Dynamic Physically Unclonable Functions

Wenjie Xiong†, André Schaller‡, Stefan Katzenbeisser‡, and Jakub Szefer†
†Yale University, New Haven, CT, USA ‡Technische Universität Darmstadt, Darmstadt, Germany
wenjie.xiong@yale.edu,{schaller,katzenbeisser}@seceng.informatik.tu-darmstadt.de,jakub.szefer@yale.edu

ABSTRACT

Physical variations in the manufacturing processes of electronic devices have been widely leveraged to design Physically Unclonable Functions (PUFs), which can be used for authentication and key storage. Existing PUFs are static, as their PUF responses remain the same regardless when the PUF is queried. Meanwhile, this paper presents the new concept of *Dynamic PUFs*, where the responses depend not only on the physical properties of the device but also on the timing of the PUF queries. One application of Dynamic PUFs is in dynamic software-hardware binding, where the control flow of the software can be tied to both the timing of the software and the physical properties of the hardware, in order to protect software execution. This paper presents a realization of Dynamic PUFs using DRAM modules. The evaluation is based on the decay-based DRAM PUFs, which can be realized today and were implemented on commodity devices for testing.

ACM Reference Format:
Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. 2019. Dynamic Physically Unclonable Functions. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299874.3318025

1 INTRODUCTION

The number of low-end embedded computing devices is continuously growing, and this growth is expected to accelerate further due to the interest in the Internet-of-Things (IoT). These low-end devices are being deployed in a variety of settings from healthcare to industrial environments, where the integrity and tamper resistance of the software is critical. Meanwhile, hardware support in the form of Physically Unclonable Functions (PUFs) can readily be used in low-end devices [1] and can be used to protect software from modification or from running on unauthorized devices [4]. PUFs extract unique and stable physical features emerging from fabrication variations of the underlying hardware modules. All existing PUFs are *static*, meaning a PUF response for a given challenge is stable and independent of the timing of the query.

This work proposes the new concept of *Dynamic PUF*s. A Dynamic PUF generates responses at device runtime, and the PUF responses depend on both the challenges and the timing of the PUF queries. Compared to the existing PUFs that give responses independently of the query times, Dynamic PUFs will give different responses even if the same PUF is queried with the same challenge, but at different times relative the most recent PUF reset. Dynamic PUFs can be realized with PUFs in DRAM, which can be accessed at system runtime and which have time-dependent responses [2, 12].

Dynamic PUFs can interact with software at runtime. With the new Dynamic PUFs, we also propose dynamic software-hardware binding, where the software can be designed to run only on authorized devices and only with correct timing. If the timing is wrong, the software will fail.

The contributions of this work are as follows:

- We introduce the concept of Dynamic PUFs, which have time-dependent responses.
- We show an application of Dynamic PUFs: the dynamic software-hardware binding.
- We develop a practical instance of a Dynamic PUF using a decay-based Dynamic DRAM PUF on Intel Galileo Gen 2 platform. We also evaluate the characteristics of the decay-based DRAM PUFs for their suitability as a Dynamic PUF.

2 DYNAMIC PUFS

PUFs extract unique and stable physical features from physical objects [1]. Given a challenge (also called a PUF query), a PUF can generate a response, which is a function of both the challenge and the physical features of the PUF. This is known as a challenge-response pair (CRP). Ideally, each physical object used as a PUF will generate a unique PUF response for each challenge (i.e., the uniqueness property of PUFs). Repeated queries with the same challenge on the same PUF should give the same response (i.e., the robustness property of PUFs), but in practice, they may vary a little due to noise. All of the existing PUF designs do not consider the timing of the PUF query in generating the response, and we call these Static PUFs.

Meanwhile, this work presents the notion of a *Dynamic PUF*. Different from the usual Static PUFs, Dynamic PUFs provide time-dependent PUF responses, i.e., the responses depend not only on the challenges but also on the timing of the PUF queries, relative to the most recent PUF reset.

2.1 Operation of Dynamic PUFs

Besides the usual PUF query operation, a Dynamic PUF has an extra operation called *Dynamic PUF Reset*, which resets the state of the PUF. A *Dynamic PUF Query* is akin to the PUF query for Static PUFs: given a PUF challenge, a PUF response will be returned. However, a Dynamic PUF will give different responses even if queried with the same challenge, but at a different time since the most recent *Dynamic PUF Reset*.

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3318025

2.2 Metrics for Dynamic PUFs

Ideal Dynamic PUF responses should have two features. First, responses at different times should be independent of each other, i.e., the response obtained at time T_x is independent of the response at time T_y when $|T_y - T_x| > \delta_{t0}$, where δ_{t0} is the time resolution of the Dynamic PUF. Second, the responses should be stable. The following metrics can be used to evaluate a Dynamic PUF.

Time Resolution: Denoted by δ_{t0}, time resolution indicates how sensitive the PUF responses are to the query time (since the last reset of the PUF). Each realization of a Dynamic PUF has a physical limit of the time resolution.

Robustness and Uniquess: The *intra distance* is the distance between PUF responses of the same challenge and query time on the same PUF device. Here, distance is a measure of how different two PUF responses are, and it can be measured typically by the Hamming Distance or the Jaccard Index. The intra distance evaluates the robustness of the PUF. Ideally, the intra distance should be small. The *inter-challenge distance* is the distance between PUF responses from different PUF devices or the same PUF device but different challenges. It evaluates the uniqueness of PUF responses across different challenges and different PUF devices. Ideally, the inter-challenge distance should be large. The *inter-time distance* is the distance between PUF responses with the same challenge but with the different query times on the same PUF device, where the time difference between query times is greater than δ_{t0}. It evaluates the uniqueness of the PUF responses across different query times.

2.3 Helper Data System for Dynamic PUFs

Usually, the inter-time distance is not ideal, and the PUF responses at different times are not independent. Thus, a Helper Data System (HDS) is needed to amplify the entropy from the raw PUF response and to extract the final PUF response corresponding to each query time. This requires a different HDS entry for each query time T_x. Also, noise exists in raw Dynamic PUF responses (the intra distance is not ideal) and error correction needs to be performed as well.

To retrieve the PUF response at a specific query time, the *Dynamic PUF Query* should indicate which helper data to use, by specifying an index to the entry in the HDS. To generate the HDS, a trusted party needs to measure the desired PUF CRPs with different query times in an *enrollment* phase.

3 DRAM PUFS AS DYNAMIC PUFS

DRAM, which is found in many commodity IoT and embedded platforms, has been shown to exhibit PUF behavior. There are three types of DRAM PUFs: decay-based DRAM PUFs [5–8, 10, 12], latency-based DRAM PUFs [2], and startup-based DRAM PUFs [11]. Here, we show that decay-based DRAM PUFs can be used as Dynamic PUFs.

In Dynamic DRAM PUFs, the PUF challenge is the address range of a *DRAM PUF region* from which the PUF will be extracted, and an *initial value* written to the cells at Dynamic PUF reset.

3.1 Decay-based DRAM PUFs

The decay-based DRAM PUF leverages the fact that DRAM cells lose data over time, which is also known as *DRAM decay*. Without refresh, each DRAM cell can only retain the data for a certain time,

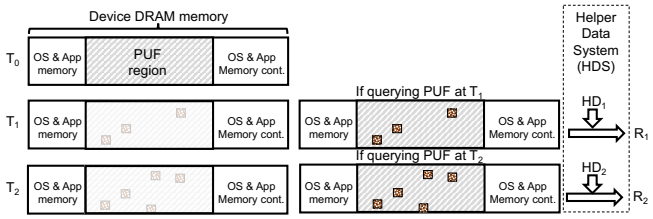

Figure 1: The responses of a decay-based DRAM PUF depend on the query time since the PUF reset at T_0.

called *retention time*. Decay-based DRAM PUF responses are based on the variation of the retention times of the DRAM cells. A decay-based DRAM PUF is composed of a set of cells within a DRAM region. The retention time of each DRAM cell is believed to be random and related to fabrication variations [10, 12]. As illustrated in Figure 1, querying the decay-based DRAM at different times (e.g., at T_1 and T_2) will result in different bit flips in the DRAM region, which is the PUF response. We use the *decay time* to denote the time between the *Dynamic PUF Reset* and the *Dynamic PUF Query*. The time resolution δ_{t0} of the dynamic decay-based DRAM PUF is on the order of seconds. Latency-based DRAM PUF [2] can be an alternative implementation of Dynamic PUF, which has time resolution δ_{t0} on the order of nanoseconds, but cannot be easily realized in commodity devices today.

DRAM has the additional property that each access to a DRAM cell (e.g., on a PUF query) implicitly refreshes all the DRAM cells in that row, which resets the Dynamic PUF. When one wants to query the PUF region several times after each reset, the PUF region should be subdivided into sub-regions. Each sub-region consists of a set of DRAM rows, and these DRAM rows do not have to be adjacent to each other. Upon a *Dynamic PUF Reset* request, all DRAM PUF sub-regions have their initial values set and refresh is disabled. To query the Dynamic PUF, an index to the HDS entry is provided as part of the *Dynamic PUF Query*. The index will also determine which sub-regions should be used for that query. On a query, only that sub-region will be accessed (and implicitly refreshed) while other sub-regions continue to decay.

3.2 Accessing the Decay-based Dynamic DRAM PUF at System Runtime

A decay-based Dynamic DRAM PUF can be implemented on a commercial, off-the-shelf device by selectively refreshing DRAM as shown in Figure 1, following [12]. First, on a Dynamic PUF Reset (T_0), the DRAM PUF region is reserved, so that neither OS nor applications use it, its cells are initialized with the initial value (e.g., all zeros), and the refresh of the DRAM module is disabled. To allow the OS and other applications use DRAM without decay, a customized kernel module can selectively access and thus refresh the DRAM rows [1], as in [12]. Upon a PUF query, the content of the DRAM PUF region is read as the raw PUF response.

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

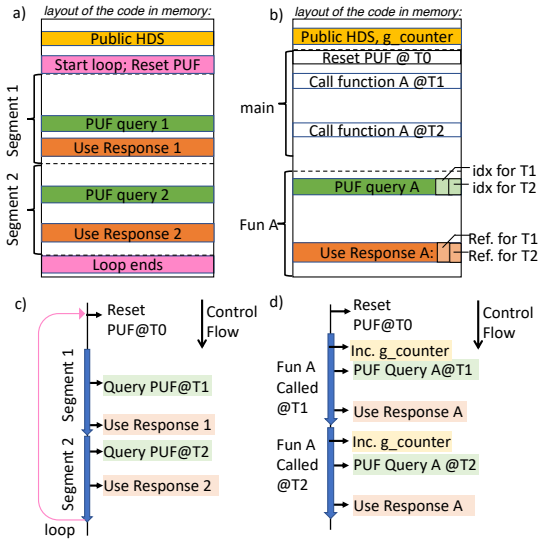

Figure 2: Example code layout of software-hardware binding for a) straight-line code in a loop and b) repeated function calls. Parts c-d) show the control flow of each protection example of a-b).

4 APPLICATIONS OF DYNAMIC PUFS

Software-hardware binding ties the software execution to a hardware token [4]. With software-hardware binding, the software can only be executed on authorized devices, and thus, it prevents illegal software distribution. In the context of software obfuscation, software-hardware binding has been used to strengthen the obfuscation [9]. When software-hardware binding is used, analyzing the binary will not reveal all the information of the software (e.g., targets of indirect jumps), and the attacker will not be able to reverse engineer the software completely. With a Dynamic PUF, this obfuscation can be even stronger, because an attacker needs to obtain all Dynamic PUF responses at different query times in order to reverse engineer the software.

One of the software-hardware binding approaches is to change all function calls to indirect jumps. Now, we show how to bind all such jumps to the Dynamic PUF responses. This can be achieved by inserting code to make a PUF query and replacing the indirect jump destination address with an exclusive-OR result of the Dynamic PUF response and a reference value. Below, we give examples of software-hardware binding with different software structures.

Straight-Line Code: As shown in Figure 2 (a,c), at the beginning of the program, the Dynamic PUF will be reset. The Dynamic PUF is queried in each segment. The PUF response then depends on the execution time elapsed between the last PUF reset (at the beginning of the program) and the PUF query time. The destination address of the function call at *Use Response* 1 depends on the PUF response, and the program will continue the execute to segment 2.

Programs with Repeated Function Calls: In this scenario, some code will be executed more than once, but in a deterministic way, as in Figure 2 (b, d). This could happen when a function is called several times repeatedly, e.g., *func A* is called twice from the main function. However, since a Dynamic PUF is used, the PUF will return different responses each time the PUF query code in the function is executed (assuming the time between the two PUF queries are larger than δ_{t0}), and the PUF queries need to use different *idx* to the HDS entries at different times.

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

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

5 EVALUATION

5.1 Dynamic PUF Characteristics

We evaluate the decay-based Dynamic DRAM PUF that can be deployed on commodity hardware today, in particular on Intel Galileo Gen 2 platforms with two 128 MiB DDR3. A heater and thermocouple are used with a control system to stabilize DRAM temperature at 40°C. We evaluated 8 MiB DRAM regions for PUF with decay times $\mathcal{T}_x$ ranging from 15s to 40s. To evaluate the time resolution δ_{t0}, we denote δ_t as the smallest time difference between different query times, and evaluated different δ_t.

Figure 3 shows the distribution of the intra, inter-challenge, and inter-time distance as discussed in Section 2.2. Here, we use the Jaccard index to measure the distance between different responses. The bit flips in each response are treated as a set of indices. The Jaccard index of the two sets (A and B) from two responses are calculated by $J(A,B) \stackrel{\text{def}}{=} \frac{|A \cap B|}{|A \cup B|}$. When the two responses are identical, $J = 1$. When the two responses do not share any bit flips, $J = 0$. As shown in Figure 3, the intra Jaccard index is very close to 1, indicating that the PUF response is robust. The inter-challenge Jaccard index from two devices is very close to zero, meaning the PUF responses from different DRAM regions are very different. The inter-time Jaccard indices for time differences $\delta_t = \{5s, 2s, 1s\}$ are also shown in Figure 3. For all the three time differences, the inter-time Jaccard indices are smaller than the intra Jaccard index, indicating the responses from different decay times are different enough to be distinguished. Thus, the time resolution is better than 1s. And a bigger δ_t makes the responses between different decay times more distinguishable.

We use the HDS proposed in [8]. For each T_x, one enrollment measurement was taken at $T_x - 2\delta_t$, $T_x - \delta_t$, $T_x + \delta_t$ and $T_x + 2\delta_t$ separately, and cells that flip in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ and $[T_x + \delta_t, T_x + 2\delta_t]$ are enrolled in the HDS for T_x. The

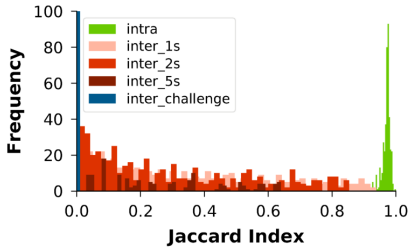

Figure 3: Distribution of inter and intra Jaccard indices.

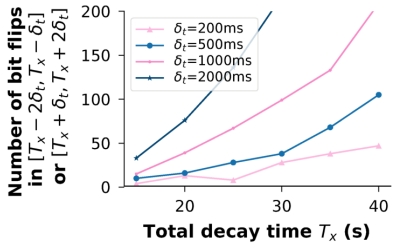

Figure 4: Available bit flips in decay-based Dynamic DRAM PUFs.

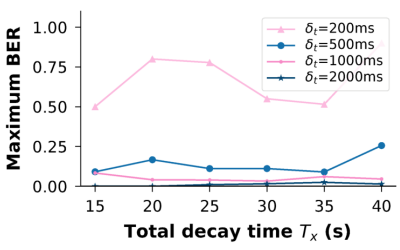

Figure 5: Bit Error Rate of decay-based Dynamic DRAM PUFs.

cells that flip in $[T_x - 2\delta_t, T_x - \delta_t]$ during enrollment should flip in response at T_x and become logical 1, and the cells that flip in $[T_x + \delta_t, T_x + 2\delta_t]$ during enrollment should not yet flip in the response at T_x and become logical 0. Then, ten measurements are taken at T_x to evaluate the bit error rate (BER) of the enrolled cells. We evaluated total decay times $\mathcal{T}_x$={15s, 20s, 25s, 30s, 35s, 40s} and time differences δ_t={2s, 1s, 500ms, 200ms}.

Figure 4 shows the average number of bit flips in the time interval $[T_x - 2\delta_t, T_x - \delta_t]$ or $[T_x + \delta_t, T_x + 2\delta_t]$ in the two boards measured. Figure 4 shows that longer T_x or larger δ_t give more bits that can be used. To generate the HDS for time T_x, $32 \times 3 = 96$ bit flips are needed in time range $[T_x - 2\delta_t, T_x - \delta_t]$ and $[T_x + \delta_t, T_x + 2\delta_t]$, considering the case 32 bits are needed for each response and 3-repetition code are used for error correction code (ECC). With 8 MiB DRAM PUF size, when the total decay time is larger than 25s for $\delta_t = 2$s, it can provide more than 96 bits. To achieve a smaller δ_t, a larger DRAM size should be used. To compute the average BER of the PUF reconstructions, with the generated HDS from the enrollment, we divide the number of cells that do not flip to their desired logical value by the total number of cells enrolled for each T_x. Figure 5 shows the maximum BER in all reconstructions for both evaluated devices. When the time difference δ_t is larger than 500 ms, the BER is smaller than 25%, which can be corrected by 3-repetition code. Figure 5 also shows that the minimum time resolution of the dynamic decay-based DRAM PUF is about 500 ms.

5.2 Applications and Dynamic PUF Overhead

To test dynamic software-hardware binding, we bind the Dynamic PUF response to a program which contains AES-128 encryption and decryption, as well as a program which contains SHA-512. Both programs contain repeated function calls and loops to show that the scheme can support programs with non-trivial structures. HDS is inserted into the program binary. For each PUF query, the HDS requires 384 Bytes (4 Bytes for each pointer, and with 3-repetition code, 3×32 cells are needed). The AES-128 application, for example, contains 9 PUF queries, the HDS takes up 3.4 KiB, and the total binary size is increased by 32.5%. The implementation is in software and no hardware overhead is required.

6 CONCLUSION

This paper presented the new concept of *Dynamic PUFs*, as well as their possible implementations and applications. Different from the usual static PUFs, Dynamic PUFs have time-dependent responses. With Dynamic PUFs, the software execution (timing) can affect the PUF response dynamically, making dynamic software-hardware binding possible. Furthermore, the characteristics of decay-based Dynamic DRAM PUF were evaluated on Intel Galileo Gen 2 boards.

ACKNOWLEDGMENTS

This work has been supported by NSF through grant #1651945.

REFERENCES

[1] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. 2012. PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 283–301.

[2] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. 2018. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 194–207.

[3] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE, 361–372.

[4] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. 2015. PUF-based software protection for low-end embedded devices. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer, 3–21.

[5] Amir Rahmati, Matthew Hicks, Daniel E. Holcomb, and Kevin Fu. 2015. Probable Cause: The Deanonymizing Effects of Approximate DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, 604–615.

[6] Sami Rosenblatt, Srivatsan Chellappa, Albert Cestero, Norman Robson, Toshiaki Kirihata, and Srikanth S Iyer. 2013. A Self-Authenticating Chip Architecture Using an Intrinsic Fingerprint of Embedded DRAM. *IEEE Journal of Solid-State Circuits* (2013), 2934–2943.

[7] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. 2017. Intrinsic rowhammer PUFs: Leveraging the rowhammer effect for improved security. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 1–7.

[8] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Skoric, Stefan Katzenbeisser, and Jakub Szefer. 2018. Decay-Based DRAM PUFs in Commodity Devices. *IEEE Transactions on Dependable and Secure Computing* (2018).

[9] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 4.

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

[11] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. 2015. DRAM based intrinsic physical unclonable functions for system level security. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 15–20.

[12] Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. 2016. Runtime accessible DRAM PUFs in commodity devices. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 432–453.