UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

BRUNO ENDRES FORLIN

G-PUF: A software-only PUF for GPUs

Porto Alegre

2019

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

BRUNO ENDRES FORLIN

G-PUF: A software-only PUF for GPUs

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

UFRGS

Supervisor: Prof. Dr. Ronaldo Husemann
Co-supervisor: Prof. Dr. Luigi Carro

Porto Alegre
2019

CIP - Catalogação na Publicação

Forlin, Bruno
G-PUF: A Software-only PUF for GPUs / Bruno Forlin. -- 2019.
47 f.
Orientador: Ronaldo Husemann.

Coorientador: Luigi Carro.

Trabalho de conclusão de curso (Graduação) -- Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Curso de Engenharia Elétrica, Porto Alegre, BR-RS, 2019.

1. Segurança de Hardware. 2. GPU. 3. PUF. I. Husemann, Ronaldo, orient. II. Carro, Luigi, coorient. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da UFRGS com os dados fornecidos pelo(a) autor(a).

BRUNO ENDRES FORLIN

G-PUF: A software-only PUF for GPUs

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Escola de Engenharia da Universidade Federal do Rio Grande do Sul, como requisito parcial para Graduação em Engenharia Elétrica

Prof. Dr. Ronaldo Husemann
Supervisor - UFRGS

Prof. Dr. Luigi Carro
Co-Supervisor - UFRGS

EXAMINATION BOARD

Prof. Dr. Altamiro Amadeu Susin
UFRGS

Prof. Dr. Tiago Roberto Balen
UFRGS

Acknowledgements

Agradecer a todos os envolvidos na realização deste projeto se prova uma tarefa tão hercúlea quanto escrever o próprio trabalho. Como a última etapa na longa caminhada que foi minha graduação em Engenharia Elétrica, este texto representa todos os encontros e experiências que tive nesses 6 anos e meio de UFRGS. Período que me ajudou a me definir como profissional e como ser humano e a qual sou extremamente grato por ter vivenciado. Eu gostaria de agradecer os meus pais Flávio e Andréia pelo eterno suporte e confiança, mesmo nos momentos mais difíceis. Aos meus orientadores, Ronaldo Husemann e Luigi Carro, pelos conselhos e direcionamentos, sem os quais este projeto nunca teria saído do papel. À Luiza Garaffa, por ajudar a dar significado a essa caminhada e por me acompanhar na busca desse sonho louco. Ao Cézar Reinbrecht, que abriu as portas e me guiou por um caminho novo. E finalmente, a todos os amigos e colegas que fizeram parte desta jornada, chegamos até aqui juntos.

Abstract

Physical Unclonable Functions (PUFs) are security primitives which allow the generation of unique IDs and security keys. Their security stems from the inherent process variations of silicon chips manufacturing, and the minute random effects introduced in integrated circuits. PUFs usually are manufactured speciffically for this purpose, but in the last few years several proposals have developed PUFs from off-the-shelf components. These Intrinsic PUFs avoid modifications in the hardware and explore the low cost of adapting existing technologies. Graphical Processing Units (GPUs) present themselves as promising candidates for an Intrinsic PUF. GPUs are massively multi-processed systems originally built for graphical computing and more recently re-designed for general computing. These devices are distributed across a variety of systems and application environments, from computer vision platforms, to server clusters and home computers. Building PUFs with software-only strategies is a challenging problem, since a PUF must evaluate process variations without rendering system performance, characteristics which are easily done in hardware. In this work we present G-PUF, an intrinsic PUF technology running entirely on CUDA. The proposed solution maps the distribution of soft-errors in matrix multiplications when the GPU is running on adversarial conditions of overclock and undervoltage. The resulting error map will be unique to each GPU, and using a novel Challenge-Response Pair extraction algorithm, G-PUF is able to retrieve secure-keys or an device ID without disclosing information about the PUF randomness. The system was tested in real setups and requires no modifications whatsoever to an already operational GPU. G-PUF was capable of achieving upwards of 94.73% of reliability without any error correction code and can provide up to 2^{53} unique Challenge-Response Pairs.

Keywords: Hardware Security, PUFs, GPUs, Software-only system, Physically Unclonable Functions, Security primitives

Resumo

Physically Unclonable Functions (PUFs) são primitivas de segurança que permitem a criação de identidades únicas e de chaves seguras. Sua segurança deriva das variações de processo intrínsecas à fabricação de chips de silício, e os diminutos efeitos aleatórios introduzidos em circuitos integrados. PUFs normalmente são fabricados especificamente para esse propósito, mas nos últimos anos várias propostas desenvolveram PUFs com componentes comuns. Esses PUFs Intrínsecos evitam modificações de *hardware* e exploram o baixo custo de adaptar tecnologias já existentes. Unidades de Processamento Gráfico (GPUs) se apresentam como candidatos promissores para um PUF Intrínseco. GPUs são sistemas massivamente multi-processados, desenvolvidos originalmente para computação gráfica e mais recentemente reprojetadas para computação genérica. Esses dispositivos estão distribuidos através de uma variedade de sistemas e aplicações, desde plataformas de visão computacional até *clusters* de servidores e computadores pessoais. Construir PUFs com estratégias puramente em *software* é um processo desafiador, já que um PUF deve avaliar variações de processo sem afetar a performance do sistema, características que são mais facilmente alcançáceis em *hardware*. Nesse trabalho, apresentamos o G-PUF, uma tecnologia de PUF Intrínseco rodando puramente em CUDA. A solução proposta mapeia a distribuição de *soft-errors* em multiplicações de matrizes, enquanto a GPU opera em condições adversas como *overclock* e subalimentação. O mapa de erros resultante será único para cada GPU, e utilizando um novo algorítmo para a extração de pares de desafio-resposta, o G-PUF consegue extrair chaves seguras e a identidade do dispositivo sem revelar informações sobre a sua aleatoriedade. O sistema foi testado em condições reais e não requer nenhuma modificação para um sistema de GPU já em operação. G-PUF foi capaz de alcançar uma *reliability* de até 94.73% sem utilizar nenhum código de correção de erros e pode prover até 2^{53} pares de desafio-resposta únicos.

Palavras-Chave: Segurança de *Hardware*, PUFs, GPUs, sistemas de *software*, Funções Físicas Inclonáveis, Primitivas de Segurança

List of Figures

Figure 1 – Pascal Streaming Multiprocessor of the GP100 chip 15
Figure 2 – CUDA Driver model 15
Figure 3 – CUDA Memory Model 16
Figure 4 – Voltage/Frequency curve. 18
Figure 5 – Flattened Voltage/Frequency curve. 28
Figure 6 – Error heat map for the GTX 680 - 1 with a logarithmic color scale. . . 35
Figure 7 – Error heat map for the GTX 680 - 2 with a logarithmic color scale. . . 35
Figure 8 – Number of errors per Threshold percentage in a logarithmic scale - GTX 1060 37
Figure 9 – Number of errors per Threshold percentage in a logarithmic scale - GTX 680 - 1 37
Figure 10 – Number of errors per Threshold percentage in a logarithmic scale - GTX 680 - 2 38
Figure 11 – Plot of Full error map with no threshold and 50% for GTX 1060. White points represent error positions. 39
Figure 12 – Plot of Full error map with no threshold and 15% for GTX 680 - 1. White points represent error positions. 39
Figure 13 – Plot of Full error map with no threshold and 15% for GTX 680 - 2. White points represent error positions. 40
Figure 14 – Hamming distance distribution for PUF responses with 512-bit challenges in different devices. 40
Figure 15 – Hamming distance distribution for PUF responses with 512-bit challenges in the same device. 41

List of Tables

Table 1 – List of features that are present in a 6.0 device, but not in a 3.0 device. 14
Table 2 – Number of errors in the full error map by threshold value for each GPU. 36
Table 3 – Reliability for each device and key size. 41
Table 4 – Uniformity for each device using 100 response samples each. 42

Contents

1 INTRODUCTION **11**

2 BACKGROUND **13**

2.1 Graphical Processing Units **13**

2.1.1 NVIDIA GPU Architecture 13

2.1.2 CUDA Programming Model 14

2.1.3 Differential Voltage/Frequency Scaling 17

3 PHYSICALLY UNCLONABLE FUNCTIONS **19**

3.0.1 Challenge-Response Pairs 19

3.0.2 Point Evaluation 20

3.0.3 Manhattan Distance 20

3.0.4 Cryptographic Primitives 21

3.0.5 PUF Applications 22

3.0.6 Previows works on PUFs 22

3.0.7 PUF Metrics 24

4 METHODOLOGY **26**

4.1 G-PUF **26**

4.1.1 Golden Matrix 26

4.1.2 Thermal Variation 26

4.1.3 Changing Parameters 27

4.1.4 Sample Generation 28

4.1.5 Compare and Store 29

4.1.6 Launch Loop 29

4.2 PUF Protocol **30**

4.2.1 Enrollment 30

4.2.2 Authentication 30

4.2.3 Key Update 31

4.3 CRP Extraction **31**

4.3.1 Circle Area 32

4.4 Validation of Results **33**

5 RESULTS AND DISCUSSIONS **34**

5.1 Experimental Setup **34**

5.2 DVFS Characterization **34**

5.3 Threshold **36**
5.4 PUF Metrics Evaluation **39**
5.4.1 Uniqueness 39
5.4.2 Reliability 41
5.4.3 Uniformity 42
5.5 Discussions **42**

6 CONCLUSION **44**

BIBLIOGRAPHY **45**

1 Introduction

With the proliferation of electronic and autonomous systems, arrives the need for better security mechanisms. This can include: secure protocols for key-exchange, software verification, certificate generation for cloud computing and system-ID. A common element to all of these applications, is the need to generate a unique, unguessable secure key for each device/computation. There are many ways to generate such keys, including Physically Unclonable Functions (PUFs). PUFs are security primitives that can be modeled as one-way functions, they take inputs (challenges) and give an output (response) that can appear random for an eavesdropper (HERDER *et al.*, 2014). This random property derives from the inherent process variations in silicon chip manufacturing and allows for them to be used as fingerprinting devices or as key-generators. Traditionally, PUFs were created with specialized circuitry, specifically manufactured for this purpose. However, in recent years there have been examples of Intrinsic PUFs (LANGE, 2012) (TEHRANIPOOR *et al.*, 2017) (VAN AUBEL; BERNSTEIN; NIEDERHAGEN, 2015) (LI; FU; LUO, 2015a) (KIM *et al.*, 2018) (BACHA; TEODORESCU, 2015), which extract Challenge-Response Pairs (CRPs) from commercial off-the-shelf products. A large portion of the work related to creating Intrinsic PUFs is focused on using memory elements like SRAMs or DRAMs (BACHA; TEODORESCU, 2015) (KIM *et al.*, 2018) (IYENGAR; RAMCLAM; GHOSH, 2014).

From the studied commercial devices, Graphical Processing Units (GPUs) standout as a good opportunity to implement PUFs, specially since GPUs have become common place in cloud computing, HPC (High Performance Computing) and in day-to-day applications due to their great performance to power ratio (KHAIRY; WASSAL; ZAHRAN, 2019). In the past decade, the use of GPUs has gained increased focus as general purpose GPUs (GPGPUs). With the flattening of Moore's law, heterogeneous computing has been chosen as the next way towards increased compute capabilities (SUTTER, 2005). This is pronounced in current fields where GPUs have taken the lead, such as HPC for weather forecast (SCHALKWIJK *et al.*, 2015) (GUO *et al.*, 2016), bioinformatics (NOBILE *et al.*, 2016) (TAYLOR-WEINER *et al.*, 2019), cryptocurrencies mining (IYER; Dipakumar Pawar, 2018), machine learning (CHEN *et al.*, 2015) and specially, Deep Learning algorithms (CUI *et al.*, 2016) (COELHO *et al.*, 2017).

However, existing solutions for GPU PUFs are restricted to older technology (VAN AUBEL; BERNSTEIN; NIEDERHAGEN, 2015), require additional features in the hardware (LI; FU; LUO, 2015b) or present no conclusive results for a GPU PUF (NITHYANAND; SION, 2011) (CHAUVET; MAHÉ, 2013). Also, many solutions as cache based SRAM PUFs (BACHA; TEODORESCU, 2015) might not work properly in GPUs

since ECC is not a standard feature in most GPU models, DRAM based PUFs (KIM *et al.*, 2018) also are not guaranteed to work, since voltage control in the DRAM memory is not facilitated in GPUs.

In this work, we explore the design of Intrinsic PUFs and how this methodology applies to newer technologies. GPUs are explored to bring up a new Intrinsic PUF. We exploit the soft errors that happen when the GPU runs on unstable conditions like undervoltage and overclock using only software. NVIDIA GPUs have been chosen for this work for their wide availability and mostly well documented overclocking utilities. However, the principles behind the PUF can be applied to other GPUs as well. The proposed method can extract several challenge-response pairs by solely running specific software benchmarks, in this case an efficient matrix multiplication. It achieves this by repeating the same computations on different operational parameters and mapping the error positions in a 2D matrix. The errors can be tracked and the frequency on which they appear on each position will vary based on the process variations of the device. This behavior is observed and consequently studied in different GPUs and architectures. We call our PUF technology as G-PUF (GPU + PUF). G-PUF was evaluated in three different GPUs, a NVIDIA GTX 1060 of the Pascal architecture, and two identical GTX 680 from the Kepler architecture.

This text is organized as follows: first a background review will be presented on Chapter 2, containing relevant information about the architecture of NVIDIA GPUs, the CUDA programming model and Differential Voltage/Frequency scalling techinques. Chapter 3 will present an overview of PUFs, their use as criptographic primitives, their applications, previous works and evaluation metrics. Chapter 4 will detail the methodology used to generate a PUF in GPUs, how it could be used in a a secure protocol, how the CRPs can be efficiently and securely extracted, and the tests needed to prove a viable PUF exists. Chapter 5 will present the results of characterization of the GPUs, as well as the collected results of the tests and some discussions about the results. Finally Chapter 6 will summarize the results and future works.

2 Background

2.1 Graphical Processing Units

GPUs were initially designed to accelerate graphical processing via hardware dedicated to execution of matrix calculations commonly used in image processing algorithms. In the 2000's, with the introduction of Shader Language (SL), the graphical pipeline of these processors became more malleable and general purpose computation started to be tested. By 2006 NVIDIA had seen the potential of this concept and had developed the CUDA API (NVIDIA, 2019b), capable of running native GPGPU applications on the Tesla architecture. the Fermi architecture (NVIDIA, 2006) brought forward the concept of Streaming Multiprocessors (SM). Several new architectures were released in the following years, and the concept was further developed with each new generation. Two architectures are explored in this work, Kepler (NVIDIA, 2012) and Pascal (NVIDIA, 2016).

2.1.1 NVIDIA GPU Architecture

Each NVIDIA architecture presents a different combination of the same basic elements. Except the eventual addition of a Hardware Accelerator, each chip differs only in the arrangement and proportion of elements. Recent architectures are composed of arrays of Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), and memory controllers (NVIDIA, 2016).

The concept of a SM is supported by the architectural design of SIMT (Single Instruction Multiple Threads). To take advantage of SIMT, the GPU architecture uses a hierarchical partition of compute workloads, divided in warps, blocks, and grids. Each SM is sometimes composed of hundreds of CUDA cores. Each core is capable of executing a single thread. A group of threads, called a thread block is arranged in warps, these warps are the minimal compute unit that CUDA can schedule, meaning that it allocates the same instructions to groups of 32 CUDA processors. In practice, this results in the GPU being able to perform the same operation on considerably more data than a single processor could, with minimal latency. This leaves implicit the performance gains of parallelization in GPUs, but any analyzes is incomplete if not taking in to account the CUDA execution model. Since the GPU is a separate device from the main processor, it is necessary to consider additional factors: memory transfer latency from host to device and vice-versa, asynchronicity since host and device do not share a clock, and device specific hardware constraints like number of cores and local memory. These factors are addressed by the API and drivers during implementation, but the developer must be aware of them.

The general features of a GPU remain the same for each generation since the inception of CUDA. According to (NVIDIA, 2019b), the compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. Each new architecture receive a new compute capability number. The Kepler devices used in this work are of compute capability 3.0, while the Pascal device used is of compute capability 6.0. The differences in architecture from 3.0 devices to 6.0 devices are detailed in annex H from the CUDA Toolkit Documentation. The main differences in features are shown in Table 1.

Table 1 – List of features that are present in a 6.0 device, but not in a 3.0 device.

Feature	Compute Capability	
	3.0	6.0
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No	Yes
Funnel shift	No	Yes
Dynamic Parallelism	No	Yes
Half-precision floating-point operations	No	Yes

There also exist other technical differences detailed in the Toolkit. To exemplify the architectures used, Figure 1 presents the GP100 chip of the pascal architecture, consisting of six GPCs, 60 Pascal SMs, 30 TPCs (each including two SMs), and eight 512-bit memory controllers (4096 bits total). For the same GP100 chip, each SM has 64 CUDA Cores and four texture units. With 60 SMs, GP100 has a total of 3840 single precision CUDA Cores and 240 texture units. Each memory controller is attached to 512 KB of L2 cache, and each HBM2 DRAM stack is controlled by a pair of memory controllers. The full GPU includes a total of 4096 KB of L2 cache.

2.1.2 CUDA Programming Model

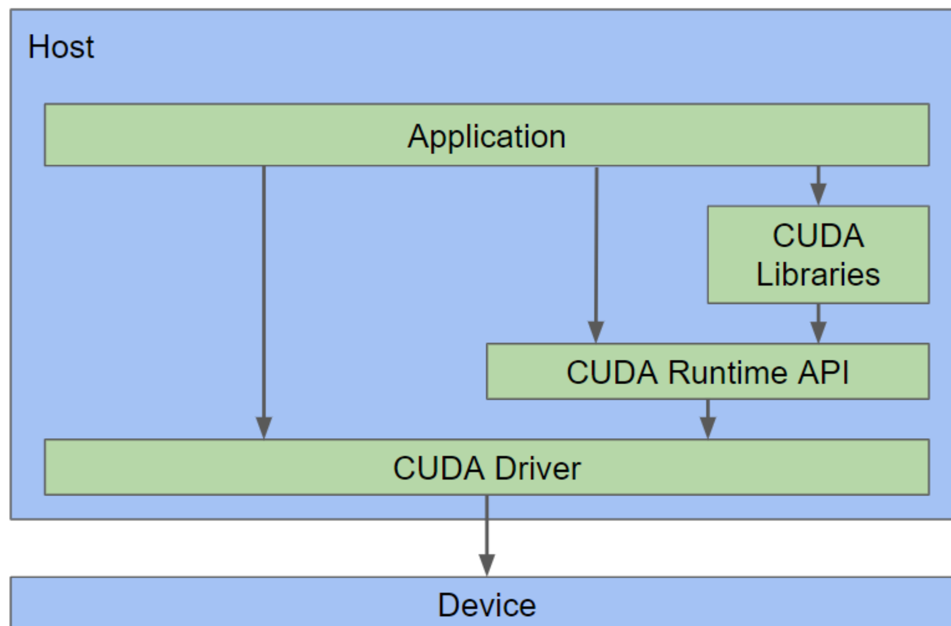
With the advent of General Purpose Computing on GPUs, came the need for a set of software layers to communicate with the GPU. Previously, any communication with the device was done exclusively through graphical drivers, and computation had to be masked as shader operations, which complicated matters. CUDA was NVIDIA's response to this problem. The Host application is usually written in C or C++ (Host code). The API allows for the application to setup the environment through direct calls to the driver. According to NVIDIA, CUDA C extends the C language by allowing the programmer to define C functions called kernels, that when called are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions (NVIDIA, 2019b). In Figure 2 the interaction between a host application and the device can be seen.

Figure 1 – Pascal Streaming Multiprocessor of the GP100 chip

Source: NVIDIA, 2017.

The host code has access to all layers, including memory transfers, sensors, and other functionalities.

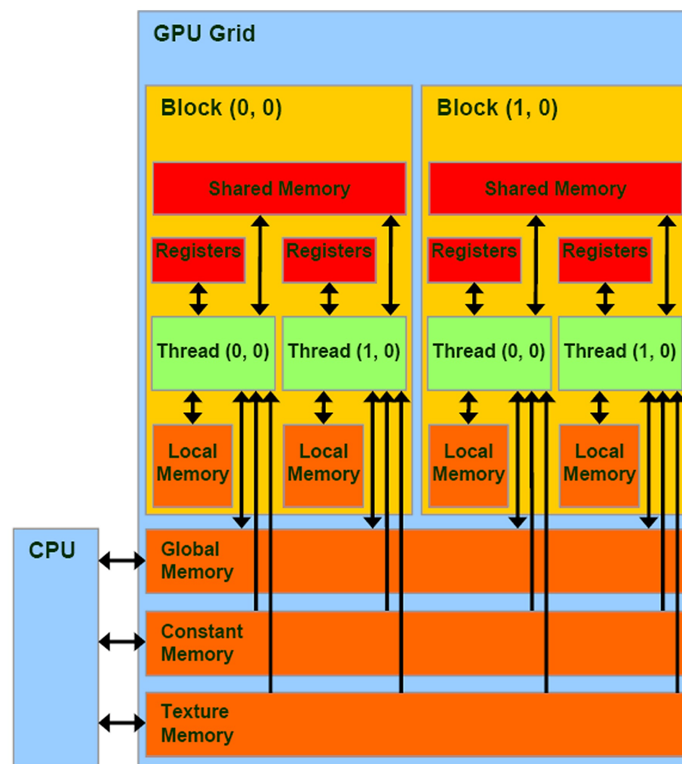
Figure 2 – CUDA Driver model



Source: Author, 2019.

CUDA kernel launches are asynchronous, unless specified by the host code, the interconnection can be further explained with figure 3. The CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. The CPU can launch kernels and copy content to the GPU memory space. With exception of some library functions, memory transfer (from device to host and vice-versa) and kernel launches are not implicit and must be done by the programmer. Memory transfer from device to host is not instantaneous and must be explicitly called. In this way, many optimizations can be accomplished by hiding the memory latency (DUANE, 2015).

Figure 3 – CUDA Memory Model



Source: NVIDIA, 2017.

NVIDIA provides many prebuilt libraries for the CUDA environment, such as cuBLAS (CUDA Basic Linear Algebra Subroutine) library (NVIDIA, 2019a). The objective of cuBLAS is to implement efficient basic linear algebra algorithms e.g. matrix multiplication; optimized for each architecture and chip by a NVIDIA team. cuBLAS is used as a benchmark for state-of-the-art research in linear algebra algorithms and usually used in applications like machine learning and image processing (SHI *et al.*, 2016). NVIDIA recently released cuTLASS a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) at all levels and scales within CUDA (NVIDIA, 2019c). It uses some of the same strategies of hierarchical decomposition and data movement implemented in cuBLAS, and gave a bigger insight in how the library

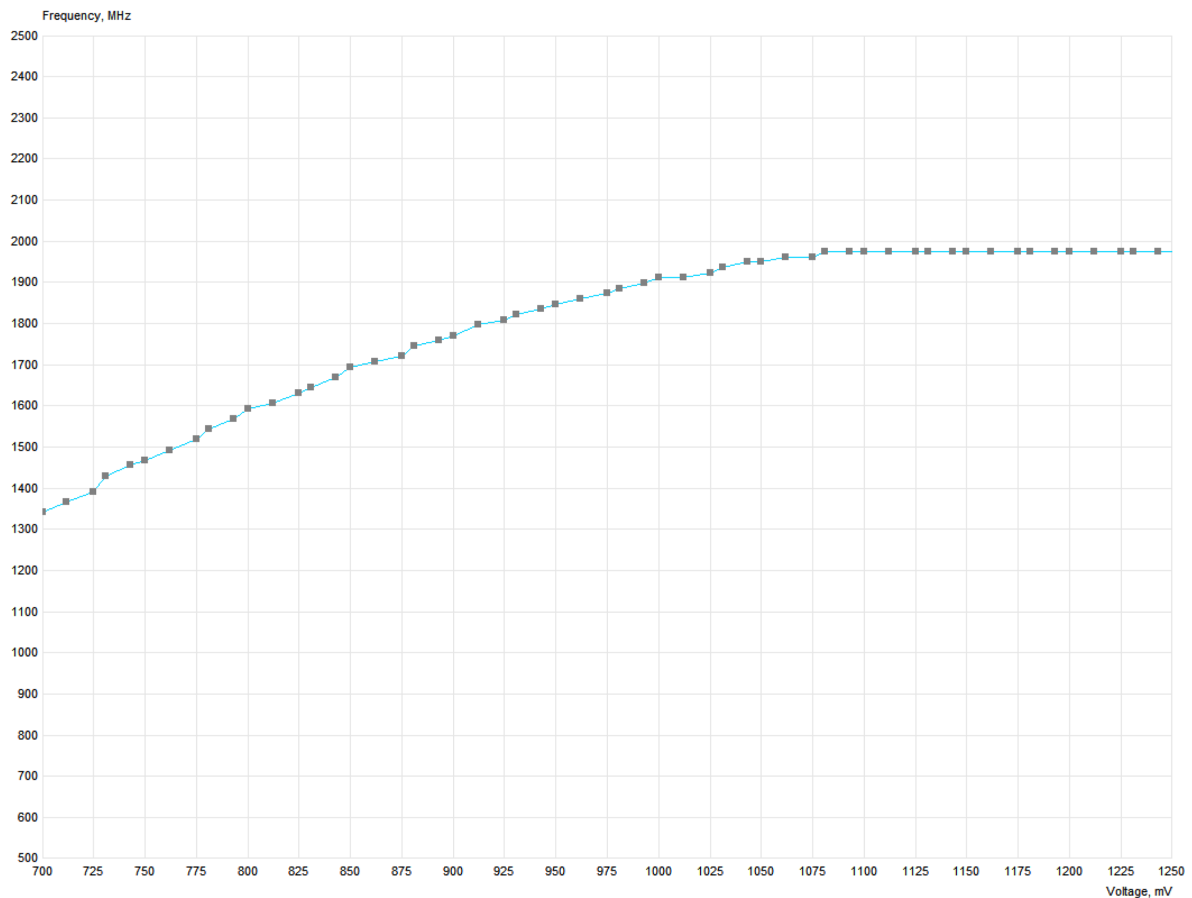
works internally. However, the exact implementation of cuBLAS is not currently made public and is differently optimized for each architecture.

2.1.3 Differential Voltage/Frequency Scaling

A common technique used in CPUs to regulate power consumption is the Differential Voltage/Frequency Scaling (DVFS) (SEMERARO *et al.*, 2002) (GONZALEZ; GORDON; HOROWITZ, 1997). The focus of the research in this area is to lower the power consumption of systems in a reliable manner. To make sense of this, it is necessary to understand the problem of critical path in a circuit. The principle behind MOSFET logic circuits is that each transistor will drive the next transistor in the pipeline to a stable state. The time it takes for a FET to reach a stable state before another load is sent is called a propagation delay. The longest chain of propagation delays in the system determines the maximum clock frequency. In order to increase the gate drive speed, and thus increase the maximum possible clock, it is necessary to increase the gate drive current. Since there is no alteration in the circuit, this can be accomplished by increasing the gate voltage (TOSHIBA, 2018). Inevitably the consequence of increasing the voltage is an increase in dynamic switching power consumption that can also result in increased temperatures. DVFS must take in to consideration the balance of critical path clock speed, circuit reliability, temperature and power consumption. Since a high enough clock speed could cause the transistors to miss a drive input and a high temperature could damage it.

NVIDIA implements an automatic control of DVFS called GPU Boost (NVIDIA, 2019d), this program allows for automatic throttling of clock frequencies and voltage scaling, based on current load, temperature and available power supply, which helps the device to maintain performance, control aging, and noise due to cooling fans rotation. This can be altered manually by the user, either via the NVIDIA Management Library (NVML) (NVIDIA, 2019e) or by using overclocking utilities like MSI Afterburner or NVIDIA Inspector. These programs allow control of operating characteristics of the devices as core voltage, memory and core frequency, maximum temperature (via fan control) and maximum power consumed. Each GPU has a predetermined Voltage/Frequency curve, that is affected by overclocking, system load, temperature and manufacturing process variations. Figure 4 presents an example curve, obtained with MSI Afterburner in a Pascal card. The vertical axis represents a range of frequencies in MHz and the horizontal axis represents the available voltages the board can safely provide in mV. While technically possible, the boards have voltage limiters to avoid damage in higher voltages. Also, the available voltages to the Pascal board extend to lower values than shown in the image. Each dot represents the possible operational points for the board, this representation is not reliable as the points tend to shift after a reboot. The curve formed from the points represent the range of values GPU Boost will operate.

Figure 4 – Voltage/Frequency curve.



Source: MSI Afterburner, 2019

In his work, Leng et al. (LENG *et al.*, 2015) found that the voltage and frequency regulation on a GPU can also affect the error rates of programs being executed on the device. Since the minimal safe operating voltage is related to the operations executed in each program and each kernel, GPUs were forced to operate while undervolting in order to find bounds for safe operating conditions.

3 Physically Unclonable Functions

According to (HALAK, 2018) a Physically Unclonable Function (PUF) can be defined as a physical entity whose behavior is a function of its structure and the intrinsic variation of its manufacturing process. On the last decade, PUFs have arrived as a new cryptographic primitive for secret key-storage and authentication. The idea behind most modern PUFs is that even tough the same manufacturing processes are used, two silicon chips, even of the same die, will not be the same, this is defined as Physical Disorder. In very large-scale integrated (VLSI) circuit technologies this phenomena can be mostly attributed to variations in the dimensions and the structures of fabricated devices.

PUFs fit in many roles and normally are associated with secure protocols, to guarantee the PUF is used as securely and efficiently as possible. Independently of design, PUFs operate in two phases: 1) enrollment and 2) authentication. During enrollment the PUF is characterized while in possession of the authentication authority, usually the server in an embedded application. The idea is for the server to store the fingerprint of the PUF, so it can later compare it with the data it receives. This data is generated by the PUF during the authentication phase, during this the PUF operates as to generate a key or fingerprint, unique for the challenges sent by the server, it then sends a response back. If a device uses the same key or fingerprint for its hole life time, it can be mapped or impersonated by an attacker. To prevent that, the server and client can perform a key-update procedure to refresh the key and increase security against brute force attacks.

3.0.1 Challenge-Response Pairs

To better understand PUFs, it is necessary to comprehend the concept of the challenge-response pair. The PUF function can be expressed as $f(.)$, denoting the complex relations of the device characteristics and unknown domain of the function. Thus for any $Challenge$, a $Response$ is given by $Response = f(Challenge)$. This defines the black-box model used in most PUF applications. Each PUF can generate a key, or ID, by answering N-bit challenges, defined as $r(n)$ e.g. N 1-bit challenges, which yield a N-bit key, or ID. Different implementations were proposed for extracting CRPs from the same available randomness, some are more direct like querying the differences in frequency of ring oscillators (KODYTEK *et al.*, 2016), or by comparing the arrival time of different inputs in flip-flop arbiters (MACHIDA *et al.*, 2015). For PUFs that use memory locations or 2D map abstractions, two proposals can be found:

3.0.2 Point Evaluation

The first implementation is a direct verification of error positions (VAN AUBEL; BERNSTEIN; NIEDERHAGEN, 2015). The PUF checks for an error at each individual position, comprised of x and y values. The challenge can be interpreted as the following question: Is there an error at position P? Described in Equation (1):

$$Challenge(P) = (x, y) \tag{1}$$

For which, the response is given by Equation (2):

$$Response = \begin{cases} 0 \text{ , if no error} \\ 1 \text{ , if error} \end{cases} \tag{2}$$

This method is flawed in multiple aspects: The number of CRPs is the same as the number of positions in the matrix, given by Equation (3):

$$TotalCRP = N^2 \tag{3}$$

It is easily mappable by an attacker monitoring the communications. After all CRPs have been exhausted, there is no more means to use the PUF.

3.0.3 Manhattan Distance

Since the simplest approach is not ideal, it is possible to acquire more CRPs adding a second point in the challenge. Based on the idea presented in (BACHA; TEODORESCU, 2015), using two points the challenge can evaluate which point is closer to an error via the Manhattan distance, as in Equation (4):

$$dist(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2| \tag{4}$$

For situations where two points are equidistant to an error, a point is designated as dominant. This process is described better by Equations (5) and (6):

$$Challenge(A, B) = (P_1, P_2) \tag{5}$$

For which, the response is given by Equation (6). Where A and B are the evaluated points and e_i is the closest error to each point:

$$Response = \begin{cases} 0 \text{ , if } dist(A, e_a) < dist(B, e_b) \\ 1 \text{ , if } dist(A, e_a) \geq dist(B, e_b) \end{cases} \tag{6}$$

An obvious advantage of this approach is the sizeable increase in the key-space. Where on the Point Evaluation approach the total number of CRPs is given by Equation (3), according to (BACHA; TEODORESCU, 2015) the total number of CRPs of this strategy can be modelled as a fully connected graph of size N^2, as shown in Equation (7):

$$TotalCRPs = \sum_{k=1}^{N^2} k = \frac{1}{2}N^2(N^2 - 1) \tag{7}$$

Besides an increase in the total number of CRPs, this strategy is also more secure. If an eavesdropper is monitoring communications between server and client, it could possible model the naive approach before it exhausted the CRPs. In this manner, an attacker will not get direct information about the location of errors, allowing for the same error to be used more than once.

3.0.4 Cryptographic Primitives

A Cryptographic Primitive can be described as a highly reliable algorithm, designed for one specific purpose, which secure system designers can use as basic building blocks (HALAK, 2018). These primitives can be classified in a manner of ways. Symmetric Ciphers, like the Advanced Encryption Standard (AES) (DAEMEN; RIJMEN, 2002), are two-way functions designed to encrypt and decrypt messages sent between two parties which share a common key. Asymmetric Ciphers on the other hand, serve as basis for key-exchange protocols, where each party has a trusted public-key which makes possible encryption, but not decryption. Thus, only the owner of the public key will be able to decript the message with his own private key. A prime example is the RSA cryptosystem (HINEK, 2009).

One-Way Hash Functions, like the SHA series of algorithms (BAKHTIARI; SAFAVI-NAINI; PIEPRZYK, 1995), map an arbitrarily long input to a short fixed-length output. Hash functions are used for a number of factors: they provide a deterministic output based on the input; it is extremely difficult to retrieve an input based on the output; it is unfeasible to find two messages that create the same hash; and finally, there is an avalanche effect, a small change in the input leads to a significant change on the output. Random Number Generators (RNGs) (BHATTACHARJEE; MAITY; DAS, 2018) are also considered a cryptographic primitive, due to their inherent resistance to modelling attacks. They are usually used to generate a nonce, a number that is used only once to avoid replay attacks.

There are other cryptographic primitives as well, but the idea remains the same, a basic building block that a designer can trust to be secure and reliable. As more attacks and vulnerabilities are discovered, new protocols, standards and algorithms must be developed to fill the gaps. PUFs are considered cryptographic primitives, but rather than depending on an algorithm, their security lies on being hard to model.

3.0.5 PUF Applications

The idea of identifying objects through unique patterns is not emergent. During the cold war, nuclear warheads would be spray-coated with a reflective substance (GRAYBEAL; MCFATE, 1989). The resulting reflective pattern would be evaluated from multiple angles and would be practically impossible to reproduce, rendering it unique and unclonable. PUFs have a wide range of applications and not all PUFs are ideal for every application. Different access times, total number of CRPs and run time availability might limit it's usability. Two recurring applications of a PUF on the literature (KODYTEK *et al.*, 2016) (MACHIDA *et al.*, 2015) (HALAK, 2018) are summarized as follows:

Secure Key Storage: Symmetric and Asymmetric encryption protocols rely on private keys, that are usually embedded in the device memory. These keys can have a fixed life-time or changed per discretion of the owner in order to increase security and avoid the key being discovered. The problem with this is that the key will need to be accessible to the device and be stored in a memory location. This way, an attacker could steal the key, either via hardware (e.g. probing the memory wires) or via software. PUFs, can be used as Physically Obfuscated Keys (POKs), meaning they only exist inside the PUF. Whenever the device needs to run an encryption it extracts the key from the PUF. Since the key is dependent on the physical parameters of the PUF, any attempt to temper with the device will destroy the key.

Low-Cost Authentication: PUFs can be used to authenticate devices, the variability in the manufacturing process provides a suitable fingerprinting method. Some PUFs like ring-oscillator or arbiter PUFs have a simple structure, making them easier to fit in a resource constrained system (e.g. IoT) than to store millions of different keys or authentication pairs in memory. The idea of designing a PUF without the need for additional hardware also belongs to this category, as it allows for existent and already deployed hardware to generate a PUF.

3.0.6 Previows works on PUFs

Most of the work related to creating Intrinsic PUFs is focused on using memory elements like SRAMs or DRAMs. One of the first works to present a SRAM PUF is (GUAJARDO *et al.*, 2007). In the paper, a technique is described to collect the initial state of uninitialized SRAM cells. As each cells has a preferred start-up value of either 0 or 1. In addition, recent research has been exploring new ways to exploit memories in order to realize PUFs. The work (BACHA; TEODORESCU, 2015), described a SRAM PUF named Authenticache, it uses cache Error Correction Code (ECC) to mount an error map due to undervoltage of the cache memory (SRAM). (KIM *et al.*, 2018) presented DRAM Latency PUF, which reduces the timing parameters of the memory and evaluates the resulting error patterns. An yet crudely explored alternative are creating PUFs in the

Graphical Process Units (GPUs). An advantage of using GPUs over memories is that any user has sufficient access rights to generate a PUF, while most memory PUFs could be limited by the virtualization of cloud server environments.

The idea of using GPUs as a source of entropy has already been explored. From 2012 through 2015, the PUFFIN Project (LANGE, 2012) investigated the existence of PUFs in commodity PC components. The project had a total budget of 1.5 million euros and was the combined work of Technische Universiteit Eindhoven, Technical University of Darmstadt, Katholieke Universiteit Leuven, and Intrinsic ID. The motivation behind the project was mainly to generate fingerprinting mechanisms for device authentication, allowing for a secure root to be established in each device. The project resulted in the discovery of SRAM PUFs in GPUs, described in the work of Van Aubel et al. In the article, the authors were able to access uninitialized memory space in the GPUs global memory (VAN AUBEL; BERNSTEIN; NIEDERHAGEN, 2015). They were also able to map the probabilities of each bit initializing to either a 1 or a 0. Bits that had statistically higher probability of resting in the same state every time, were used for the generation of a device ID. On the other hand, bits that were in a meta-stable state had the same probability to rest in a 0 or a 1, thus were used as True Random Number Generators (TRNG) for cryptographic key generation. The problem, however, was that this behavior was only observed in older NVIDIA GPUs, since other brands and more recent cards all initialized their memories at boot, since this is considered a critical security design flaw. Moreover, this technique required the bits to be handpicked by the user/program in a setup phase, the authors do not discuss the viability of this, or the fact that inevitably the chosen bits would have to be stored on the memory for later use, reducing the security of the measure. Also, this behavior is intrinsically limited to the period right after the memory is booted, potentially requiring any program that used this behavior to also store tables of pre-computed values on the devices memory.

Nithyanand et al. describes in (NITHYANAND; SION, 2011) a methodology to use the delays and differences in frequency of different components such as memory buses, CPUs and GPUs to mount a PUF. The main idea of the paper is to embed the responses of the PUF in the computation done by a software that needs to be protected against counter fit. This way, the shortcomings of a black-box PUF are nullified e.g. replay attacks. Li et al. outlines the possibility of using the core frequency variation in CUDA cores as a source of entropy for PUFs (LI; FU; LUO, 2015b). This variation stems from the fact that process variations in the production of the chips, as dopant fluctuations and lithographic aberrations can cause significant variations in transistor manufacturing. They state that the ratio of frequencies between the fastest and the slowest core in a GPU chip can reach as high as 2.2. They aimed to produce a single CRP, meaning a POK, to generate a device ID. The disadvantages of this technique are directly related to its dependency of external circuitry to evaluate core frequency. Commercial GPUs in general have no interest (or

need) to monitor individual core frequency, so there is no built in hardware for this task. This in turn prevents the general usage of this technique. Also, the authors state that further work is required for the proposal to become a full fledged ID mechanism, since a single CRP is easily reproducible by attackers, and to effectively balance the effects of aging in core frequency. Finally, Chauvet and Mahé describe in (CHAUVET; MAHÉ, 2013) the usage of the massively parallel structure of the GPU as a deep entropy pool. The authors present a novel cryptography architecture that aims to restrain key acquisition, management and computation exclusively to the GPU. The GPU is used as a specific random bit generator which uses CUDA kernels to extract random data from the hardware. The acquired data is then fed to a Pseudo-RNG, which is used as a cryptographic key. The paper focuses heavily on the development of efficient parallel mechanisms to perform RSA cryptographic computations. In this manner, the key and algorithm are kept in the device, reducing the amount of memory transactions and increasing the security. Unfortunately, the mechanisms through which entropy is collected from the GPU remain undisclosed, since the application is a commercial product. Also, there is no mention of using the current architecture as an means for device identification.

3.0.7 PUF Metrics

Following the black-box model, these variations cannot be modelled given the principle of Physical Disorder explained in the beginning of this section. Such variations can occur due to manufacturing variability, but also due to transient factors such as temperature, voltage, frequency and aging. Process variation will manifest mostly in an inter-chip analysis, while intra-chip variations correspond to fluctuations in environmental and operating conditions (HALAK, 2018). The following metrics are the most referenced metrics in literature (HERDER *et al.*, 2014) (HALAK, 2018) (IYENGAR; RAMCLAM; GHOSH, 2014) (BACHA; TEODORESCU, 2015), given that they represent the most basic PUF parameters.

Uniqueness: This is the guarantee that each PUF will be unique, even among identical chips, due to variations in the manufacture process. There are two ways to represent this metric, either as a direct analyses of the distribution of inter-chip Hamming Distances, or by calculating Equation (8):

$$Uniqueness = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(r_i(n), r_j(n))}{n} \times 100\% \tag{8}$$

Where,

k are the different devices being evaluated;

r_i and r_j are the n-bit responses from each device i and j for the same challenge;

The ideal value for this metric is 50%. At this value, the metric represents a random distribution of differences for each position in two responses from different devices. From the PUF perspective, this means that different devices are as different as if the responses where randomly generated.

Reliability: It is important to know how deterministic a PUF is. Most PUFs are susceptible to environment variations like ambient temperature. In the case of G-PUF it is trivial to maintain temperature control inside the chip since we never deviate from nominal operating temperatures, so instead we evaluate the stability of the CRPs for different samples. This metric utilizes intra-chip HD, as shown in Equation (9):

$$Reliability = (1 - \frac{1}{k}\sum_{i=1}^{k}\frac{HD(r_i(n), r_{i-1}(n))}{n}) \times 100\% \tag{9}$$

Where,

k is the number of different samples being analyzed;

r_i and r_{i-1} are the n-bit responses for the same challenge;

The ideal value for this metric is 100%. This value means that the PUF has deterministic a response for each challenge.

Uniformity: To avoid modelling attacks, it is important that to an attacker the responses of a PUF seem as random as possible. In a set of binary words, this appears as an equal proportion of 0s and 1s. This is achieved by calculating the average Hamming Weight (HW), which is a measure of the number of bits $! = 0$, defined by Equation (10):

$$Uniformity = \frac{1}{k}\sum_{i=1}^{k} HW(r_i(n)) \times 100\% \tag{10}$$

Where,

k is the number of responses;

r_i is the response for any given challenge;

The ideal value for this metric is 50%. This value indicates that the responses from the PUF are equally distributed between 0s and 1s.

4 Methodology

This chapter provides a description of the application's design and the steps required to extract and characterize the G-PUF. The first section is a functional description of each step in the application, that will also serve as a basis to explain the underlying mechanisms of the PUF. The second section will explain the novel technique developed to extract the CRPs from the device.

4.1 G-PUF

G-PUF is an application developed to provide a GPU with the ability to generate a unique device ID and criptographic keys as needed. It accomplishes this by executing cuBLAS multiplications on the GPU while under adversarial conditions. It uses the resulting soft-errors to produce a 3D map of the most frequent error positions in the resulting matrix. This map is then processed and the results for each challenge are used to mount an ID or key for the device. It can be used by both a server and a client on secure protocols as a cryptographic primitive.

4.1.1 Golden Matrix

G-PUF first creates two random square matrices, hereby denoted as A and B of size NxN containing double variables. The matrix of doubles is used to guarantee maximum stress level and occupancy on GPUs of any architecture. A fixed seed is used to create the random matrices, the reasoning behind this is two fold: random numbers will increase the chance that different cores will receive different numbers to compute and that the tests will be repeatable. An analysis of errors between different calculations will not be evaluated on this work. The size of the matrix is decided by the designer in order to guarantee maximum occupancy in the GPU, which is a relation between the number of cores available and the version of the cuBLAS implementation. cuBLAS is a highly efficient implementation of the BLAS library in CUDA. An initial run of this function is stored and defined as a Golden Run, $Golden = AB$, which we obtain in normal operating conditions and can guarantee it's correctness.

4.1.2 Thermal Variation

G-PUF collects CRPs from core operation, and GPU temperature is directly related to the devices load, voltage, and frequency, it becomes difficult to maintain a constant temperature target during tests. Although the effects of temperature were observed during

testing and empirical analysis showed improved error/pass rates for certain temperature ranges, G-PUF only has a pre-heating phase where temperature is elevated to the average achieved during CRP collection. This procedure avoids an oscillation of temperature larger than a few degrees Celsius during the collection of CRPs, instead of a possible difference of 20 degrees between the idle state and continuous load. To achieve this, the program simply queries the current temperature of the GPU using the NVIDIA Management Library and executes the same computation until it reaches a predetermined temperature target. In our tests this target was determined via an empirical analysis. This value is also set by the designer and tends to be a conservative estimate.

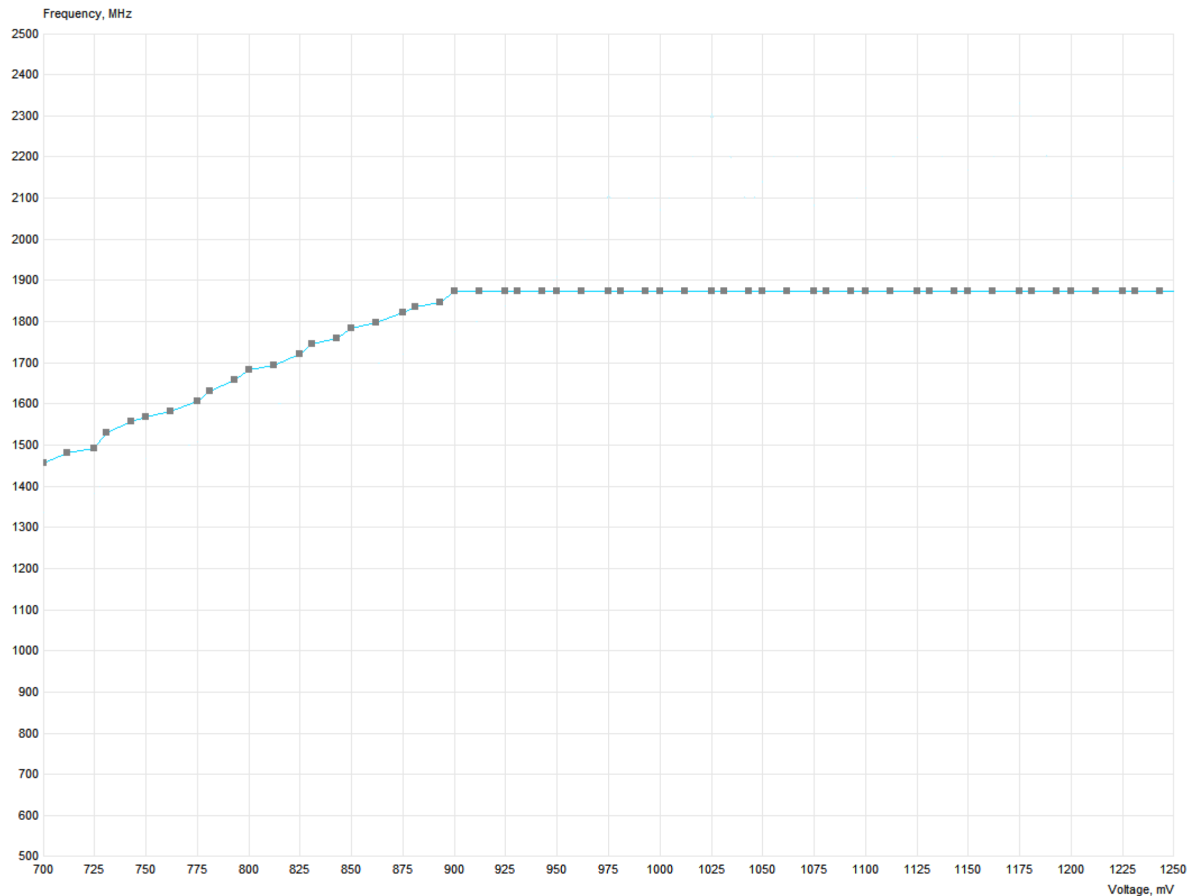
4.1.3 Changing Parameters

The GPU drivers for NVIDIA boards in Windows operating systems offer a good control for the GPU's operational parameters. As described in section 2.1.3, the GPU has a predetermined voltage/frequency curve. GPU Boost will select the operating point based on a number of factors, including: current load, temperature, power headroom, and other selectable parameters. It is possible to induce errors in the GPU when the clock frequency is increased beyond the voltage level can maintain in a stable state. This can be achieved by different means. First, a technique that is frequently used in DVFS research, is to fix a frequency value and start to lower the board's voltage in fixed steps. This technique is viable, and could be achieved by using the NVML. Extensive research yielded little to no results on how to use the library to set parameters directly. To circumvent that, third party programs were used.

Using the MSI Afterburner software, it is possible to directly change the curve for Pascal and newer architectures, flattening it in the graphical interface as shown in Figure 5. To achieve this the board needs to be overclocked, this will offset the curve upwards in the frequency axis, then points with higher voltage than the selected voltage point must be manually brought down to the same frequency level. This will cap GPU Boost's ability to increase the voltage, since it considers only the highest frequency achievable in a stable state. The problem with this technique is that it requires a manual change of settings at every change in operating conditions and also is not available for Kepler and Maxwell devices.

NVIDIA Inspector allows to change core voltage, power limits and frequency via the command line. The available commands allow the user to ignore GPU Boost and fix voltage values, to reduce available power supply up to 20%,and offset core frequency. Since the voltage/frequency curve does not present a linear behavior, calculating the offset for each new GPU added would require inside knowledge on the implementation of each board depending on sub-vendor. Thus, for Pascal devices the chosen technique relied on finding the lowest viable operational voltage for the GPU and then increasing the clock offset until

Figure 5 – Flattened Voltage/Frequency curve.



Source: MSI Afterburner, 2019

errors appeared. The lowest voltage point is defined as a voltage and frequency pair that allows for a cuBLAS matrix multiplication to be executed without crashes for 100 passes. The downside is that this exploration phase can lead to a system crash, which would require a manual reboot. For Kepler devices, voltage controls are not fully supported, so instead the power limiter is used. This process is more complicated, as limiting the power does not deactivates GPU Boost, and the actual clock will vary depending on current load, temperature and the selected power range. Since the available range for power variation is smaller than the available range of voltage values, a characterization test on the GPU became possible, where each defined clock offset is tested 100 times for each possible power reduction setting.

4.1.4 Sample Generation

The process of sample generation is straight forward. At the start, the GPU is undervolted via a system call to a bash script. Next, matrices A and B are given as input for a cuBLAS GEMM matrix multiplication, which will result in a tainted matrix. In order

to acquire the largest number of error samples in an efficient manner, the program will store each output matrix in the memory adjacently.

Previously the program calculates how many matrices of size NxN fit in 80% [1] of the device's memory and defines the maximum number of consecutive multiplications. Each of these multiplications is defined as an iteration and each time the memory is completely filled, a pass is completed. After a pass, a system call is issued so the voltage and frequency are reset.

4.1.5 Compare and Store

Each completed pass will be passed to a CUDA kernel responsible for comparing each position value to that of the golden matrix. Each position that contains an error will be stored, along with the iteration in which the error occurred. Since the outpu matrix can be represented as memory positions in a 2D space of size NxN, each error will map to a single position.

The position of each error in the execution is dictated by three variables, the hardware and driver layers, and the software used. The algorithms used by the warp-scheduler and the CUDA driver API (hardware and firmware respectively) are not usually divulged by NVIDIA, so we can threat them as a black-box for the purpose of the PUF. In regards to software, even though the exact mechanisms behind cuBLAS are hidden, it is possible to say that the matrix multiplication is done by dividing the whole matrix in sub sectors, and each position being assigned to a thread, which eventually maps to a single core.

When lowering the voltage and overclocking, computational errors will manifest themselves at the physical core, regardless of what position of the output matrix is assigned to it. With this in mind, it is possible to use the error position as a marker to track the faulty core. Several acquisitions of errors on the matrix will reveal the pattern in which the threads are allocated. If one or more cores are faulty, their virtual position will shift in the output matrix, when multiple runs are evaluated, thus providing an error map.

4.1.6 Launch Loop

During the undervolted multiplication process, there is the possibility that an error occurs in the control logic of cuBLAS which results in a sticky error. Sticky errors are errors that destroy the primary CUDA context in the application e.g. segmentation fault. The primary context holds all the management data to control and use the device e.g. list of allocated memory, loaded modules and specially mapping between CPU and GPU

[1] This value is decided based on the available memory in the GPU to avoid the risk of overflowing the memory.

memory. When the context is corrupted, all subsequent calls to functions in the device will cause exceptions. To solve this, it is necessary to kill the application that launched the CUDA context.

To avoid the need for human oversight of tests, a launcher program was designed. This program is responsible for launching the G-PUF application and managing result permanence from one run to the next. Thus, it is possible to define a number of desired error samples before the application ends.

4.2 PUF Protocol

An abstraction will be used to make the protocol generic in terms of applications: Alice and Bob are two parties trying to communicate in an unsecured channel. The channel is insecure since neither Alice nor Bob can be sure the channel is not being monitored by a malicious third party named Eve.

4.2.1 Enrollment

On this phase, Alice is in possession of the device and must collect the error map. The first step is to define an operational point for the clock and voltage/power supply. This is done by executing a pass, checking for errors and if none are found, lowering the voltage/power supply by one step and increasing the clock by one step. If errors are found, the operating point has to maintain errors for at least 10 passes before being admitted, otherwise it changes the settings again, this is to ensure a consistent sampling of errors. Different operating points offer different error rates, it should be possible to develop an optimized strategy to increase error rates, but this was not evaluated in this work. With the operational point defined, the device begins to be characterized, and an error map assembled. The time taken on enrollment will vary from device to device, as it is directly correlated to the time it takes to execute each pass. The collection stage ends when there are positions with 10 times more errors than others. At the end of the enrollment, a full error map is collected, allowing for Alice to store only a N^2 sized map, instead of all CRPs listed, and to hand-over the device to Bob.

4.2.2 Authentication

In the authentication phase, Alice sends a series of M challenges to Bob, corresponding to a M-bit key. Bob will then generate a local error map. The collection stage for Bob won't be as exhaustive as it was for Alice, since Bob is only interested in the most common error positions. After this is complete, the evaluation stage begins, where Bob will count the errors inside each region, compare them and generate a response for each challenge sent by Alice. At this moment, both Alice and Bob can agree on the same key,

generated by the PUF, or Bob can send the key back to Alice in order to verify his identity. During the whole communication, all that Eve could intercept was Alice's challenge and Bob's response. Given the randomness of the response, Eve cannot predict Bob's response for any of Alice's challenges.

4.2.3 Key Update

The last phase occurs between enrollment and authentication, as well as when never deemed necessary by the protocol. This phase involves Alice sending helper data to Bob via the insecure channel. The data is comprised of the clock and voltage/power values, as well as the threshold defined for the error map. This way Bob can refresh the number of CRPs available by changing the threshold.

4.3 CRP Extraction

After the G-PUF has collected the error samples, the extraction of response bits begins. This description abstracts any secure protocol and instead focus on the algorithm used to extract responses for a given challenge. In the following scenarios, a server is tasked with acquiring a 128-bit key from a client equipped with a GPU running the G-PUF application. For the proposed CRP extraction methods, each CRP corresponds to a single bit in a cryptographic-key. So, for a 128-bit key, a total of 128 CRPs are needed.

All of these methods deal directly with the error map. On the enrollment phase, all that it is required is to store the error map of the GPU. The advantages of this are two-fold, first there is no need to have all of the CRPs pre-calculated and stored in the server, this can save a lot of space for a server that contains a lot of enrolled PUFs. Second, since the key is contained in the error map, challenges can be transmitted publicly as well as responses, since each challenge will only be used once.

The procedure for extracting the error map starts when the client receives a call for identification. Since the operation takes a relatively long time, the client can start collecting samples in the down time from computations. Due to the nature of the errors, and the algorithms discussed in section 4.1.5 the error map can be abstracted as a 2D matrix, the same size of the result matrix, with a Z axis referring to the accumulation of errors in the same virtual position. The more errors are collected, the easier it will be to find preferred positions for the occurrence of this errors, dictated by the mechanism presented in section 4.1.5 and the inherent variation on the manufacturing of the chip. Depending on the level of security or reliability desired, the client can choose the number of samples to be collected and also a threshold to evaluate only the most reoccurring errors. When it has collected enough samples to generate an error map with the required size, it will be ready to generate an ID.

4.3.1 Circle Area

Even though the traditional strategies presented in section 3.0.1 can increased the number of CRPs, they are still dependent exclusively on the size of the matrix, which for realistic sizes still provides a relatively low number of CRPs. To solve that, one approach is to increase the variables used to evaluate a response. Taking advantage of the nature of the 2D virtual mapping of the the matrix, we instead evaluate regions of the error map instead of individual errors. The regions will be created using algorithm 4 to create a mask on top of the map, and then count the values inside the circle.

Algorithm 1: Algorithm to create a mask of radius R and center c_x, c_y.

1 Create Circle Mask $(X, Y, center, radius)$;
Input : Two arrays of integers X and Y of size N, an array with two integers $Center$ bigger or equal to 0 and smaller than N^2, and a positive integer $radius$ bigger than 1 and smaller than N/2
Output : Binary position map of size N^2 with masked region set as True
2 c_x, c_y = center;
3 dists_sqrd = (X_arr - c_x)**2 + (Y_arr - c_y)**2;
4 **return** dists_sqrd <= radius**2;

The challenge in this case consists of two center points and a radius as in Equation (11). The center points A_c and B_c will be located inside the 2D error matrix and the radius R will create two circular areas A and B, restrained to the borders of the 2D map.

$$Challenge(A_c, B_c, R) = (P_1, P_2, R) \tag{11}$$

The response will be evaluated by counting the number of errors inside each area and evaluating which one has the most errors. This can be expressed by Equation (12):

$$Response = \begin{cases} 0 \text{ , if } count(A) < count(B) \\ 1 \text{ , if } count(A) \geq count(B) \end{cases} \tag{12}$$

The total CRP is given by Equation (13):

$$TotalCRPs = \frac{1}{4}N^2(N^2 - 1) * (N - 1) \tag{13}$$

With this method, an attacker monitoring the communication won't be able to pinpoint exact error positions. Also, this allows for an expanded CRP space in comparison to Equation (7), since it also derives from the combination of two points in the matrix, but has the additional dimension of radius values. However, the two main advantages will become clear when the resulting patterns of errors are presented. First, the errors in GPUs come in groups and sometimes distinguishable patterns, it would become easy to guess

the position of errors by evaluating the distance of different points. Second, most error maps present regions of high concentration of errors, only a few positions present the same error every time, with this technique its possible to consider the variability of placement of these regions.

4.4 Validation of Results

For each board, a total of 10 million individual errors will be collected. This number was decided based on preliminary tests with the GTX 1060 board, which should an stabilization of preferred error positions before this number of errors. Each board will then go in to a enrollment phase, where full error maps are collected. Uniqueness and uniformity will evaluate the full error map of each board with 100 random challenges for each test. The tests to evaluate reliability will be executed by randomly sampling error passes and assembling 10 partial error maps, these partial error maps will be then evaluated against each other in different combinations 10 challenges each. Resulting in a total of 450 challenge-response pairs being evaluated for each board. Smaller tests will also be conducted with different key-sizes to evaluate how it affects this metric.

Uniqueness and reliability will yield two distribution curves of each sample's Hamming Distance to another board response and to another partial error map's response, respectively. Ideally uniqueness would result in a stacked column of samples in the 256 bit mark, but in a more realistic and desirable scenario, it will present a normal curve centered around the 256 bit mark, meaning a indistinguishable response from a random response. Reliability in the other hand ideally would present all the values stacked at the 0 bit mark, but in a more realistic scenario it should present a normal distribution concentrated to the left of the plot, meaning low distances between responses.

5 Results and Discussions

5.1 Experimental Setup

The experiments were conducted in three different boards, two NVIDIA GTX 680 from the Kepler architecture with 2 GB of DRAM memory, 1536 CUDA cores and driver version 441.22, hereby denoted GTX 680 - 1 and GTX 680 - 2; and one NVIDIA GTX 1060 from the Pascal architecture with 6 GB of DRAM memory, 1280 CUDA cores and driver version 418.86. All tests were executed with CUDA driver version 10.2. As previously stated, NVIDIA Inspector was used for overclocking and undervolting. Pascal boards have a predetermined voltage/frequency curve, which is accessible and provides direct control over operating conditions. Kepler does not have this support, so in order to undervolt, the same program was used to alter the power limiter and starve the board.

5.2 DVFS Characterization

The first step to create the PUF was to prove that this method could reliably generate soft-errors. The first version of the program was developed for the GTX 1060 board and focused on the exploration of different ways to extract errors. Multiple benchmarks where tested, but most NVIDIA samples did not put the GPU under considerable load e.g. less than 1% total load. To circumvent this, cuBLAS was selected to execute matrix multiplications capable of occupying all the cores with minimal memory latency. Then, different combinations of voltage and frequency were manually selected until errors started to appear.

A second version of the program was then developed, containing further automation, which allowed for the characterization of devices. Both GTX 680 boards where characterized, as can be seen in Figure 6 and Figure 7. These graphs represent the mean of all errors collected in 100 passes for each combination of frequency and power supply evaluated. As stated before, NVIDIA drivers do not allow for easy voltage control in the Kepler, so instead the power is controlled. The power controls range from 0% to 20% reduction. A distinct frequency range was chosen for characterization, this range allowed for errors to occur with higher power reductions. This was desired, as increasing the clock frequency with power settings close to stock values elevated the temperature past the safe limit. Thus, GPU Boost would start to throttle down the frequency, as it still has control over the board. Both Figures represent a heat map of values, represented by a logarithmic scale on the color bar, this was needed as higher clock frequencies and higher power supply induced several thousand more errors per pass than lower settings.

Figure 6 – Error heat map for the GTX 680 - 1 with a logarithmic color scale.

Figure 7 – Error heat map for the GTX 680 - 2 with a logarithmic color scale.

5.3 Threshold

Selection of threshold values is a complex matter in this scenario. Since both Alice and Bob must agree on a threshold value that will allow for a less complex and time consuming error collection phase during authentication. Using the same scenario described in section 4.2, Alice will perform the enrollment and collect a more complete error map, since she is not constrained by time in this phase. This error map collected by her is called a full error map, meaning it contains many more samples than the error map collected during authentication, which gets the name of partial error map. Further study is necessary to evaluate how many samples exactly are enough to differentiate a full error map from a partial error map. In this work, a difference of at least 10 times more errors is expected. The threshold values will vary from device to device, but specially from architecture to architecture due to different error distributions. Table 2 presents the number of errors for each GPU's full error map in different threshold values. This difference can be better seen when plotting these values in a logarithmic scale, as show in Figures 8, 9 and 10.

Table 2 – Number of errors in the full error map by threshold value for each GPU.

Threshold (%)	**GTX 1060**	**GTX 680 - 1**	**GTX 680 - 2**
5	342158	26389	911087
10	279157	644	119516
15	241746	302	8660
20	214574	197	1596
25	176187	133	1047
30	134719	62	744
35	111169	32	517
40	75826	23	427
45	48804	20	378
50	26671	17	356
55	17953	16	313
60	10259	16	243
65	6660	16	152
70	4120	8	79
75	1539	6	46
80	1026	3	21
85	1025	2	12
90	516	2	6
95	6	1	3

As expected, the number of errors diminishes as the threshold increases. A promising signal for the PUF is that each board behaves differently from the other, both in the number of errors and the scale that they vary. The results presented in Figures 9 and 10 had to be presented with a logarithmic scale, otherwise the results past 15% were unnoticeable in the graphs.

Figure 8 – Number of errors per Threshold percentage in a logarithmic scale - GTX 1060

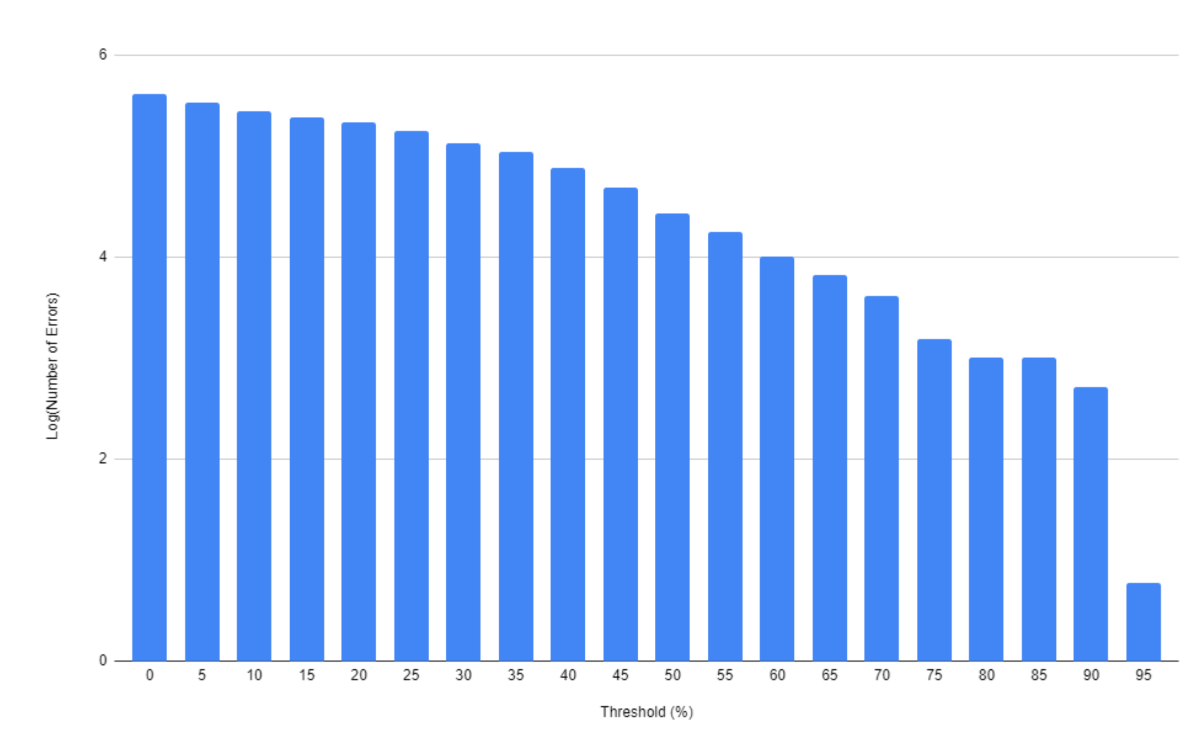


Figure 9 – Number of errors per Threshold percentage in a logarithmic scale - GTX 680 - 1

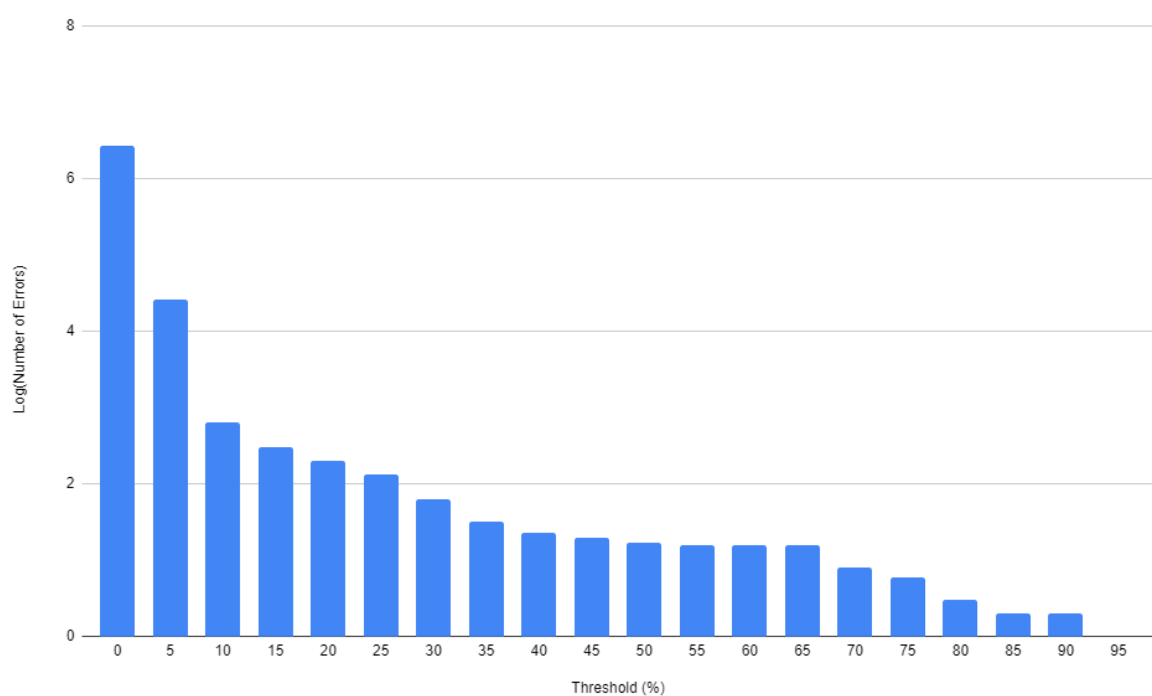
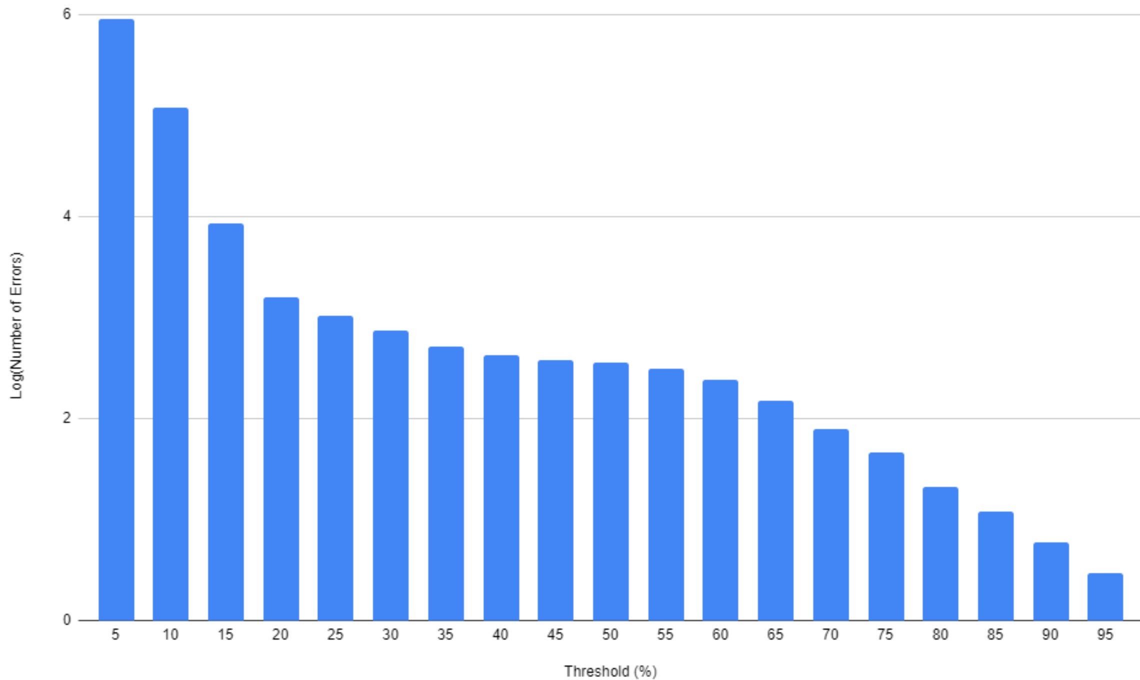


Figure 10 – Number of errors per Threshold percentage in a logarithmic scale - GTX 680 - 2



The difference in architectures can be perceived in the histogram. The Kepler boards have shown to spread the errors more than the Pascal board. This effect is visible when evaluating the full error map for each board and the effect of the threshold. As can be seen in Figure 11, the Pascal board presents a predictable pattern in the error distribution. More than 90% of the errors appear in groups of 512 equally spaced in a 16 columns by 128 lines, the rest of the errors appear individually or in smaller groups. The Kepler boards on the other hand present a more hectic behaviour as can be seen in Figures 12 and 13, with error appearing individually or in groups of 16, either horizontally or vertically. This difference in behaviour can be attributed to different optimizations applied to each architecture in the cuBLAS implementation as well as different allocation mechanisms in the hardware e.g. half warps present in Kepler.

It is also interesting to note that both Kepler boards present a similar error pattern on the left side of the Figures. No means were found to verify why this behaviour happens, but it can be speculated that it has to do with the order that the matrix is copied from main memory to the cache, or even perhaps some unknown race condition when the GPU first receives the kernel. Even though they present different behaviors, the fact that only a few positions repeat errors frequently in the Kepler board is problematic for the PUF. This could mean that the authentication phase for this architecture could take longer, since it would need to collect more errors before being able to filter them with the threshold, or even that they could have a stronger bias.

Figure 11 – Plot of Full error map with no threshold and 50% for GTX 1060. White points represent error positions.

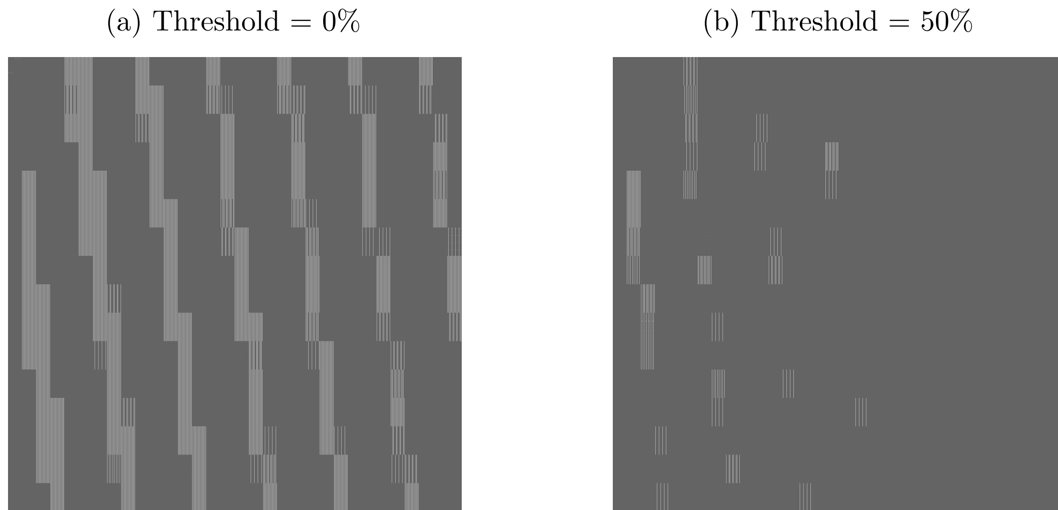
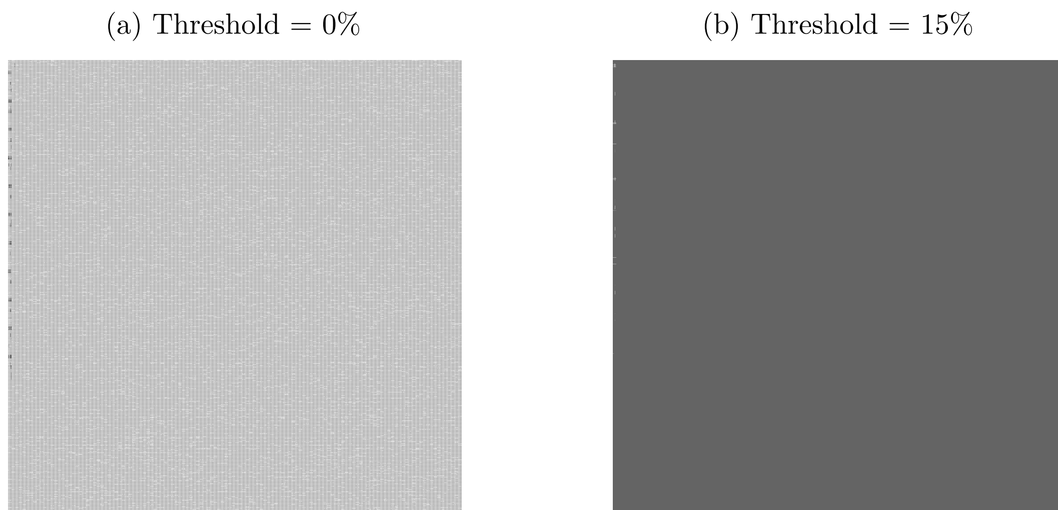


Figure 12 – Plot of Full error map with no threshold and 15% for GTX 680 - 1. White points represent error positions.



5.4 PUF Metrics Evaluation

5.4.1 Uniqueness

The uniqueness metric can be evaluated by exploring the differences between devices from the same architecture (intra-arch) and from different architectures (inter-arch). The intra-arch comparison was made between both GTX 680 boards and the inter-arch comparison was made between the GTX 1060 and GTX 680-2. By calculating the Hamming Distance between responses for the same challenge in different devices it is possible to get the distribution of distances for each comparison. In Figure 14 it is possible to see the distribution of Hamming Distances for 100 challenges of 512 bits.

Figure 13 – Plot of Full error map with no threshold and 15% for GTX 680 - 2. White points represent error positions.

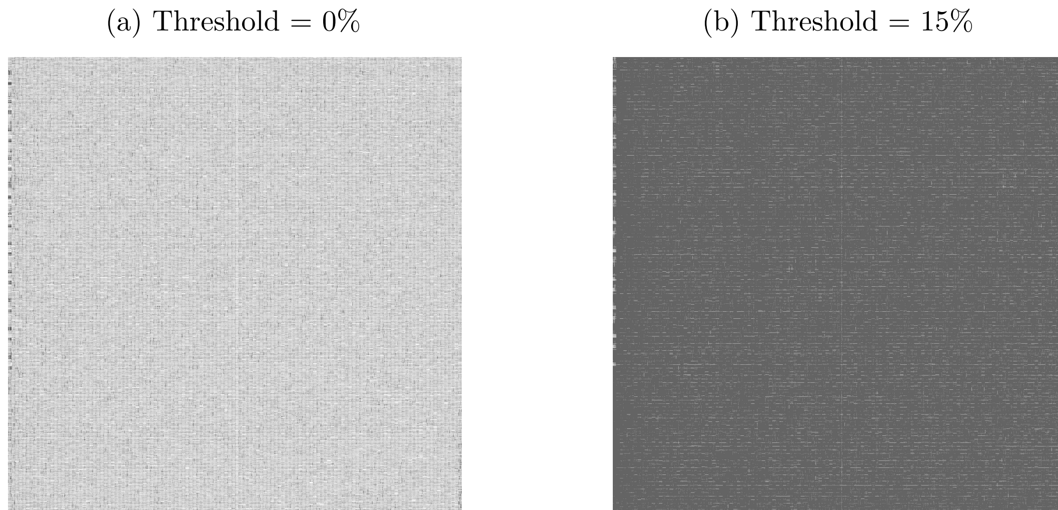
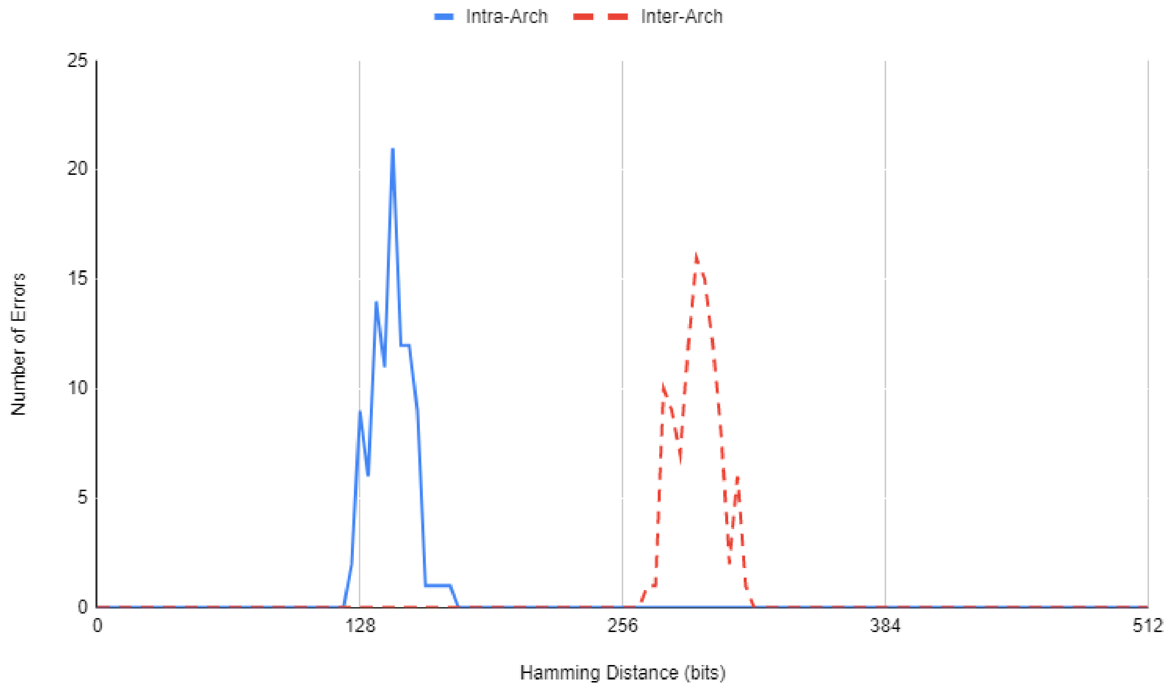


Figure 14 – Hamming distance distribution for PUF responses with 512-bit challenges in different devices.



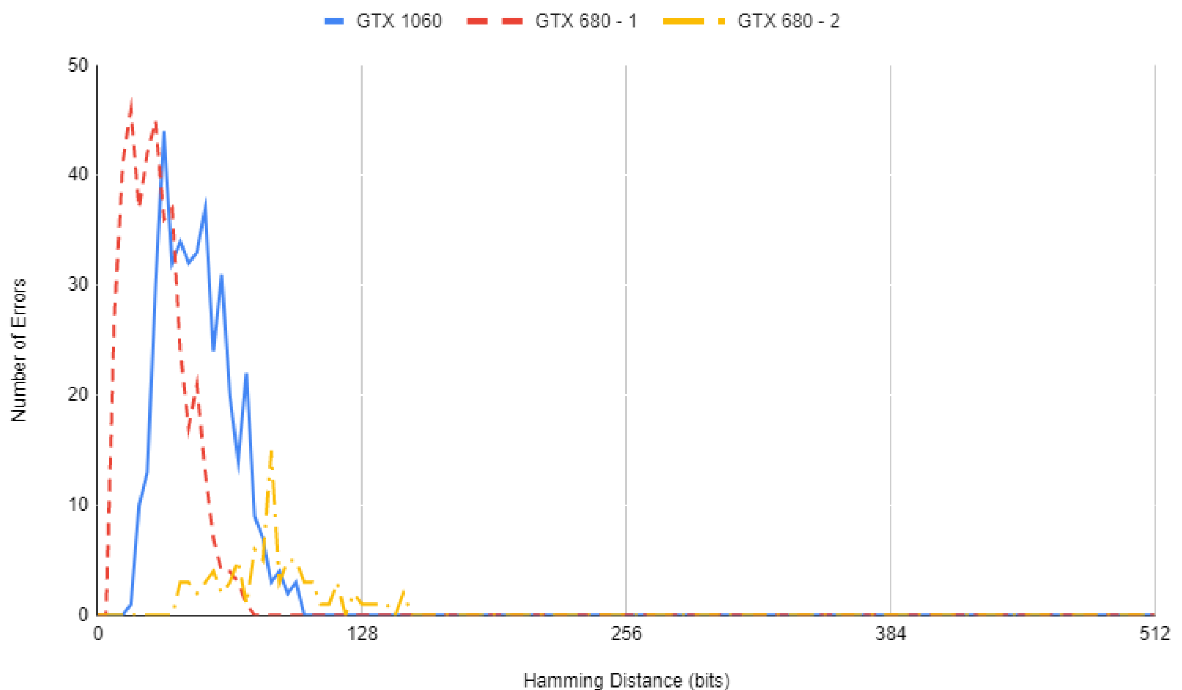
The plot in Figure 14 shows clearly a difference in the mean distribution for the different analysis. A completely indistinguishable distribution would lie in the 256 bit mark. This means that the distance from the responses of different architectures are noticeably different, enough that an attacker could potentially discover the evaluated architecture. However, the differences from responses of the same architecture are not as distant as

would be desired. Given a low enough reliability for a certain CRP, it could cause a mistake in identification.

5.4.2 Reliability

Reliability required the collection of multiple partial error maps for each device. Specifically, 9 partial error maps per device, this resulted in 45 unique combinations that were evaluated using 10 random challenges of 512 bits. This in turn resulted in 450 unique samples to be evaluated using Equation (9). The distribution of the samples for each board can be seen in Figure 15.

Figure 15 – Hamming distance distribution for PUF responses with 512-bit challenges in the same device.



The results presented in Figure 15 show good results for reliability in two of the boards analyzed. The curve representing GTX 680 - 2 presented the worst results, potentially due to a smaller concentration of errors on certain positions as shown in Table 2. Even when superposing the results with Figure 14 the results show premise, for the reliability and uniqueness curves do not superpose each other, which would mean they are indistinguishable. The results for different key-sizes can be seen in Table 3.

Table 3 – Reliability for each device and key size.

Device	GTX 1060				GTX 680 - 1				GTX 680 - 2			
Key Size	512	256	128	64	512	256	128	64	512	256	128	64
Reliability (%)	91.37	91.34	91.60	91.32	94.73	94.65	95.52	96.04	84.17	82.77	81.55	82.29

Table 3 presents similar results as Figure 15. There is some variation with key-sizes, but the greater variation is with the board used.

5.4.3 Uniformity

To evaluate uniformity, each full error map was filtered with the highest threshold that allowed for the maximum spread of errors. This will vary for each GPU, and in this experiments, the values were 50%, 5% and 15% for GTX 1060, GTX 680 - 1 and GTX 680 - 2, respectively. A hundred random 512 bit challenges were sent and the responses collected. The average of the Hamming Weights for each response was calculated using Equation (10). Thus, the uniformity for each GPU is presented in Table 4.

Table 4 – Uniformity for each device using 100 response samples each.

Device	GTX 1060	GTX 680 - 1	GTX 680 - 2
Uniformity (%)	56.03	49.96	60.84

5.5 Discussions

Investigating the heat maps of Figures 6 and 7 it is clear that each GPU presents a slightly different behaviour for each set of parameters. This justifies the choice of including the operational parameters in the key-update protocol, as changing the parameters will change the amount of errors generated. From these maps it is also possible to see that the combination of high power settings and overclock yields a significantly higher amount of errors, although this is interesting for the G-PUF operation, it might compromise the boards life-time, by elevating the temperature and voltage. A balance must be struck that takes in to consideration error generation and the temperature.

The results presented by evaluating the thresholds in Table 2 vary greatly for the two architectures. Pascal has a more condensed error pattern, which facilitates that multiple errors occur in the same position. Kepler, on the other hand, spreads the errors across all the error map, which allows only a few positions to actually accumulate enough errors. This means that Pascal boards can have a smaller collection phase and still have reliable results, while Kepler must collect many more samples to guarantee accumulation.

The evaluation of the PUF metrics shows promising results. Uniqueness and reliability can be evaluated by plotting the distribution of hamming distances between different sample sets. Inter-arch uniqueness presented a mean distance of more than 50%, meaning it is possible to distinguish architectures based on the responses. The inter-arch distribution results were not so promising, it is clear that the concentration of errors in the left side of the map as show in Figures 12 and 13 made distinction between the devices harder. Reliability results vary for device and architecture, both GTX 1060 and GTX 680

- 1 presented good results, considering the G-PUF doesn't use any error correction code as required by many other PUFs. The GTX 680 - 2 presented the worst results for all the tests, specially uniformity and reliability, this is believed to be the result of less samples being analyzed for this board.

Although the collection of samples and generation of CRPs took a relatively long time in the experiments (hours and minutes respectively), by being a software-only solution, G-PUF is susceptible to optimizations. The code used was not designed to be a benchmark, and instead was created for reliability and contained several debugging functions. Preliminary versions of an optimized code allowed an speed-up of up to 100 times in the collection phase, collecting one million samples in a few minutes. CRP extraction could also be optimized by using different algorithms or even transferring the calculations of CRPs from the Host to de GPU.

6 Conclusion

The main objective of developing a functional PUF in a GPU using only software was achieved. G-PUF allows for multiple CRPs to be generated and used to distinguish devices while allowing for deterministic responses. Although the uniformity presented a noticeable bias, and reliability result were not consistent, G-PUF presented good results for larger key-sizes with up to 94.73% reliability. These parameters had a direct relation with the number of error samples collected for each PUF, Pascal cards needed fewer error samples to present a more representative error map due to the clustered nature of the errors. G-PUF demonstrated better results in newer architectures, which are more resilient to errors and present further optimizations. This is favorable, since it allows for its use in state-of-the-art GPUs, where other GPU PUFs could not function properly. While not an all encompassing solution, G-PUF serves as a stepping stone for further development of intrinsic PUFs in GPUs, it shows how physical characteristics can be extracted from the device without specialized hardware monitors or functionalities.

As future works, there are a number of improvements that could be made to the program, they include: automation of temperature characterization and operational point determination, improvement of CRP extraction program to execute faster (current bottleneck of the system), and lower memory footprint in the system. Other analyses could also be made to test the effects of different types of matrices and evaluate how the error pattern changes for other types like integer or half-precision. Tests could also be made with different initialization values, instead of random numbers a selected pattern, that could reveal new error patterns due to optimizations in the cache lines. Overall, the tests themselves need to be improved as to encompass other GPU architectures, test more boards per architecture and also adapt the program to other brands of GPUs. There are also other venues left to be explored, like the graphical processors of the GPUs which also present soft errors, and the tensor and RT cores of newer GPU architectures.

Bibliography

BACHA, A.; TEODORESCU, R. Authenticache: Harnessing cache ecc for system authentication. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2015. p. 128–140.

BAKHTIARI, S.; SAFAVI-NAINI, R.; PIEPRZYK, J. *Cryptographic Hash Functions: A Survey*. [S.l.], 1995.

BHATTACHARJEE, K.; MAITY, K.; DAS, S. A search for good pseudo-random number generators : Survey and empirical studies. *CoRR*, abs/1811.04035, 2018. Disponível em: <http://arxiv.org/abs/1811.04035>.

CHAUVET, J.-M.; MAHÉ, E. Secrets from the GPU. may 2013. Disponível em: <http://arxiv.org/abs/1305.3699>.

CHEN, T.; LI, M.; LI, Y.; LIN, M.; WANG, N.; WANG, M.; XIAO, T.; XU, B.; ZHANG, C.; ZHANG, Z. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. dec 2015. Disponível em: <http://arxiv.org/abs/1512.01274>.

COELHO, I. M.; COELHO, V. N.; LUZ, E. J. d. S.; OCHI, L. S.; GUIMARÃES, F. G.; RIOS, E. A GPU deep learning metaheuristic based model for time series forecasting. *Applied Energy*, Elsevier, v. 201, p. 412–418, sep 2017. ISSN 0306-2619. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0306261917300041>.

CUI, H.; ZHANG, H.; GANGER, G. R.; GIBBONS, P. B.; XING, E. P. GeePS. In: *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*. New York, New York, USA: ACM Press, 2016. p. 1–16. ISBN 9781450342407. Disponível em: <http://dl.acm.org/citation.cfm?doid=2901318.2901323>.

DAEMEN, J.; RIJMEN, V. *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN 3540425802.

DUANE, M. Y. *CUDA for Engineers*: An introduction to high-performance parallel computing. [S.l.]: Addison-Wesley, 2015.

GONZALEZ, R.; GORDON, B. M.; HOROWITZ, M. A. Supply and threshold voltage scaling for low power cmos. *IEEE Journal of Solid-State Circuits*, v. 32, n. 8, p. 1210–1216, Aug 1997.

GRAYBEAL, S. N.; MCFATE, P. B. Getting out of the starting block. *Scientific American*, Scientific American, a division of Nature America, Inc., v. 261, n. 6, p. 61–67, 1989. ISSN 00368733, 19467087. Disponível em: <http://www.jstor.org/stable/24987511>.

GUAJARDO, J.; KUMAR, S. S.; SCHRIJEN, G.-J.; TUYLS, P. Fpga intrinsic pufs and their use for ip protection. In: PAILLIER, P.; VERBAUWHEDE, I. (Ed.). *Cryptographic Hardware and Embedded Systems - CHES 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 63–80. ISBN 978-3-540-74735-2.

GUO, X.; TANG, B.; TAO, J.; HUANG, Z.; DU, Z. Large Scale GPU Accelerated PPMLR-MHD Simulations for Space Weather Forecast. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016. p. 576–581. ISBN 978-1-5090-2453-7. Disponível em: <http://ieeexplore.ieee.org/document/7515738/>.

HALAK, B. *Physically unclonable functions: From basic design principles to advanced hardware security applications*. [S.l.]: Springer International Publishing, 2018. 1–250 p. ISBN 9783319768045.

HERDER, C.; Yu, M.; Koushanfar, F.; Devadas, S. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, v. 102, n. 8, p. 1126–1141, Aug 2014. ISSN 1558-2256.

HINEK, M. J. *Cryptanalysis of RSA and Its Variants*. 1st. ed. [S.l.]: Chapman & Hall/CRC, 2009. ISBN 1420075187, 9781420075182.

IYENGAR, A.; RAMCLAM, K.; GHOSH, S. Dwm-puf: A low-overhead, memory-based security primitive. In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. [S.l.: s.n.], 2014. p. 154–159.

IYER, S. G.; Dipakumar Pawar, A. GPU and CPU Accelerated Mining of Cryptocurrencies and their Financial Analysis. In: *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on*. IEEE, 2018. p. 599–604. ISBN 978-1-5386-1442-6. Disponível em: <https://ieeexplore.ieee.org/document/8653733/>.

KHAIRY, M.; WASSAL, A. G.; ZAHRAN, M. A survey of architectural approaches for improving gpgpu performance, programmability and heterogeneity. *Journal of Parallel and Distributed Computing*, v. 127, p. 65 – 88, 2019. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731518308669>.

KIM, J. S.; Patel, M.; Hassan, H.; Mutlu, O. 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)*. [S.l.: s.n.], 2018. p. 194–207.

KODYTEK; Filip; Lórencz; Róbert; Buček; Jiří. Improved ring oscillator puf on fpga and its properties. *Microprocessors and Microsystems*, v. 47, p. 55–63, 2016.

LANGE, T. *PUFFIN INFSO-ICT-284833*. 2012. <http://www.puffin.eu.org/>. Accessed: 12/09/2019.

LENG, J.; Buyuktosunoglu, A.; Bertran, R.; Bose, P.; Reddi, V. J. Safe limits on voltage reduction efficiency in gpus: A direct measurement approach. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. [S.l.: s.n.], 2015. p. 294–307.

LI, F.; FU, X.; LUO, B. A hardware fingerprint using gpu core frequency variations (poster). In: . [S.l.: s.n.], 2015. p. 1650–1652.

LI, F.; FU, X.; LUO, B. POSTER. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. New York, New York, USA: ACM Press, 2015. p. 1650–1652. ISBN 9781450338325. Disponível em: <http://dl.acm.org/citation.cfm?doid=2810103.2810105>.

MACHIDA, T.; YAMAMOTO, D.; IWAMOTO, M.; SAKIYAMA, K. A new arbiter puf for enhancing unpredictability on fpga. *The Scientific World Journal*, v. 2015, p. 1–13, 2015.

NITHYANAND, R.; SION, R. Solving the software protection problem with intrinsic personal physical unclonable functions. 2011.

NOBILE, M. S.; CAZZANIGA, P.; TANGHERLONI, A.; BESOZZI, D. Graphics processing units in bioinformatics, computational biology and systems biology. *Briefings in Bioinformatics*, Narnia, v. 18, n. 5, p. bbw058, jul 2016. ISSN 1467-5463. Disponível em: <https://academic.oup.com/bib/article-lookup/doi/10.1093/bib/bbw058>.

NVIDIA. *Fermi Whitepaper*. 2006. <https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Accessed: 12/09/2019.

NVIDIA. *Kepler Whitepaper*. 2012. <https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf>. Accessed: 12/09/2019.

NVIDIA. *Pascal Whitepaper*. 2016. <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>. Accessed: 12/09/2019.

NVIDIA. *cuBLAS cuBLAS Toolkit Documentation*. 2019. <https://docs.nvidia.com/cuda/cublas/index.html>.

NVIDIA. *CUDA ToolkitToolkit Documentation*. 2019. <https://docs.nvidia.com/cuda/>. Accessed: 12/09/2019.

NVIDIA. *cuTLASS cuTLASS: Fast Linear Algebra*. 2019. <https://github.com/NVIDIA/cutlass>.

NVIDIA. *GPU Boost*. 2019. <https://www.geforce.com/hardware/technology/gpu-boost/technology>. Accessed: 2019-11-29.

NVIDIA. *NVIDIA Management Library*. 2019. <https://developer.nvidia.com/nvidia-management-library-nvml>.

SCHALKWIJK, J.; JONKER, H. J. J.; SIEBESMA, A. P.; Van Meijgaard. Weather Forecasting Using GPU-Based Large-Eddy Simulations. *Bulletin of the American Meteorological Society*, v. 96, n. 5, p. 715–723, may 2015. ISSN 0003-0007. Disponível em: <http://journals.ametsoc.org/doi/10.1175/BAMS-D-14-00114.1>.

SEMERARO, G.; Magklis, G.; Balasubramonian, R.; Albonesi, D. H.; Dwarkadas, S.; Scott, M. L. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In: *Proceedings Eighth International Symposium on High Performance Computer Architecture*. [S.l.: s.n.], 2002. p. 29–40.

SHI, S.; Wang, Q.; Xu, P.; Chu, X. Benchmarking state-of-the-art deep learning software tools. In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. [S.l.: s.n.], 2016. p. 99–104. ISSN null.

SUTTER, H. *A much briefer version under the title*. [S.l.], 2005. v. 30, n. 3. Disponível em: <http://www.gotw.ca/publications/concurrency-ddj.htm>.

TAYLOR-WEINER, A.; AGUET, F.; HARADHVALA, N.; GOSAI, S.; ANAND, S.; KIM, J.; ARDLIE, K.; ALLEN, E. V.; GETZ, G. Scaling computational genomics to millions of individuals with GPUs. *bioRxiv*, Cold Spring Harbor Laboratory, p. 470138, feb 2019. Disponível em: <https://www.biorxiv.org/content/10.1101/470138v3.abstract>.

TEHRANIPOOR, F.; Karimian, N.; Yan, W.; Chandy, J. A. Dram-based intrinsic physically unclonable functions for system-level security and authentication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 25, n. 3, p. 1085–1097, March 2017.

TOSHIBA. *MOSFET Gate Drive Circuit*. 2018. <https://toshiba.semicon-storage.com/la-pt/top.html>. Accessed: 12/09/2019.

VAN AUBEL, P.; BERNSTEIN, D. J.; NIEDERHAGEN, R. Investigating SRAM PUFs in large CPUs and GPUs. jul 2015. Disponível em: <http://arxiv.org/abs/1507.08514>.