    }
    ...
```

Listing 4.5: Insert Placeholder for Checksum

The opaquely false predicate also ensures that the inserted assembly code is not executed. When this placeholder value is patched, it is replaced by another 32-bit integer and it is ensured that if this integer is a valid instruction, it will not be executed.

4.5.2 Call Graph Scrambling

The patched LLVM IR has its call graph completely destroyed. The call graph is reconstructed at runtime. Each PUF response recreates parts of the call graph. If tampering is detected by the PUF not returning the correct responses at specific time or the checksum returns a non-zero result, a completely different call graph will be reconstructed that most likely will crash the program. The function address is computed given:

$$PUF\_response + Reference\_value + Checksum$$

As mentioned a marker is inserted for the reference value. The difference between this marker and the one used for making the checksum of the function equal to zero is that this value will be used directly in the code, however to also ensure that the reference value is not executed in case it is a valid instruction we need to exclude it from the control flow. To achieve this, a simple technique is used. A separate function is created that only contains the necessary assembly instructions. This function is not called anywhere. To use the reference value present in the assembly block, it will need to be linked against during compilation time. This can be achieved by creating a global variable with external linking with the same name as the label inside the assembly block. This is shown in listing 4.6 using a high level language instead of LLVM IR.

```
    ...
    // after compilation this will have the address
    // of the label inside the assembly block.
    extern uint8 reference_value;
    // The reference value will be accessed
    // by this pointer.
    void *label_address[] = {&reference_value};

    void unused() {
      asm volatile("reference_value:.word 0x00010205");
    }
    ...

    lookup_table[function] =
      puf_response + *(uint32*)(label_address[0]) + checksum;
    ...
```

Listing 4.6: Inline Assembly for Marker

The same technique is used to store the start address and size, that are then retrieved by the inserted checksum code.

4.6 Obfuscation

Only a single checksum function is used in the LLVM pass, that is shown in a high level language in listing 4.7.

```
uint32 checksum(uint32 start, uint32 size, uint32 c) {
    uint32 block_start = start & 0xffff0000 | size & 0x0000ffff;
    uint32 block_size  = size & 0xffff0000  | start & 0x0000ffff;
    uint32 checksum = 0;
    while (block_size > 0) {
      checksum     = c * (*((uint32*)block_start) + checksum);
      block_size  -= 1;
      block_start += 4;
    }
    return checksum;
}
```

Listing 4.7: Hash Function

The start address and the size of the block to be checksummed are calculated at runtime by a simple byte swap[5] of the high and low order bits, so that the actual addresses cannot be pattern matched directly, which would make the process of locating the block to be checksummed easier.

A separate checksum function is inserted for each of the identified functions that are patched in the LLVM pass to prevent multiple calls to the same function, which can be easily identified. This can lead to hundreds of checksum functions having to be inserted which compile to the same instructions. To prevent pattern matching, the obfuscation

[5] when the markers in the binary for the start and size of the bytes to be checksummed are patched, the bits are swapped

techniques mentioned in chapter 3 are used. The instructions are replaced according to the rules described in Table 4.1, variable r denotes a random integer. The rules were collected from various code obfuscators[20, 12, 11]. In addition, control flow flattening based on jump tables is used, as shown in figure 3.8. For opaque predicates used throughout the LLVM pass, the ones described in Table 4.2 are used. An opaquely false predicate can be obtained by simply inverting the condition. The predicates work $\forall x \in \mathbb{Z}$. The use of obfuscation results in different variants of the same checksum function.

AND Substitution $a = b \wedge c$
$a = (b \oplus \neg c) \wedge b$
$a = \neg(\neg b \vee \neg c) \wedge (r \vee \neg r)$
$a = (\neg b \vee c) - (\neg b)$
OR Substitution $a = b \vee c$
$a = (b \wedge c) \vee (b \oplus c)$
$a = (b \wedge \neg c) + c$
$a = [(\neg b \wedge r) \vee (b \wedge \neg r) \oplus (\neg c \wedge r) \vee (c \wedge \neg r)] \vee [\neg(\neg b \vee \neg c) \wedge (r \vee \neg r)]$
Subtraction Substitution $a = b - c$
$a = b + (-c)$
$a = b + r;\ a = a - c;\ a = a - r$
$a = b - r;\ a = a - c;\ a = a + r$
Addition Substitution $a = b + c$
$a = b - (-c)$
$a = -(-b + (-c))$
$a = b + r;\ a = a + c;\ a = a - r$
$a = b - r;\ a = a + c;\ a = a + r$
$a = (b \oplus c) + 2 * (b \wedge c)$
$a = (b \wedge c) + (b \vee c)$

Table 4.1: Supported Operation Substitutions

Opaquely True predicates
$(x \wedge 1 == 0) \vee [3 * (x^2 + x)] \bmod 2 == 0$
$(x \wedge 1 == 1) \vee (x^2 + x) \bmod 2 == 0$
$(2x + 2) * (2x) \bmod 4 == 0 \vee (x^2 + x) \bmod 2 == 0$
$3 * (x^2 + x) \bmod 2 == 0 \wedge (x^2 + x) \bmod 2 == 0$
$(x^2 + x) \bmod 2 == 0 \wedge (2x + 2)(2x) \bmod 4 == 0$
$(x + x^3) \bmod 2 == 0 \wedge (2x + 2)(2x) \bmod 4 == 0$

Table 4.2: Opaquely True Predicates

4.7 Patching the binary

Patching of the compiled binary is done as the last step before it is deployed to the BeagleBone black. The patching itself is a relatively simple process. It looks for specific markers that were left during the execution of the LLVM pass and replaces these markers with desired values. Only three types of markers need to be identified within the compiled binary that need to be replaced.

The First marker, inserted as an additional instruction within each function that was patched in the LLVM pass, is replaced to make the checksums equal to zero. The resulting value that replaces this marker is the result of the extended euclidean algorithm applied on the modular linear equation that was constructed by interpreting the instructions as 32-bit integers, as explained by Collberg and Nagra [9].

$$c^{(k+1)-z} * x_z + (\sum_{j=0;j\neq z}^{k} c^{(k+1)-j} * x_j) \equiv 0 \ (mod\ 2^{32})$$

where:

- x_j is a 32-bit integer value of an instruction.
- c is an odd constant.
- x_z is the inserted marker for which the equation is solved.

Replacement of the markers that are used for the start address and size of the to be checksummed block of bytes is done as described in algorithm 3

$\phi \leftarrow$ all functions;
for *each marker in Checksum Markers* **do**
  if ϕ *is empty* **then**
    $\phi \leftarrow$ all functions;
  end
  (address, size) $\leftarrow$ choose and remove a random function from ϕ;
  swap and interchange high-order and low-order bits of *address*, *size*;
  replace the marker in the compiled binary with (address, size);
end

Algorithm 3: Checker Network Creation

Replacing the markers will gradually create a checker network, where each block of bytes to be checksummed is checked by another block, as shown in the figure 4.6. Depending on whether the LLVM pass has been configured to spawn multiple checksum functions, each node in the checker network may be covered more than once.

Finally, the last marker left to replace are the reference values. In the LLVM pass each calculation that reconstructs a single function address in the lookup table was assigned a PUF response based on the location in the call graph. This same data is consumed when

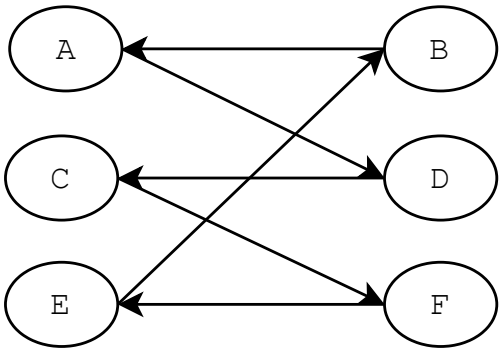


Figure 4.6: Checker Network

replacing these markers. We find the address of the function in the TEXT segment of the compiled binary and calculate the reference value according to:

$$Reference\_value = Function\_Address - PUF\_response$$

And replace the marker with that value. All of the described changes to the resulting binary are visualized in Figure 4.7

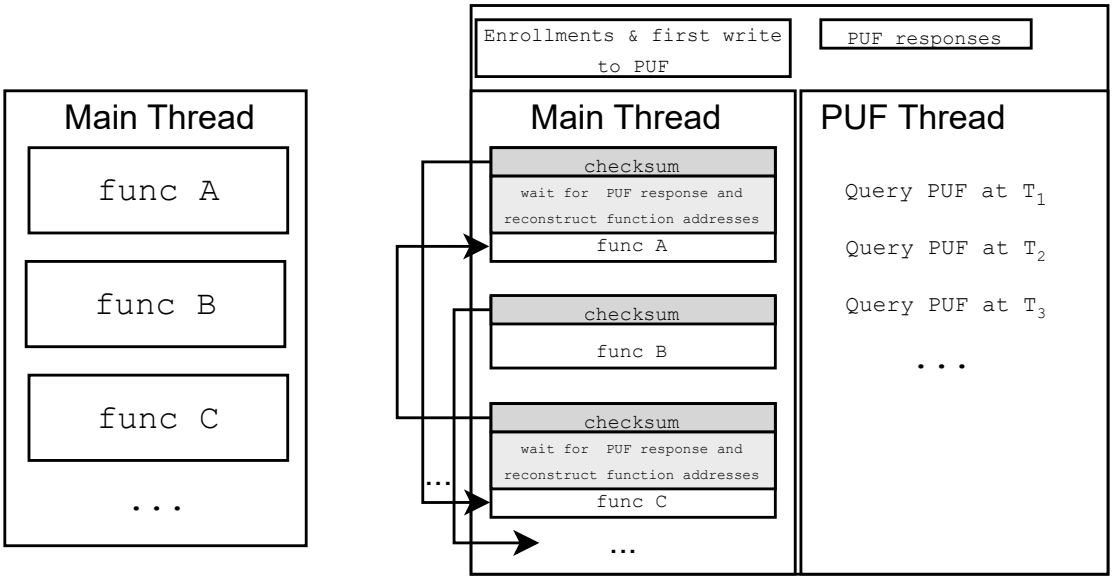


Figure 4.7: Before and After

5 Evaluation

This last chapter is dedicated to the evaluation of the implemented hardware-software authentication scheme. All results in this chapter were obtained by evaluating the scheme on two different BeagleBone Black boards. At the end of the chapter there is a discussion of comparing the differences to the previous work.

5.1 Decay Based DRAM PUF on the BeagleBone Black

A total of 50 measurements were taken at different decay times T = {10s, 20s, 30s, 40s, 50s, 60s} on two different boards. In the measurements we mainly were interested in how similar two measurements from two different boards are ($Jaccard_{Inter}$) and how similar two consecutive measurements on the same board are ($Jaccard_{Intra}$). An ideal PUF would have the former close to 0.0 and the latter close to 1.0. To evaluate the similarities between two sets of DRAM cells, indices of the bit flips were identified within each of the measurement done and compared the resulting two sets using the Jaccard Index:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The results of the measurements performed are shown in Table 5.1, where each value is the average of all measurements performed on two different boards. The calculations, i.e. entropy, bit flips, etc., were performed as close as possible to the description from the paper by Schaller et al. [32]. The results obtained on the BeagleBone Black are similar to the results obtained on the PandaBoard and the Intel Galileo Gen 2 used in their work. All measurements obtained for $Jaccard_{intra}$ and $Jaccard_{inter}$ are visualized in Figure 5.1 to show that the decay of the DRAM cells on the BeagleBone exhibits PUF behavior.

Time	Entropy	$J_{intra/avg}$	$J_{intra/min}$	$J_{inter/max}$	$DecayCells_{avg}$
10s	7.715×10^{3}	9.276×10^{-1}	6.786×10^{-1}	0.000	4.654×10^{2}
20s	1.190×10^{5}	9.534×10^{-1}	8.130×10^{-1}	1.225×10^{-4}	9.761×10^{3}
30s	4.831×10^{5}	9.517×10^{-1}	8.211×10^{-1}	4.468×10^{-4}	4.900×10^{4}
40s	1.064×10^{6}	9.738×10^{-1}	8.980×10^{-1}	9.963×10^{-4}	1.251×10^{5}
50s	1.871×10^{6}	9.725×10^{-1}	8.860×10^{-1}	1.983×10^{-3}	2.494×10^{5}
60s	2.756×10^{6}	9.808×10^{-1}	9.221×10^{-1}	3.183×10^{-3}	4.056×10^{5}

Table 5.1: PUF Measurements with size 16MB

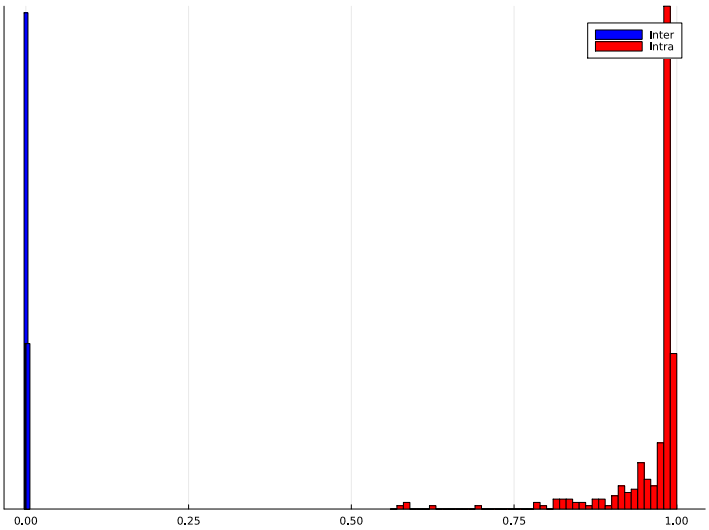


Figure 5.1: Histogram for $Jaccard_{Intra}$ and $Jaccard_{Inter}$ values from multiple PUF instances with size 16MB, on the BeagleBone Black

The experiment with 50 measurements was repeated several times. In each run, the value of $Jaccard_{intra/min}$ for the 10s decay time fluctuated, e.g. for one of the measurements it was only ≈ 0.18. For the 20s decay time, it was at least above 0.6 and mostly above 0.7. For 30s and more, the jaccard index mostly fluctuated between 0.8-0.9. The average $Jaccard_{intra}$ index for all decay times was mostly 0.9. The experiments performed on two different boards showed that the similarities, i.e. $Jaccard_{inter}$, between the decayed cells are absent and close to 0, while the $Jaccard_{intra}$ is close to 1, which is close to an ideal PUF.

When performing the measurements, it occasionally happened that the number of decayed cells within a measurement was lower than in other measurements with the same decay time. The results shown in the table 5.1 were obtained after repeating the experiment five times with all 50 measurements, showing only the results of the last experiment performed in which this one-off measurement did not occur. This one-time measurement occurred in three out of five experiments, in at least two decay times, and resulted in an overall lower Jaccard index. It is unclear how this measurement could be deterministically repeated to find out how to avoid it.

Further experiments were conducted because using this measurement during the DRAM cell enrollment process would result in PUF responses that would most likely fail without extensive error correction. For each measurement, the PUF kernel module is loaded and immediately enables decay of the cells. After sufficient time has elapsed, the kernel module is removed and the raw DRAM contents are read and processed. It has been found that the time elapsed between successive measurements has a direct effect on the rate of cell decay. The time between each measurement refers to the time during which the cells are in normal operation, i.e. the DRAM refresh is activated. For example, if we

were to take multiple measurements for the 20s decay time and run them back-to-back with no time in between, this would result in more total bit flips for the decay time, than if we used a timeout between each consecutive measurement. Turning off the DRAM refresh in quick succession and allowing the DRAM cells to decay multiple times for the same decay time will result in more total bit flips.

To verify this, experiments with decay times T = {20s, 30s, 40s, 50s} were performed on a 4MB PUF on a single device. A total of eight measurements were taken for each decay time, four with an 8 minute timeout between each measurement and the other four with no timeout in between. The same calculations as in table 5.1 were repeated with these experiments. The results are shown in table 5.2 and table 5.3.

Time	Entropy	$Jaccard_{intra/avg}$	$Jaccard_{intra/min}$	$DecayCells_{avg}$
20s	1.657×10^{3}	9.356×10^{-1}	8.818×10^{-1}	9.850×10^{1}
30s	7.264×10^{3}	9.744×10^{-1}	9.683×10^{-1}	5.020×10^{2}
40s	2.000×10^{4}	9.711×10^{-1}	9.648×10^{-1}	1.557×10^{3}
50s	4.371×10^{4}	9.621×10^{-1}	9.488×10^{-1}	3.783×10^{3}

Table 5.2: Measurements with using a 4MB PUF with 8 minutes between each measurement taken

Time	Entropy	$Jaccard_{intra/avg}$	$Jaccard_{intra/min}$	$DecayCells_{avg}$
20s	2.838×10^{3}	9.059×10^{-1}	8.756×10^{-1}	1.777×10^{2}
30s	2.054×10^{4}	8.978×10^{-1}	8.871×10^{-1}	1.605×10^{3}
40s	7.599×10^{4}	9.046×10^{-1}	9.038×10^{-1}	7.143×10^{3}
50s	1.897×10^{5}	9.301×10^{-1}	9.206×10^{-1}	2.087×10^{4}

Table 5.3: Measurements with using a 4MB PUF with no timeout

The difference for the Jaccard index is minimal between the two experiments, but the difference can actually be seen by looking at the entropy and number of decayed cells in the two tables. For the 20s time decay in the first experiment, where there was an 8 minute timeout between each measurement, the number of decayed cells was only $\approx$ 98 on average. In contrast, when no timeout was used between each measurement, the number of cells that decayed was $\approx$ 177 on average, almost 2x more cells. This metric has also been visualized for all decay times in Figure 5.2, where the trend can be clearly seen. Furthermore, the decayed DRAM cells for the same decay time of the two experiments were compared using the Jaccard index to calculate how similar they are, and the results are visualized in Figure 5.3.

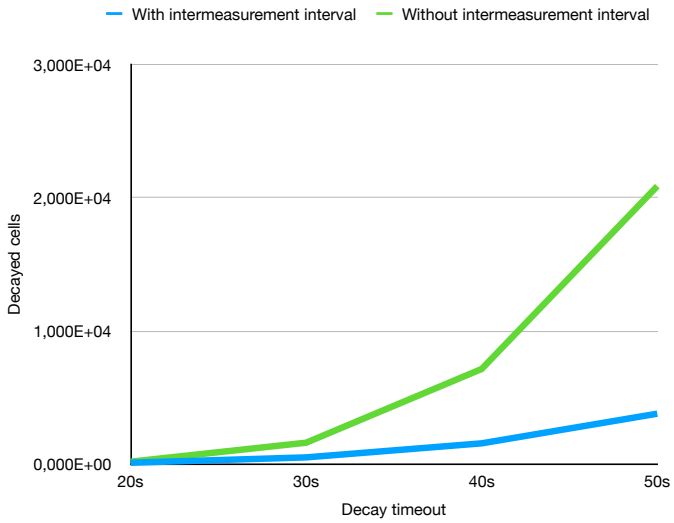


Figure 5.2: Difference with using an intermeasurement interval

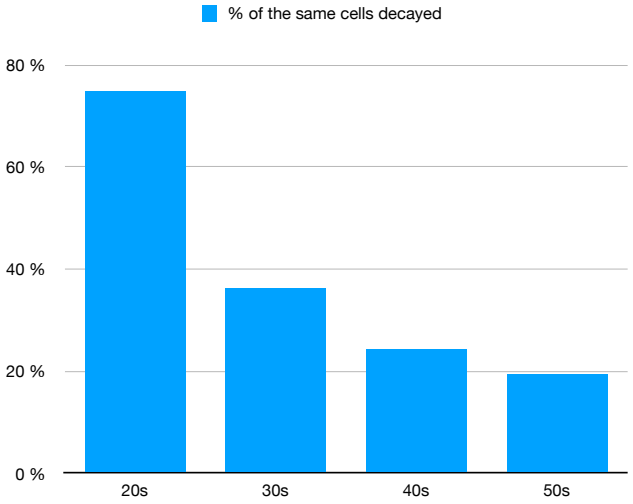


Figure 5.3: Percentage of the cells which are common to both experiments

The same DRAM cells that decayed in the experiment with an 8-minute intermeasurement interval were also mostly present in the second experiment without a timeout. As the number of bit flips in the no timeout experiment gradually increased, the similarities between the two sets decreased for each increasing decay timeout.

5.1.1 Conclusion

The way in which DRAM cells are enrolled has a direct effect on the decay of the cells themselves. If no timeout is used between two consecutive measurements, the number

of decayed cells will be greater than if a timeout is used. This also has a direct effect on the reconstruction of the PUF responses in the PUF kernel module. For example, if we consider the decay timeout of 50s with 4 measurements taken without a timeout between each measurement, and we have enrolled cells from these measurements, when these enrollments are used to patch a program, they will be authenticated against cells that will only be present if the same decay timeout is repeated at least as many times as the number of measurements taken. This means that the program itself would have to be run several times in succession before it received the correct PUF response. This may not be the desired effect.

Furthermore, one could interpret the results shown in figure 5.3 as if one could combine the measurements from the two experiments, i.e. with and without intermeasurement interval, and get a more accurate and stable result both when the program is used repeatedly and when it is used only occasionally. This is not the case. The figure only shows that there are cells that are present in both experiments, but what the figure does not show us is the time when these cells decay. It turns out that when the process of turning off/on the decay of cells is repeated consecutively for the same timeouts, the cells will decay faster. This means that the same cells that were decaying at 40s would instead already be decayed at 30s. This means that the dynamic property of the PUF is violated. Thus, when a patched program is run consecutively with the enrolled cells from measurements with an intermeasurement interval, it will happen that the cells that are supposed to decay at a later time are already decayed at an earlier time when the response is reconstructed, and thus an invalid response is reconstructed, which leads to the reconstruction of the wrong call graph of the program. And if a program was patched with enrolled cells from measurements without an intermeasurement interval, it will happen that the reconstruction of the PUF response will fail, until the program is re-run enough times.

In order to obtain stable PUF responses that work across device restarts but also while the device is powered up, 4 measurements were taken for the decay times T = {20s, 30s, 40s, 50s}, and during the DRAM cell measurements a timeout of $\approx$ 10 minutes was used between each measurement taken. This timeout was also used during the warm-up period in the PUF kernel module to ensure that the device would run for at least 10 minutes if the device was booted after an extended period without power. After the timeout, the PUF could be queried any number of times during the day. However, if the program was re-run repeatedly, after $\approx$ 6-7 re-runs the PUF response would fail at 30 seconds because the cells that should have decayed at 50 seconds had already decayed at 30 seconds. However, if some time elapsed between each re-run of, say, 30-40 seconds, the program could be executed multiple times with this timeout in between without any problems. To achieve better stability over frequent but not fast repetitions of reconstructing the PUF responses at the same decay times, an error correction of up to 8 bits, or 25%, was used for 32-bit PUF responses. If a lower error correction had been used, the successive re-runs of the program without a timeout in between would have failed sooner.

When cells are decayed repeatedly with the same decay times, a change in temperature may cause the cells to decay more quickly leading to more bit flips. All evaluations

done for this work were done at room temperatures, other temperatures and their effects were not investigated due to lack of equipment. Temperatures were collected using the on-board bandgap temperature sensor that measures the case temperature, which was stable at an average of 60°C. During the experiments where measurements for the same decay times were consecutively repeated without a timeout, the bandgap sensor was on average at 65°C, which, based on what was shown by Xiong et al. [40], explains how different temperatures directly affect how fast the cells decay.

Overall, this should only be a problem for a program that needs to be run repeatedly. For long-running programs such as servers or clients that communicate over the network or collect data from sensors, this should not be a problem, as they would run once, collect the PUF responses at the desired decay times, and not run again until possibly a device reboot.

5.2 Overhead of the Implemented Scheme

The hardware-software scheme was applied to the Rust program shown in listing 5.1 which opens a simple TCP server that encrypts the contents of each incoming message and sends back the SHA3 hash of the encrypted contents. This relatively short program uses more than a dozen dependencies, including indirect dependencies, that will result in various control flows which will end up being patched.

```
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};
use hex;
use sha3::{Digest, Sha3_256};
use aes::cipher::{generic_array::GenericArray, BlockEncrypt, KeyInit};
use aes::Aes128;

fn handle_client(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    loop {
        match stream.read(&mut buffer) {
            Ok(size) => {
                if size == 0 { break; }
                work2(&mut stream, &mut buffer[..]);
            }
            Err(_) => { break; }
        }
    }
}
fn main() -> std::io::Result<()> {
    std::thread::spawn(work1);
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    for stream in listener.incoming() {
        match stream {
          Ok(stream) => { std::thread::spawn(|| handle_client(stream)); }
          Err(e) => { eprintln!("Error accepting client: {}", e); }
        }
    }
```

```
29      Ok(())
30  }
31  fn work1() {
32      loop {
33          std::thread::sleep(std::time::Duration::from_secs(5));
34          println!("working...");
35      }
36  }
37  fn work2(stream: &mut TcpStream, msg: &mut [u8]) {
38      let key = GenericArray::from([8u8; 16]);
39      let cipher = Aes128::new(&key);
40
41      let mut hasher = Sha3_256::new();
42      for chunk in &mut msg.chunks_mut(16) {
43          let mut block = GenericArray::from_mut_slice(chunk);
44          cipher.encrypt_block(&mut block);
45          hasher.update(&block[..]);
46      }
47      let result = hasher.finalize();
48      let hex_result = hex::encode(&result[..]);
49      stream.write_all(hex_result.as_bytes()).unwrap();
50  }
```

Listing 5.1: Rust program on which the scheme was applied

The LLVM IR generated from the program and all its dependencies was linked into a single LLVM IR module consisting of ≈ 30k lines of code. After passing the LLVM IR module through the LLVM pass the resulting LLVM IR consists of ≈ 200k lines of LLVM IR code, an increase of ≈ 6x. The control flow graph of the program is too large to be shown[1]. The increase is mainly due to the insertion of the checksum functions and their verbose obfuscation. Since the LLVM IR module consists of a large number of functions, the LLVM pass inserted the same number of checksum functions.

5.2.1 Compilation time and Binary Sizes

When generating LLVM IR from Rust programs, dependencies from the standard library but also external dependencies are compiled and the LLVM IR is generated from them. When linked into a single unit, this results in a larger amount of functions that end up in a single LLVM IR module, then compared to a language such as C. This consequently means that a large number of checksum functions will be inserted. Generally speaking the more functions are present in a LLVM IR module the more of an impact will the LLVM pass have on the binary sizes and compilation time.

In table 5.4 the impact of the number of generated checksum functions on the binary size and compilation time is presented. It is important to note that the compilation was done using the available LLVM utilities by compiling the LLVM IR into object files which are then statically linked with all of the necessary libraries, and not using the official

[1]The control flow graph is stored in the repository and can be accessed at `https://github.com/Despire/puf-software-protection/blob/c201c7e0543eedff754e76bc05a512d549ef7203/images/graph.pdf`

Count	Compilation Time	Original Size	Patched Size	Size Increase
0	18s	4.3 MB	-	0%
1	23s	4.3 MB	5.1 MB	18.6%
2	28s	4.3 MB	5.6 MB	30.2%
3	33s	4.3 MB	6.2 MB	44.1%

Table 5.4: Impact of the number of generated checksum functions per function present in unmodified LLVM-IR module

rust tool chain, so the base size of the compiled binary is much larger at 4.3MB than if compiled using the official toolchain. This was done as the generated LLVM IR code was passed through an out-of-tree LLVM pass, after which it is not possible[2] to pass the LLVM IR back to the rust toolchain for further compilation. Nevertheless, the impact can still be deduced from these results.

The second biggest impact on the binary size is the inserted code for calculating the function addresses in the lookup table. Similar to the checksum functions, this is linearly proportional to the number of functions in the LLVM-IR module that have an entry assigned to them in the lookup table. The patched rust program consists of ≈ 500 well defined functions. Without checksums or the lookup table and its reconstruction, the size of the binary is 3.8MB. By inserting only the code to rebuild the lookup table the binary size increased from 3.8MB to 4.3MB, an increase of 13.1%.

The least impact on the binary size are the enrollments. The size depends on the number of decay times for which a response is requested from the PUF and how large the error correction for the responses will be, as it is configurable. For example, for 4 decay times and an error correction of 25% for each 32-bit PUF response at each decay time, a total of 644 bytes[3] are used to hold these enrollments in the binary.

5.2.2 Impact of the DRAM refresh

The BeagleBone Black has a single memory interface to which the entire 512MB DRAM module is connected. Therefore, when using the PUF kernel module, refresh must be disabled for the entire DRAM module. The DRAM refresh interval is typically 64ms, but can vary. The table 5.5 examines four user-defined refresh intervals where the memory does not end up in a corrupted state, and their CPU time during the user-defined refresh operation of the PUF kernel module.

The time it takes to refresh the whole 512MB is ≈ 64.92ms by reading a single word from each row of memory. So anything less than 64ms would be an overhead. With a 64ms refresh interval, the next refresh would start immediately after the previous refresh

[2]This in fact may be possible, but, seems to not be documented anywhere in the available public Rust documentations.

[3]In this case, enrollment for a single decay time consists of 1 byte indicating the length of the error correction bytes, since it is configurable, 16 two-byte integers used for Reed-Solomon error correction, and 32 four-byte integers where the pointers to the DRAM cells are encoded.

Refresh Period	CPU time
32ms	42%
64ms	40%
96ms	28%
100ms	27%

Table 5.5: Impact of the custom refresh of the 512MB DRAM module

finished, minimizing any decay errors so that the memory used by the OS does not end up in a corrupted state. Refresh intervals greater than 64ms are also viable, and in fact use almost twice as little CPU time. In the kernel module, a refresh interval of 96ms was chosen so that even if the custom refresh was run for longer periods of time, the memory wouldn't get corrupted while also using less CPU time overall.

5.2.3 Multiple Threads

If a program has more than one thread of execution, the patching scheme should also apply to successfully patch such programs. The LLVM pass introduces the following global state into the program. Two global variables are created to handle the PUF responses. A global array where responses are populated at specific times, and a global variable that is used to calculate the next offset within the PUF array. It has a value of 1 and is treated as a constant within the inserted code. Finally, a global lookup table is created where function addresses, based on the PUF responses in the PUF array, are written to and read from.

Access to these variables is not synchronized across threads. The spawned PUF response collecting thread, inserted by the LLVM pass, is the only writer to the global PUF array and a reader of the global variable for index computation. All other threads are readers only of the PUF response array and writers to the index computation global variable. Since the PUF array is set to zero until a PUF response is returned, there is no race condition here. After an entry is made to the PUF array, if a reader thread has not had a chance to read it, the next read will catch the change and compute entries into the lookup table, as once a write is made to the PUF array no change is made afterwards to that entry. The readers of these PUF arrays are writes to the global lookup table. They may use different PUF responses to compute the function addresses, but if multiple writers are writing to the same index in the lookup table to fill the function address, they will all compute the same address, and thus this also does not pose a race condition.

Also, the checksum of the code is static and the result is known and does not change at runtime. Therefore, this is also safe with multiple threads. Each thread is a writer to the global variable used to calculate the index in the PUF array. However, all writers execute the expression:

$$puf\_responses\_index = puf\_responses\_index + checksum$$

The checksum will always be zero. So the value of the global variable never changes as all writers write the same value back, and the PUF collecting thread will place the responses in the correct place in the global PUF array. Only if the checksum does not return zero this will result in undefined behavior.

5.2.4 Blocking Code

Blocking code is inserted for all possible entry points into the LLVM-IR module that waits for the correct PUF response to be collected so that the lookup table can be partially rebuilt at each PUF response. This implicitly causes any program patched with this scheme to run at least as long as the longest enrolled decay time.

5.3 Possible Attacks

Without access to an authorized device on which the PUF is deployed and for which the program was patched the only possible way to reverse engineer the binary is to brute force the PUF responses to reconstruct the call graph. The time needed to spend for resolving the PUF responses depend on the error correction used and the number of decay times used and the size of the PUF responses.

For example, if we consider decay times T = {20s, 30s, 40s, 50s} and at each timeout a 32-bit PUF response is required to decode part of the lookup table and an error correction of 5% or 2 bits is used, the adversary would need to brute force 2^{30} possible values on each response. Since on average an answer can be brute forced halfway through, consider that the attack would need to brute-force 2^{29} possible values for each decay time, which is $4 * 2^{29}$. This is achievable on modern computers. If larger PUF responses are to be used, longer decay times should be used, since more bits will be decayed at a later time than at an earlier time. A 128-bit response could be obtained after $\approx$ 20s of decay using a 4MB PUF size, as shown in the table 5.2, and with an error correction of 25%, an adversary would have to brute force $4 * 2^{95}$ values to rebuild the entire table, which may already be enough security in terms of how much computing power the adversary would need.

5.3.1 With Access to PUF

The adversary would need access to the correct PUF instance for which the binary was enrolled. If the adversary was able to gain access to authorized devices, but the binary was not enrolled on those devices, the adversary would not be able to recover the PUF responses. With access to an authorized device, and the PUF interface on which the binary was enrolled, reverse engineering the binary is easier. The only defence in such a scenario is the timing of the PUF queries, as a PUF is easily broken by a replay attack.

If the adversary has root access on the device its possible to replace the PUF kernel module with a fake implementation that records the times at which requests are made. These can then subsequently be used to reconstruct the PUF responses and rebuild the lookup table.

If root access is not possible, there are still ways to rebuild the lookup table. Access to a debugger on an authorized device is sufficient to obtain the PUF responses and reconstruct the lookup table. The patched program only protects against a few dynamic attacks. Inserted breakpoints are caught by the checksum code and result in undefined program behavior. By default, when a breakpoint hits, it freezes all threads, and if enough time is spent examining the state of the program's registers and memory at that breakpoint, the PUF-collecting thread will read the response at a later time when the attacker steps through the instructions again. However, this can be worked around; if the PUF thread is identified, it can simply be made not to freeze like the other threads. By statically analyzing the binary, it is possible to find the location of the lookup table by following the indirect function calls. This can be used to set a watchpoint at runtime to the given address, and then examine the changes to recover the PUF responses. Anti-debugging techniques would have to be used to prevent debugging on an authorized device, where an attacker would first have to statically analyze the binary and remove the anti-debugging techniques to recover the PUF responses. In general, you want to make the dynamic analysis experience as irritating as possible.

In chapter 3 it was mentioned that a decay-based DRAM PUF is a weak PUF because the number of challenge-response pairs is small. If a single device uses the same DRAM region as a PUF, many patched programs will authenticate against it. It would be possible to collect enrollments by directly reading the patched binaries, and based on the just described debugger attack, it would be possible to create mapping tables of which cells have which values at the desired decay times for the given challenge. With enough collected challenge-response pairs, the PUF responses could be brute-forced using only these mapping tables, without needing access to the PUF. However, this can be circumvented by using different DRAM regions and thus different PUFs for each binary on a single device. This complicates the construction of those the mapping tables by needing to collect vast amounts of binaries from which enrollments would be collected for different PUFs running on the same device. These mapping tables would only be valid for a single device, and thus this attack would need to be replicated on each authorized device for all deployed binaries and can quickly become impractical.

5.4 Comparison with Previous Work

This work is based on the work of Xiong et. al. [41], in which the dynamic decay-based DRAM PUF was used to patch programs that should be executed at specific time intervals and without any variations, as this would result in deviations in the decay of the DRAM cells and could cause the program to fail to authenticate even on an authorized device. In the paper, the authors described how different control flow structures could be patched using their HESP scheme, which they evaluated on a simple program using AES encryption decryption using 32-bit PUF responses at decay times T = { 20, 30, 40, 50, 60, 70, 80, 90, 100}. The sample program had sleep instructions inserted into the control flow to simulate collecting the PUF responses at the requested decay times.

In the paper, the authors do not consider or even mention patching programs that may

use blocking function calls in their control flow, such as system calls or input/output calls over a network, operations that may be common for an IoT device connected to other parts of a larger system. These blocking calls could block for a few seconds, and it could happen that the PUF response is read at the wrong time, triggering the built-in tamper response mechanisms. By considering only programs that are intended to run within certain time intervals, the set of programs that could end up using this HESP scheme narrows down, since any program that communicates with other parts of the system is at risk of potential failure. Further, the authors described that the timings of the PUF queries must be recorded using a kernel module, or they must be known in advance. The larger the program to be patched, the more time consuming the recording of the PUF queries may become, as one would most likely need to cover 100% of all possible paths in the source code with a PUF query, in which case selecting specific decay times may be more desirable to unlock specific control flow paths within the program. Furthermore, what if more than the main thread is executing code, and each thread has different timings at which they query the PUF response? A function could be patched to query the PUF at 20s from a particular path, but another thread could call the same function earlier and query the PUF at 10s. This would require some kind of PUF synchronization to be implemented. Consider the example program in Listing 5.2

```
fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080")?;
    for stream in listener.incoming() {
        match stream {
          Ok(stream) => { std::thread::spawn(|| handle_client(stream)); }
          Err(e) => { eprintln!("Error accepting client: {}", e); }
        }
    }
    Ok(())
}
```

Listing 5.2: TCP server

A simple TCP server, where the times at which the code is executed are not known, because each incoming connection can execute a different path where each instruction can be executed at different timings that depend on when the given incoming connection is handled by the TCP server. The described HESP scheme from Xiong et al. would fail in this simple scenario. One could enroll the DRAM cells for the decay times T = { 20, 30, 40, 50, 60, 70, 80, 90, 100} and patch this program to query the PUF at the specified decay times, and yet each of them would fail because the timings of the incoming connection are not known, unless each of the incoming connections were executed exactly at the specified timings, which implicitly means that the program would be part of a system that is sensitive to timing variations. Even if a PUF reset was used at the start of the for loop on each incoming connection, a non-deterministic blocking function call in the control flow could cause the PUF to be read at a later time than intended. Furthermore, using a PUF reset with more than one execution thread would result in inter-thread interference. Therefore, a different approach was taken in this work compared to the work of Xiong et al.

Creating a general purpose framework that would patch all sorts of different control flow structures is a hard problem that is not entirely solved by implementing an LLVM pass that would patch the LLVM IR code to use the PUF. At the LLVM-IR level, when a function is called we do not know for how long or exactly at what time this function will be executed, because that context is missing at LLVM-IR level. We can only make certain assumptions that allow us to patch the LLVM IR to preserve the original control flow. In order to preserve the original control flow, as it may contain blocking calls, the choice was made to use a separate thread that would query the PUF responses at requested decay times, and where each PUF response would be used by other executing threads to gradually reconstruct parts of a lookup table of function addresses, eventually reconstructing the original call graph after collecting all PUF responses. This choice allows a wider range of programs, including programs that have more than one thread of execution, to use the PUF that are not limited to simple control flow primitives such as conditionals, loops, etc. This is a completely different approach from the work of Xiong et al., where they use the PUF queries directly in the original control flow of the program.

This choice also has its drawbacks. Compared to the implementation of Xiong et al., where only a single PUF response is held in run-time memory at a time, in this work all PUF responses are held in memory, which makes the attack using a debugger more straightforward compared to their solution, although both versions are susceptible to this kind of attack. In addition, the original control flow of a program is adjusted to have blocking code that waits for a given PUF response that computes the addresses of functions before executing them. This is done for all paths from all possible entry points into the LLVM-IR module, so that given any path into the LLVM-IR, the function addresses are computed after receiving the correct PUF response. In the Autopatcher implementation from Xiong et al., the PUF queries are part of the original control flow that is supposed to query the PUF at the right time. Thus, in cases such as when a pointer to a function containing the PUF query is passed to an external library or within the same module, it may be executed at the wrong time and be at risk of failure.

Finally, Xiong et al. [41] used error correction in the form of repetition codes, where each bit of a PUF response was encoded using 3 bits from 3 different DRAM cells, thus requiring 96 bit PUF responses instead of 32 bits. This setup has been shown to work under the condition where the temperature was stabilized at 40°C, but with varying temperatures the chance of failure is increased, so in this work Reed-Solomon codes were used where the redundancy bytes for error correction are part of the enrollments that are inserted directly into the binary and then used by the kernel module for error correction.

5.5 Future Work

In this work, the base of a patching scheme using a dynamic decay-based DRAM PUF has been implemented on a new device. There is still work that could be done to improve this implementation. Namely temperature effects, longevity of components, anti-debugging, or the possibility to implement a latency based PUF to be used instead of a decay-based.

5.5.1 Temperature Effects

All of the evaluations in this work have been done at room temperatures where the temperature of the components on the device fluctuated only based on the current workload executed on the device. As has been discussed, temperature has a direct effect on the stability of PUF responses, thus further evaluating PUF responses under different temperatures to examine the effect on the decaying of the cells would need to be done to achieve more stable responses.

5.5.2 Longevity

All evaluations for this implementation were performed over a period of ≈ 1 month, with the device under minimal load with only the PUF evaluations running. Some of the related work mentioned in the chapter 2 has evaluated the stability of the responses of random access memory PUFs over longer periods of time. Thus, the effect of DRAM aging on the stability of the responses would need to be documented.

5.5.3 Anti-Debugging

As mentioned earlier, if an attacker has access to an authorized device and a binary that uses enrollments for the PUF running on that device, it is possible to recover the PUF responses using a debugger. Therefore, the LLVM out-of-tree pass can be further enhanced to include anti-debugging techniques to make reverse engineering more time-consuming, where the attacker would first have to identify the anti-debugging code and remove it using static analysis before attaching the program to a debugger.

5.5.4 Latency-based PUF

Xiong et al. [41] described that latency-based DRAM PUFs could not be implemented on commodity off-the-shelf devices because it would require access to modifying the timing latencies of the DRAMs, which is not common on these devices. The BeagleBone black documentation actually states that modifying DRAM timing latencies is possible and is documented on the CPU's official website [17]. The $SDRAM\_TIM\_1$ register is writable according to the documentation, so the DRAM timings should be modifiable. In this work we did not explore this option, so in future work it may be possible to replace the decay-based DRAM PUF and its long response evaluation times with latency-based PUFs that can be evaluated much faster.

6 Conclusion

In this thesis, a new hardware-software binding scheme using LLVM and Linux kernel modules was implemented and described using dynamic decay-based DRAM PUFs on a commodity off-the-shelf device, namely the BeagleBone Black. The stability of the PUF responses was evaluated with experiments conducted on the device over a period of $\approx$ 1 month. The evaluation showed promising results where the DRAM on the device was shown to have PUF characteristics. It was shown that stable PUF responses can be constructed to be used by a binary to authenticate against an authorized device at runtime during normal operation of the device.

The implemented scheme provides the basis for demonstrating that a DRAM PUF can be used on the selected device, since the device used in the previous work was discontinued and thus could not be built upon. Further work would need to be done to evaluate the stability of the PUF responses under different conditions, namely varying temperatures and component aging.

Decay-based DRAM PUFs have long evaluation times, requiring at least $\approx$ 10-15 seconds to construct a reasonably stable response. Related work is already addressing this problem by using latency-based DRAM PUFs, which evaluate much faster, in a matter of milliseconds, but require software access to modify the DRAM latencies. The documentation of the selected off-the-shelf device used in this thesis describes that the DRAM timing parameters can be modified at runtime by writing to certain registers, it may be possible to explore the possibility of implementing a latency-based PUF on the selected device.

In effect, a PUF is just another tool that extends the time it takes for an attacker to make changes to the distributed binary. Surreptitious software employs a layer of defense at the software level, whereas PUFs are a tamper detection mechanism at the hardware level, which is stronger. The purpose of using a PUF is for the software to authenticate itself that it is running on verified hardware. By using a PUF, the attacker needs not only the distributed binary, but also the hardware for which it was developed, unless he/she plans to spend more time reverse-engineering the software without the hardware.

Bibliography

[1] Duty cycle. Accessed on 2-4-2024, https://en.wikipedia.org/wiki/Duty_cycle.

[2] Reed-solomon codes. Accessed on 5-4-2024, https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html#:~:text=Reed%20Solomon%20codes%20are%20a,symbols%20of%20s%20bits%20each.

[3] Row hammer. Accessed on 3-4-2024, https://en.wikipedia.org/wiki/Row_hammer.

[4] Static single-assignment form. Accessed on 6-4-2024, https://en.wikipedia.org/wiki/Static_single-assignment_form.

[5] Nikolaos Athanasios Anagnostopoulos, André Schaller, Yufan Fan, Wenjie Xiong, Fatemeh Tehranipoor, Tolga Arul, Sebastian Gabmeyer, Jakub Szefer, John Chandy, and Stefan Katzenbeisser. Insights into the potential usage of the initial values of dram arrays of commercial off-the-shelf devices for security applications, 04 2017.

[6] B. M. S. Bahar Talukder, Biswajit Ray, Domenic Forte, and Md Tauhidur Rahman. Prelatpuf: Exploiting dram latency variations for generating robust device signatures. *IEEE Access*, 7:81106–81120, 2019.

[7] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype, 2006. Accessed on 31-3-2024, https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf.

[8] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2009. Chapter 4, pages 297-428.

[9] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2009. Chapter 7, pages 564-650.

[10] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley, 2009. Chapter 8-9, pages 651-840.

[11] eShard. Obfuscator-llvm. Accessed on 20-4-2024, `https://github.com/eshard/obfuscator-llvm`.

[12] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, page 27–38, New York, NY, USA, 2016. Association for Computing Machinery.

[13] BeagleBoard.org Foundation. Beaglebone black. Accessed on 6-4-2024, `https://git.beagleboard.org/beagleboard/beaglebone-black`.

[14] Florian Gondesen, Sananda Mitra, and Kwok-Yan Lam. Feasibility of puf-based authentication on attiny devices with off-the-shelf sram. In *Proceedings of the 6th ACM on Cyber-Physical System Security Workshop*, CPSS '20, page 2–10, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2007.

[16] Maryam S. Hashemian, Bhanu Singh, Francis Wolff, Daniel Weyer, Steve Clay, and Christos Papachristou. A robust authentication methodology using physically unclonable functions in dram arrays. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, page 647–652, San Jose, CA, USA, 2015. EDA Consortium.

[17] Texas Instruments. Am335x and amic110 sitara processors technical reference manual. Accessed on 6-4-2024, `https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf?ts=1712392265607`.

[18] Texas Instruments. Arm cortex-a8 (am3358). Accessed on 6-4-2024, `https://www.ti.com/product/AM3358`.

[19] Intel. Intel galileo gen 2 board. Accessed on 2-4-2024, `https://www.intel.com/content/www/us/en/products/sku/83137/intel-galileo-gen-2-board/specifications.html`.

[20] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm – software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9, 2015.

[21] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The dram latency puf: Quickly evaluating physical unclonable functions by exploiting the latency-reliability tradeoff in modern commodity dram devices. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 194–207, 2018.

[22] Indra Kumari, Mi-Kyung Oh, Yousung Kang, and Dooho Choi. Rapid run-time dram puf based on bit-flip position for secure iot devices. In *2018 IEEE SENSORS*, pages 1–4, 2018.

[23] Chris Lattner. The architecture of open source applications (volume 1) llvm. Accessed on 5-4-2024, `https://aosabook.org/en/v1/llvm.html`.

[24] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern dram devices: implications for retention time profiling mechanisms. *SIGARCH Comput. Archit. News*, 41(3):60–71, jun 2013.

[25] Shayesteh Masoumian, Georgios Selimis, Rui Wang, Geert-Jan Schrijen, Said Hamdioui, and Mottaqiallah Taouil. Reliability analysis of finfet-based sram pufs for 16nm, 14nm, and 7nm technology nodes. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1189–1192, 2022.

[26] Charis Mesaritakis, Marialena Akriotou, Alexandros Kapsalis, Evangelos Grivas, Charidimos Chaintoutis, Thomas Nikas, and D. Syvridis. Physical unclonable function based on a multi-mode optical waveguide. *Scientific Reports*, 8, 06 2018.

[27] Jack Miskelly and Máire O'Neill. Fast dram pufs on commodity devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3566–3576, 2020.

[28] Fatemeh Najafi, Masoud Kaveh, Mohammad Reza Mosavi, Alessandro Brighente, and Mauro Conti. Epuf: A novel scheme based on entropy features of latency-based dram pufs providing lightweight authentication in iot networks, 2023.

[29] Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. Identifying and understanding self-checksumming defenses in software. *CODASPY 2015 - Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218, 03 2015.

[30] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[31] Andre Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Intrinsic rowhammer pufs: Leveraging the rowhammer effect for improved security. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, May 2017.

[32] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. Decay-based dram pufs in commodity devices. *IEEE Transactions on Dependable and Secure Computing*, 16(3):462–475, 2019.

[33] Geert-Jan Schrijen and Vincent van der Leest. Comparative analysis of sram memories used as puf primitives. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, page 1319–1324, San Jose, CA, USA, 2012. EDA Consortium.

[34] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14, 2007.

[35] Soubhagya Sutar, Arnab Raha, and Vijay Raghunathan. D-puf: An intrinsically reconfigurable dram puf for device authentication in embedded systems. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, 2016.

[36] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John Chandy. Investigation of dram pufs reliability under device accelerated aging effects. pages 1–4, 05 2017.

[37] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John A. Chandy. Dram-based intrinsic physically unclonable functions for system-level security and authentication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(3):1085–1097, 2017.

[38] Rui Wang, Georgios Selimis, Roel Maes, and Sven Goossens. Long-term continuous assessment of sram puf and source of random numbers. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 7–12, 2020.

[39] Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. Dynamic physically unclonable functions. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 311–314, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Wenjie Xiong, André Schaller, Nikolaos A. Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Run-time accessible dram pufs in commodity devices. Cryptology ePrint Archive, Paper 2016/253, 2016. `https://eprint.iacr.org/2016/253`.

[41] Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. Software protection using dynamic pufs. *IEEE Transactions on Information Forensics and Security*, 15:2053–2068, 2020.

[42] Wei Yan, Fatemeh Tehranipoor, and John A. Chandy. Puf-based fuzzy authentication without error correcting codes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1445–1457, 2017.