Aalto University

School of Science

Master's Programme in Information Networks

Johannes Vainio

Physically Unclonable Functions as Trust Anchors for Connected Embedded Device Security

Master's Thesis

Espoo, May 22, 2018

Supervisor: N. Asokan, Professor

Thesis advisor(s): Antti Kettunen, Ph.D. (Physics)

AALTO UNIVERSITY
School of Science
Master's Programme in Information Networks

ABSTRACT OF THE
MASTER´S THESIS

Author: Johannes Vainio

Title of the thesis: Physically Unclonable Functions as Trust Anchors for Connected Embedded Device Security

Number of pages: 48+15

Date: May 22, 2018

Major: Information Networks

Supervisor: N. Asokan

Thesis advisor: Antti Kettunen, Ph.D. (Physics)

Physically Unclonable Functions (PUF) carry promise for solving some of the current problems in current embedded device security. Devices are often deployed in unmonitored areas accessed over the untrusted public internet. Confidential communications require secrets on the device, but simple persistent memory can easily be read during transit from a factory to the deployment by untrusted personnel.

A PUF can be thought of as a mechanism for extracting deterministic randomness from something that cannot be copied or cloned. In this thesis, we design, implement and evaluate an FPGA PUF that can produce an estimated 250 bits of physically unclonable, reliable entropy from measuring slight variations in ring oscillator frequencies deployed on the FPGA fabric. The frequency variations result from submicroscopic silicon chip manufacturing variations that lie below manufacturing tolerances and cannot thus be copied.

The PUF can be used as a device trust anchor, providing a stable basis for various cryptosystems, in turn making it possible to authenticate devices and initiate encrypted channels without any "leaps of faith" in trusting unknown key fingerprints at first sight.

Keywords: Physically Unclonable Function, PUF, RO-PUF, trust anchor, embedded device security, IoT security, FPGA, SoC, device attestation, device authentication, challenge-response protocol

Publishing language: English

Acknowledgements

Greatest thanks for this thesis go to my thesis supervisor Antti Kettunen for his time and mental energy in keeping up the constructive criticism as I kept writing about distance measures, positional errors and other tedious PUF details. I would like to thank Juhani Mäkelä for preparing and suggesting this subject. As my lack of background in this kind of “real” engineering made me suspect my chances with this topic, I am grateful for the trust that was placed in me. Thanks belong also to Nixu Corporation for allowing me to use company time and equipment for this thesis.

Acronyms

ASIC Application-Specific Integrated Circuit
CLB Configurable Logic Block
CPU Central Processing Unit
CRP Challenge-Response Pair
FPGA Field-Programmable Gate Array
LISA Longest Increasing Subsequence Algorithm
LUT Look-Up Table
MUX (De)Multiplexer
PL Programmable Logic
PRNG Pseudo-Random Number Generator
PS Processing System
PUF Physically Unclonable Function
RO Ring Oscillator
SoC System on a Chip
RRAM Resistive Random Access Memory
SRAM Static Random Access Memory
TRNG True Random Number Generator

Glossary

Basic Element FPGA primitive; most commonly a LUT, a Register or a MUX.
Bitstream FPGA configuration data
Challenge The input of a PUF evaluation; see CRP
CLB Modular FPGA component. Contains two Slices and one routing matrix.
CRP A PUF input and its associated expected output.
FPGA An integrated circuit containing configurable logic components.
Inverter A NOT logic gate; inverts its input
LISA A sequencing algorithm presented by Yin & Qu (2010).
LUT A Basic Element that can be configured to act as any logic gate.
MUX A multiplexer has a signal input, selector input and many alternative outputs. The value of the selector input decides to which output the signal input is routed to. Demultiplexers have many inputs and one output.
PL Name for an FPGA on an embedded device
PS Name for a CPU on an embedded device
Register on-chip memory; single-bit registers are also Basic Elements.
Response The output of a a PUF evaluation; see CRP
RO A combination of logic gates that outputs an oscillating signal
Routing matrix FPGA component controlling the connections between Basic Element pins.
Slice A modular FPGA component. Contains a small number of Basic Elements.
SoC A name for a chip that integrates several components, allowing it to act as a complete system.

1 Introduction

The purpose of this Master's thesis is to determine the feasibility of implementing a practically viable Physically Unclonable Function (PUF) in a field-programmable gate array (FPGA), by attempting to do so. Our use case is to use the PUF as a trust anchor, the source of trust for cryptographic operations performed on the device. The requirements for PUF performance will be defined in more detail in relation to our use case.

PUFs are based on exploiting physical random variations that are often intrinsic to the device. A silicon PUF is a circuit designed for amplifying these unpredictable sub-microscopic manufacturing variations on a silicon chip. Our PUF design uses frequency differences of ring oscillators (ROs), caused by this variation in FPGA silicon chips, to calculate physically unclonable output bits.

Our target device is the Avnet MiniZed board (Avnet n.d.), which carries a Zynq7000 system-on-a-chip (SoC). The Zynq has, among other things, both a programmable logic (PL) block containing the FPGA and a processing system (PS) containing a normal microprocessor (CPU), making it possible to build complicated systems without leaving the chip. Our PUF was deployed on five MiniZed boards in order to evaluate our PUF’s behavior on different chips. Analysis has been performed to evaluate the quality of our PUF using measures presented in academic literature.

Given an input challenge, a PUF calculates a deterministic but hard-to-guess response. In other words, the PUF’s behavior should be reliable but unpredictable. It should not be possible to predict the behavior of the same function on any other device, and it should not even be possible to manufacture such a device, due to the variations existing on a scale below manufacturing tolerances. The PUF will use challenge-response pairs (CRPs) generating responses with at least enough combined entropy to be suitable for use as unique device fingerprint or trust anchor.

2 Background

What is a PUF?

A PUF is a *function* that turns an input (called a *challenge*) into an output (called a *response*), usually with some uncertainty. A PUF extracts randomness from something physical: in our case, this source will be sub-microscopic manufacturing variations in an FPGA silicon chip. The extracted randomness can be measured using the concept of entropy (Maes & Verbauwhede 2010). A PUF can also be thought of as a “device fingerprint”, as it can be used to uniquely identify a device. The concept of PUF was first introduced by Pappu et al. (2002) under the name *physical one-way function*.

A *trust anchor* is an entity in a cryptographic system that is assumed to be trusted, rather than have its trust derived from some other entity (IETF 2016). In the context of public-key cryptography, a trust anchor is defined as “[...] an authoritative entity represented by a public key and associated data” (IETF 2010). In the context of trusted platform modules, where a co-processor continuously records the software state of a system, the first measurement during a boot uses a trust anchor, that often is the system firmware (Abera et al. 2016). Trusted platform modules in turn often act as trust anchors for larger systems (Schmidt et al. 2008). PUFs are well suited for use as trust anchors due to being unclonable, unpredictable and hardware-based.

The physical variations that PUFs are based on are often made visible by slight but deterministic differences in objects that have been designed to be identical. For example, on integrated circuits the small variations caused by manufacturing processes cause behavioral effects such as different signal speeds or varying memory cell settling behavior across design-identical components. Different PUF designs extract the variations from physical objects in different ways. Our PUF will be based on differences between the speeds of design-identical ring oscillators placed on an FPGA.

The concept of a challenge-response pair (CRP) is central in the theory of PUFs. A PUF can support a varying number of challenge-response pairs. The challenge and response can be single bits, or arbitrarily long. A PUF should always produce as close to the same output as possible on the same device, and an unpredictably different output on any other device.

The CRP behavior of a PUF can be formulated as (Maes & Verbauwhede 2010):
`PUF(Challenge) -> Response + random_variation`
Due to the possibility of bit flips, a random variable should be added to the equation to depict the fact that bits in the response may get flipped from the expected value. The challenge can affect the response generation in various ways, depending on the design of the PUF. In our case, the challenge will consist of two values used to select ring oscillators.

A PUF that is based on the physical properties of something is called an intrinsic PUF, and PUFs based on chip variations are called silicon PUFs. While there are PUFs that are not based on silicon chip variations, including optical PUFs, coating PUFs and even “paper PUFs” , we will limit our scope to intrinsic silicon PUFs (Maes & Verbauwhede 2010).

Enrolling procedure

In order to know what the correct response to a challenge is, the PUF should be enrolled. In the enrolling procedure, all the CRPs that are planned to be used when the device is deployed should be evaluated and stored. Depending on the method used, unreliable bits might be filtered out in this phase by simply excluding them from being used as challenges. After enrolling all CRPs we want to use, a PUF can be used for for simple authentication of deployed devices as is shown in figure 1.

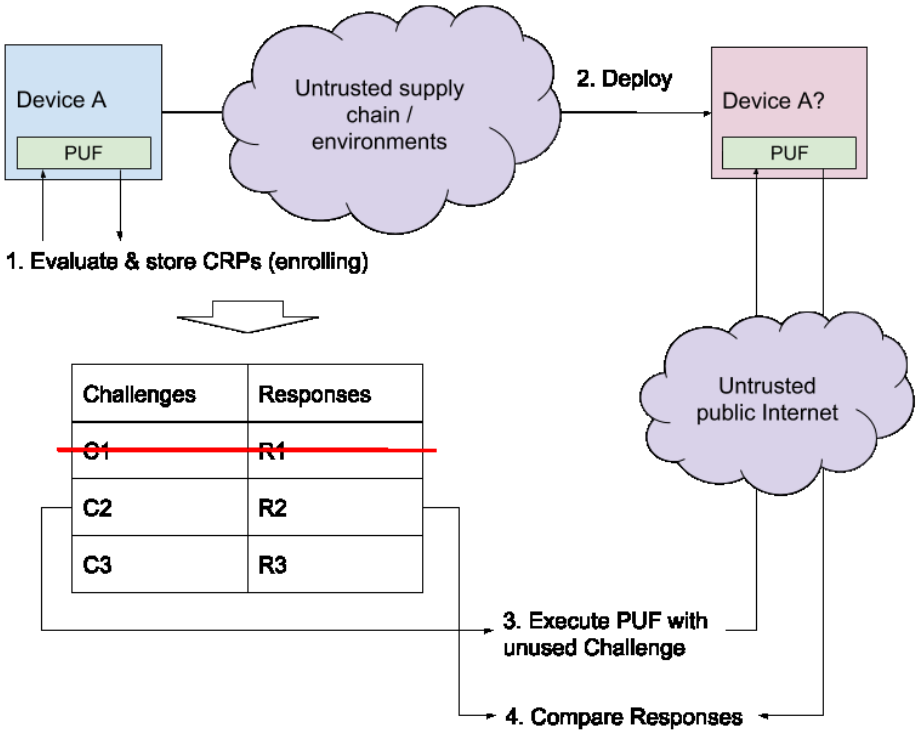


Figure 1: *An example of a simple PUF-based device authentication scheme with single-use CRPs (Suh & Devadas 2007).*

Properties of a PUF

According to Maes and Verbauwhede (2010), an ideal PUF has seven properties presented in table 1. On the other hand, Zhang et al. (2014) recognize only three more coarse-grained properties: *Persistent and Unpredictable*, *Unclonable* and *Tamper Evident*. The first covers Maes and Verbauwhede's properties *3* and *5*. *Unclonable* and *Tamper Evident* are exact matches. Zhang et al. do not include *One-way* in their properties, while property 1 is assumed given. Their view on property number 2 is unclear.

Of these seven properties, tamper evidence is particularly hard to achieve on general-purpose FPGAs, as current physical attacks (Lohrke et al. 2016, Zhang et al. 2014) can read and measure FPGA elements without physical contact. Here unclonability means unclonability in the physical sense, but attackers can bypass physical cloning by either characterization or modeling attacks, which we discuss later.

	Property	Explanation
1	*Evaluable*	A PUF should be able to produce responses to challenges.

2	*Unique*	It should be always possible to identify a PUF instance by its CRP behavior.
3	*Reproducible*	The same challenge on the same PUF should always produce the same, or very similar, response.
4	*Unclonable*	It should be very hard to move or copy the function from its physical environment. On silicon PUFs, this is achieved by using small, uncontrollable chip irregularities.
5	*Unpredictable*	It should not be possible to predict the response to a CRP from observing other CRPs on the same PUF or some other instance of the PUF.
6	*One-way*	It is not possible to reproduce the original challenge from an observed response.
7	*Tamper evident*	(Physical) attack attempts should leave traces, possibly altering the behavior of the PUF.

Table 1: Properties of an ideal PUF

Benefits of using a PUF

The motivation to use a PUF is usually device authentication or key material generation, to be used for either protecting data or intellectual property. Since the source of entropy is the physical chip, a PUF can be used both as-is to uniquely identify a chip, and also as a part of a cryptographic system to provide device-specific secrets. Key material from a PUF can be used in a wide variety of cryptographic applications, including symmetric and public-key cryptography. Combining these mechanisms can allow for ensuring both the confidentiality and integrity of the device contents and communications, and integrity of its identity.

Physical security is a challenge for embedded devices. While the complex data centers where much of today's cloud computing happens are usually under strict physical security, an embedded device might be deployed in any vehicle, building or public space, where access to it might not be controlled, or controlled by a third party.

For the use case of establishing a secret on a deployed embedded device, using a PUF has a number of benefits as compared to either using a stored secret, a true random number generator or a pseudo-random number generator.

Benefits over a stored secret
A secret or key held in non-volatile memory can potentially be read either programmatically or physically. In case a remote software-based attack can be developed, for example by exploiting a software vulnerability, it is often easily scalable to all devices of the same type. Supplier security should also be taken into account, as malicious contractors or suppliers can read hard-coded device serial numbers or keys stored in flash memory during transit and deployment. Specialized physically protected chips might be explored as a solution.

Benefits over a TRNG

It is possible to implement a True Random Number Generator (TRNG) in an embedded device, using for example analog noise measurements from a temperature sensor. This true randomness could be used to generate unpredictable keys. However, unless we compute and store the key on the device before deployment, we have no way of knowing what key the deployed device will have. Furthermore, there is no way of generating the same key twice, so the generated key would have to be stored.

When we don't know what key the device has, our only option is to trust the connection claiming to be the device on the first use (Trust On First Use; TOFU). After a TOFU connection has been established, it is secure, but if an attacker intercepts or spoofs the first request, all future traffic can be compromised. Using a PUF allows us to know beforehand what the key, trust anchor or identifier will be, thus allowing us to establish a trusted channel from the start.

Benefits over a PRNG

A Pseudo-Random Number Generator (PRNG) produces random-looking but predictable numbers given a seed value (Maes et al. 2012). A PRNG takes a seed value as input and outputs statistically random but deterministic output. Using PUFs on a set of devices can be thought to bear some resemblance to using PRNGs with a unique seed value on each device. However, the usefulness of PRNG-backed cryptography is severely restricted by the need of keeping the seed secret, thus providing little benefit over a stored key (Maes et al. 2012). A PUF does not require us to rely on such a brittle secret, as the randomness is embedded in the chip itself.

As such, using a PUF for cryptography has benefits in that (Maes et al. 2012):

1. We do not need to store sensitive keys in memory, as the PUF can regenerate the secret at any time.
2. The PUF is bound to a single physical chip, so it cannot be stolen except by full characterization or modeling of the PUF.

Attacking a PUF

A good PUF should remarkably improve the situation compared to any of the methods discussed above. While a PUF is very hard or impossible to clone, attacking a PUF is possible by other means. Attacks against PUFs include modeling attacks and characterization attacks (Rührmair et al. 2013; Maes & Verbauwhede 2010).

Modeling attacks are based on building a software model of the behavior of the PUF, with the help of tools like machine learning. Performing this kind of attack requires visibility into unobfuscated CRPs, but will compromise also other CRPs before they have been used.

Characterization attacks are based on collecting CRP behavior or physical measurements that can be directly used in attacking later CRPs. Characterization does not lead to the ability to predict unseen CRPs, but being able to run and store all CRPs might lead to the ability to masquerade as the PUF, even if the CRPs are obfuscated.

Zhang et al. (2014) separate non-modeling attacks further into *side channel* attacks and *physical cloning* attacks. They consider the SRAM attack by Helfmeier et al. (2013) the only

successful physical cloning up to that date, while other methods based on measuring, timing or power manipulation they consider side-channel attacks.

Common PUF designs

Arbiter PUF

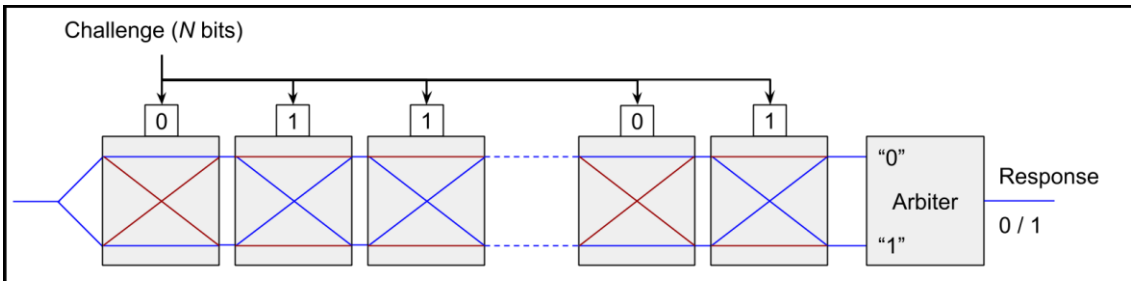


Figure 2: *An Arbiter PUF. Blue lines represent signal routing. Challenge bits determine the routing within each element.*

The first delay-based silicon PUF design is the Arbiter PUF proposed by Lim et al (2005). It is based on configurable delay lines, producing 1 if the line "1" was faster and 0 otherwise. Figure 2 demonstrates the signal routing behavior on a block level. Cross-coupled NAND gates are often used in the arbiter block to determine the faster signal.

The main weakness of the Arbiter PUF is its susceptibility to model-building attacks using machine learning. Due to the simple design's vulnerability more complex combinations of delay lines and logic gates have been proposed. These more complex combinations are more resilient but even these have been shown to be learnable (Rührmair et al. 2013).

The large sensitivity of the Arbiter PUF to routing differences makes it unsuitable for our target FPGA device. Arbiters require additional configurable delay lines to calibrate them correctly on an FPGA; on an ASIC, accurate controlling of the route design length is possible, so that any variance is caused by random process variation.

Butterfly PUF

The Butterfly PUF first proposed by Kumar et al. (2008) is based on two cross-coupled latches or flip-flops. Figure 2 presents a gate-level representation of the design. The excitement signal sets the other latch to logic-1 and resets the other to logic-0. Due to manufacturing variations, the state of both latches will settle on either value.

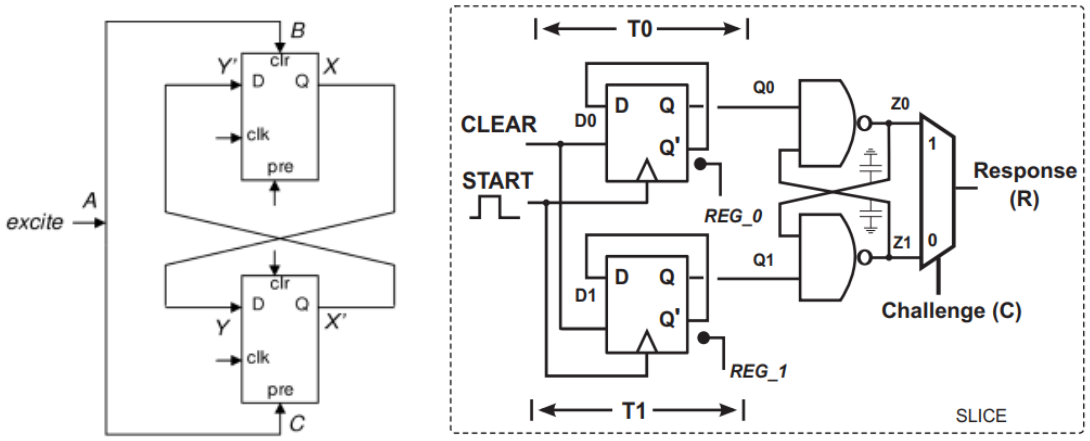


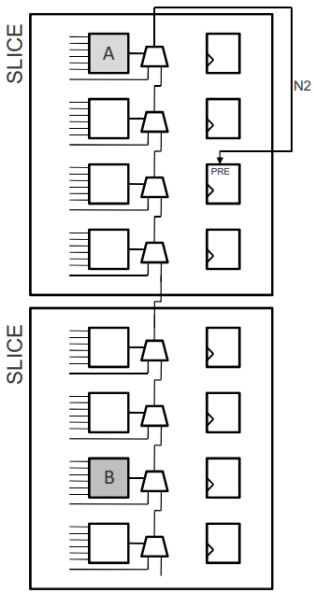
Figure 2 (left): the Butterfly PUF (Morozov et al. 2010)
Figure 3 (right): the Ultra-Compact Identification Generator (Chongyan & O'Neill 2015)

The Ultra-Compact Identification Generator

A design by Chongyan & O'Neill (2015) called the "ultra-compact identification generator" (figure 3) is also based on two flip-flops, the output of which is passed to two cross-coupled NAND gates. The NAND gate to first receive the signal from the flip-flop will determine the output of the bit unit. The proposing article places emphasis on the feasibility of FPGA deployment and compact footprint. Generating a single bit unit on an FPGA requires two registers and two *Look-Up Tables* (LUTs) configured as NAND gates. Selection logic can be added after the registers to make different pairings, allowing different challenges to be used.

SRAM PUF

Static Random-Access Memory (SRAM) PUFs are based on the settling behavior of SRAM memory bits (Kumar et al. 2008; Aysu et al. 2015; Herder et al. 2014). An SRAM cell (figure 5) is based on cross-coupled inverters that use positive feedback to maintain their state. Without a write operation, bits in SRAM tend to settle as either 0 or 1, and this tendency is determined by random process variations. SRAM PUFs are not viable on many modern devices, including the MiniZed board, due to the device resetting all memory to zero at power-up. SRAM PUFs have been demonstrated to be vulnerable to cloning (Helfmeier et al. 2013).



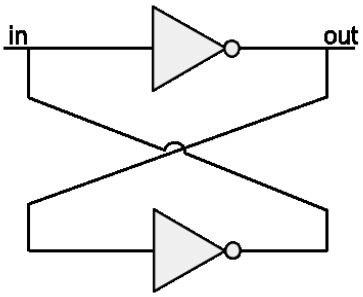


Figure 4 (left): *Anderson's glitch PUF (Anderson 2010)*
Figure 5 (right): *an SRAM cell*

Glitch PUF

Glitch PUFs are based on a signal spike (called a glitch), usually generated by signal speed variation caused by random process variation on a design-identical route. Anderson's (2010) FPGA-friendly PUF is an example of this concept. Many PUFs are very sensitive to routing asymmetries, and it is hard to guarantee symmetrical routing on FPGAs. To that end, this PUF is based on the binary arithmetic carry chains present in Xilinx FPGAs (see figure 4). Each of his PUF's bit units operate within two adjacent *Slices*, where there is a dedicated wire between the carry chains; the connection between them does not go through the routing matrix. The glitch is generated by the difference in the speeds of LUTs A and B when transitioning to the other logic state. Distance between the LUTs determines the length of the glitch pulse. Anderson saw the lack of cross-device portability of designs as a problem, and thus he openly provides the source code of his design, to help in the evaluation of PUF designs in literature.

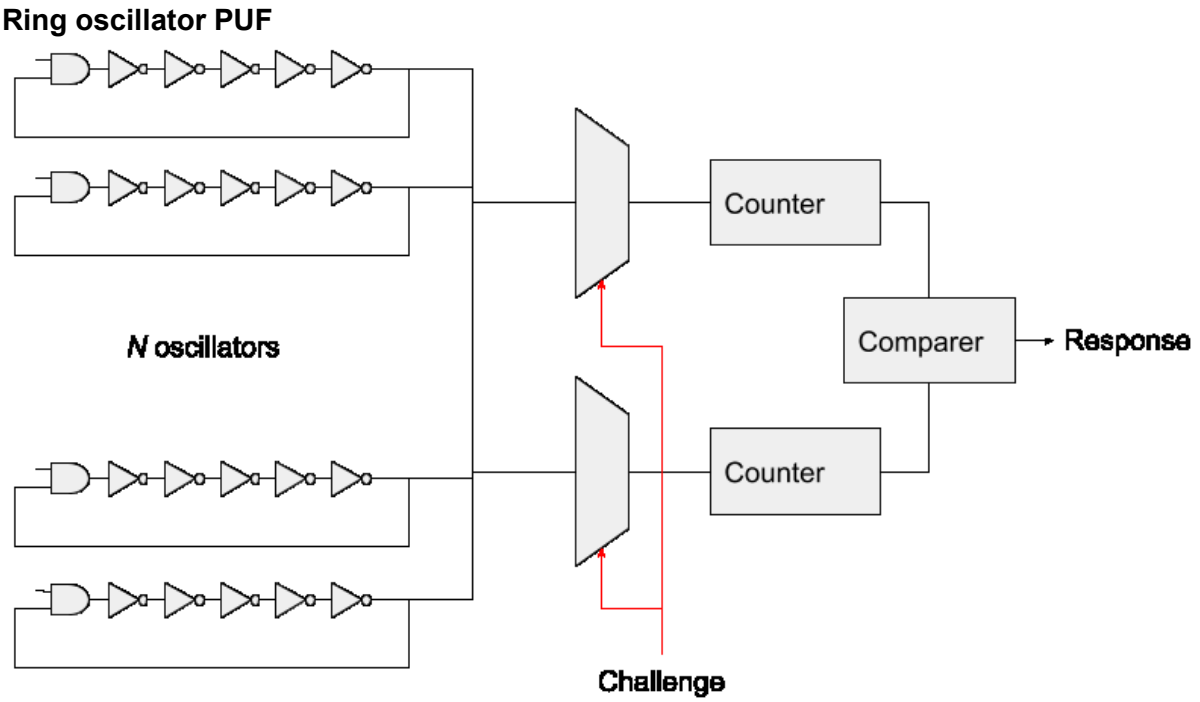


Figure 6: *A ring oscillator PUF (Suh & Devadas 2007).*

The ring oscillator PUF (RO-PUF) is based on the fact that random manufacturing variations have an unpredictable and measurable effect on the frequencies of ring oscillators, usually enough to overcome random noise and environmental effects. As the effect is random but deterministic, PUFs can be built using this behavior. The earliest version of a RO-PUF was proposed by Gassend et al (2002); in their concept, the challenge alters the signal routing (similarly to the Arbiter PUF) within each of a pair of ring oscillators, and oscillator pair frequencies were then compared to determine the faster oscillator and thus the output bit. As their design shares the additive delay characteristics with the Arbiter PUF, it is vulnerable to the same modeling attacks (Maes & Verbauwhede 2010).

The design that has since become the baseline RO-PUF design in the literature was proposed by Suh & Devadas (2007), shown in figure 6. Their design differs from Gassend et al. in that all oscillator routings are designed identical, and the challenge selects which RO units are to be compared. The RO frequencies are again compared with each other, and the output bit is again decided based on which oscillator was measured to be faster. *N* ring oscillators are deployed on the FPGA, each connected to a pair of demultiplexers. The challenge is used as the selector input for the demultiplexers, thus selecting which oscillators are connected with the edge counters. When the PUF is powered, the edge counters count the number of rising signal edges that are output by the two ring oscillators selected by the challenge. After either a specified oscillation count is reached or certain period of time has elapsed, the PUF is powered off and the counts are compared to determine the output bit. RO-PUFs have achieved good measurement results in the literature (Maiti & Schaumont 2010; Maes et al. 2012).

3 Problem definition

Use case

Our use case is to calculate an adequate number of unpredictable, unique, physically unclonable bits to build a device-specific secret that is suitable to use as a device trust anchor. This secret trust anchor allows us to setup confidential communications with the device while being certain of its identity. A PUF enrolling procedure will be performed in a trusted environment before deployment.

In the conceptual use case scenario for the purposes of this thesis, the device is to be deployed in the field in an environment that is relatively safe (e.g. a normal locked room) but not controlled by us (e.g. on someone else's premises). We are connected to the device over the public internet, which is untrusted. We will not have physical access to it most of the time, but we can arrange visits with advance notice to maintain or replace the device.

It is a fact of cybersecurity is that there can never be perfect protection against an attacker who has physical access, time and money. With a PUF, we can aim to make these kinds of attacks as hard to perform and scale as possible. It should be possible to make the attack require similar physical access to each new attacked device, possibly requiring removal from deployed site and leaving tamper evidence in the process.

Threat model

There are three assets we are trying to protect:

- Cryptographic identity of the device
- Confidentiality of device data and communications
- Cryptographic identities of other similar deployed devices

Furthermore, the threats our device is facing can be divided into three categories:

Network threats. Relevant network attacks include for example spoofing and man-in-the-middle (MITM) attacks. In a spoofing attack, someone might pretend to be our device in order to receive legitimate communications. In a MITM attack an attacker intercepts and possibly manipulates communication packages to and from the device.

Malicious software. An attacker might get software to run on the device using e.g. a vulnerability in 3rd party software on the device, or through physical access. Malicious software insertion should be mitigated using standard best practices in hardening and software updating. This attack can be quite cost-effective compared to a true physical attack.

Physical attacks. An attacker might steal a device, take it into a laboratory test bench, and start probing it with physical sensors. Physical attacks can e.g. read and manipulate secrets on the chip and board memory.

Requirements

R1: Adequate entropy with no fundamental weaknesses
The length of the bit string should be long and unpredictable enough to provide adequate entropy for robust cryptography. While 128 bits of entropy should be safe for the foreseeable future, 256 bits of entropy is guaranteed to be enough against any brute-force attack due to fundamental limitations arising from physics (Schneier 1999, 2013).

The design should not have any inherent remotely exploitable vulnerabilities or other fundamental weaknesses, such as being susceptible to modeling attacks, and should be as resistant to physical attack as is reasonably possible. The design concept's potential for further hardening should be included in this consideration.

R2: Maximum reliability and longevity within the expected operating environment
PUF reliability means the response from a PUF instance is always the same. We are aiming for 100% reliability, which means that the PUF evaluated on the same device should always return the same identical response, and never flip any of its bits under the expected operating conditions.

In our case, reliability of the response is prioritized over the number of usable bits, challenge-response pairs and even entropy. If each PUF instance returns exactly the correct response, we can avoid implementing error-correction methods. The PUF should remain usable for as long as possible, preferably 10 years, and give advance warning of aging deterioration.

The PUF should remain reliable under temperatures varying within the predicted operating temperature range. The effect of altered voltages on reliability will not be considered in our evaluation, as voltage is non-trivial to alter on our test device, and this effect not central to our use case.

R3: Reasonable efficiency
It is possible that some bits in the PUF output are predictable, and thus contribute less than a full bit of entropy to the response. A perfectly efficient PUF would have only response bits that produce close to one bit's worth of entropy each. We can estimate efficiency using *entropy density*, calculated as the ratio of produced bits and the entropy carried by them (Maes et al. 2012). It is possible to achieve an adequate amount of entropy (**R1**) using either a smaller number of high-quality bits or a larger number of lower-quality bits. We shall later estimate the entropy for an attacker with full knowledge of bit-specific biases.

R4: Scalability and portability of design
All parts of the system should be designed so that it is as widely applicable to different devices as possible, including near-future ones. The design should be easy to increase in size. The PUF should be as agnostic of underlying hardware specifics as possible, and it should be cheap and easy to deploy on new devices.

R5: Understanding the sources of error
We will research the sources and effects of error in the behavior of our PUF. While this is not strictly a requirement on the PUF implementation, since the implementation is unlikely to be

perfect, it is important to measure and understand the different sources of error that prevent our PUF from performing ideally. This builds trust in the implementation and lays the groundwork for future improvements on the PUF design.

Selected concept

The concept we have selected for our implementation is a controlled ring oscillator PUF with software-readable oscillation counter registers. Using software running on a CPU in conjunction with a PUF is known as a Controlled PUF or CPUF (Maes & Verbauwhede 2010). This “RO-CPUF” concept was selected for its following good qualities that help us achieve our requirements:

- Relative resilience against attacks (R1). Arbiter PUFs have been shown to be very vulnerable to model-building attacks due to the fact that every observed CRP evaluation reveals information that is also used in the generation of other CRPs (Rührmair et al. 2013, Maes 2013: 110). SRAM PUFs have been attacked using a Focused Ion Beam (FIB) and measuring leaking Near Infrared photonic emissions, leading to a complete characterization (Helfmeier et al. 2013). Meanwhile, Herder et al. (2014) report only two RO-PUF attacks, one manipulating the frequency of the oscillator by driving a signal on the ground plate, and another physically measuring the oscillations. The first is a non-issue for us because the attack would have to happen in the enrolling phase; the second is defeated by multiple oscillators running simultaneously in close proximity, which will happen in our design. More recently, however, RO-PUFs have been successfully attacked using Laser Voltage Probing (Lohrke et al. 2016), making it relevant to consider mitigations against direct frequency measurements.

- **Good measurement results in literature (R1, R3).** In literature, RO-PUFs have reported the best reliability and unpredictability measurement results when compared to implementations of the other basic designs (Maes 2013: 115, 125). Reliability measures have been good, which is a priority to us.

- **Modularity and scalability (R4).** On an FPGA we do not have precise control over the physical routing of signals in the chip, as we might have when designing an ASIC. For example, Arbiter PUFs are very sensitive to routing error, making them unsuitable for FPGAs. As each oscillator unit is modular and has an identical internal component layout, we can assume that the internal connections are also identical in design. Implementing a ring oscillator requires only an odd number of NOT gates, one AND gate and in our case one OR gate. These can easily be implemented using the configurable look-up tables (LUTs) that normally make up the bulk of elements on any FPGA. This is in contrast to e.g. the exotic memristor PUF (Mazady et al. 2015). The RO units are also reasonably compact, each occupying 7 of the 8 LUTs present in a single CLB. Increasing the number of oscillators is simply a matter of adding them and making the selection logic accommodate larger selectors.

- **On-chip controllability (R3, R2).** Our design can be controlled from the on-chip processor. We can choose from the software which RO units to run, in which order, and for how long. When we have an established a trusted channel, we can even remotely configure

the processor to use new challenges for the PUF, or run diagnostic runs to detect PUF aging. We can also potentially perform pre-processing and post-processing on the PS while remaining on the same chip, increasing the security performance.

- **Scalar output (R2, R5).** While for example a Glitch PUF or Butterfly PUF is based on simple units producing just 0 or 1, we have the benefit of collecting absolute measurements from our oscillators. Acquiring raw speed data about the oscillators instead of only the final output bits allows us to make our enrolling method robust and predictable with greater confidence and fewer measurements. Scalar data combined with controllability also adds the option of adjusting the run length, while containing more information about the underlying chip variation. However, while easing development and enrollment, on a deployed device this feature increases the attack surface, leading to considerations if the enrollment and deployment should use different FPGA configurations.

4 Methodology

Technical methods & design

Device

Our target device is the Avnet MiniZed containing a Xilinx Zynq-7000 SoC chip. The chip contains both a programmable logic (PL) and a processing system (PS) section. The PS contains a single ARM Cortex CPU. The board was chosen due to it containing a widespread, established and modern chip and its affordable pricing, making it possible to acquire several devices for the purposes of the project. The board has been designed as an evaluation board, containing a variety of features at a low price. As a major objective of this study is researching the feasibility of implementing a good PUF, it also makes sense to use the smallest device possible for developing the proof-of-concept solution.

Our PUF design

As mentioned, our design is based on the RO-PUF concept with a CPU-based controller program. The controller program has access to four 32-bit registers through an AXI-Lite interface. All the registers are readable and two of the registers are writable by the controller. Registers 1 and 2 are for input; the first ten bits (bits 0-9) of these registers are used as selection bits (RO address). Figure 7 provides a block level visualization of our design, also demonstrating the selection logic. The selection values control which two oscillators are powered up when the power bits are set, and they also control the routing from the same oscillators to the output counters.

The last bits (bit 31) of the input registers control the power to the PUF. They go through an AND gate before connecting to the signal pin on the input de-multiplexers (inmux1 and inmux2). The purpose of the AND gate is to ensure the oscillation does not start before both registers are written, and power can be turned off with only one register write. Figure 8 demonstrates a ring oscillator. We use an OR gate in front of each oscillator to accommodate two alternative sources for the power signal.

The input registers have 21 unused bits, and these can be used to expand the number of ROs that can be selected. Our design has 1000 ROs, and currently we could select among $2^{10} = 1024$ units. 2^{31} is so large that it does not impose limits on the practical oscillator count for the design. Registers 3 & 4 are connected to an edge-counting addition block. Each new rising edge of an oscillation arriving to either of the counters increments the connected register by one.

Thus our PUF takes two 10-bit addresses as input and produces two oscillation counts of up to 32 bits as output. However, if we consider the design at a higher level, including the CPU as a part of the PUF, and use the controlling program to perform simple frequency comparisons, the relationship between response and challenge bitstring lengths is

```
Challenge: (K * L * 2) bits
Response: K bits
Where  0 < K <= N/2 and L = ceil(log2(N))
```

The length of the challenge required is the length of a single address *L*, multiplied by the number of bits *K* we want to be in the response, multiplied by 2 because we need to select two oscillators. If we use simple pairwise comparison, *K* is bounded by *N*/2. *L* is the base-2 logarithm of *N* rounded up. For example, if we have 1000 oscillators, we need 10 address bits:

```
L = ceil(log2(1000)) = ceil(9.97) = 10
```

With 1000 oscillators, the formula results in a 20 bit-challenge for each response bit, meaning 10000 challenge bits are required for the maximum length response of 500 independent bits. The challenge might be procedurally generated to avoid storing it verbatim.

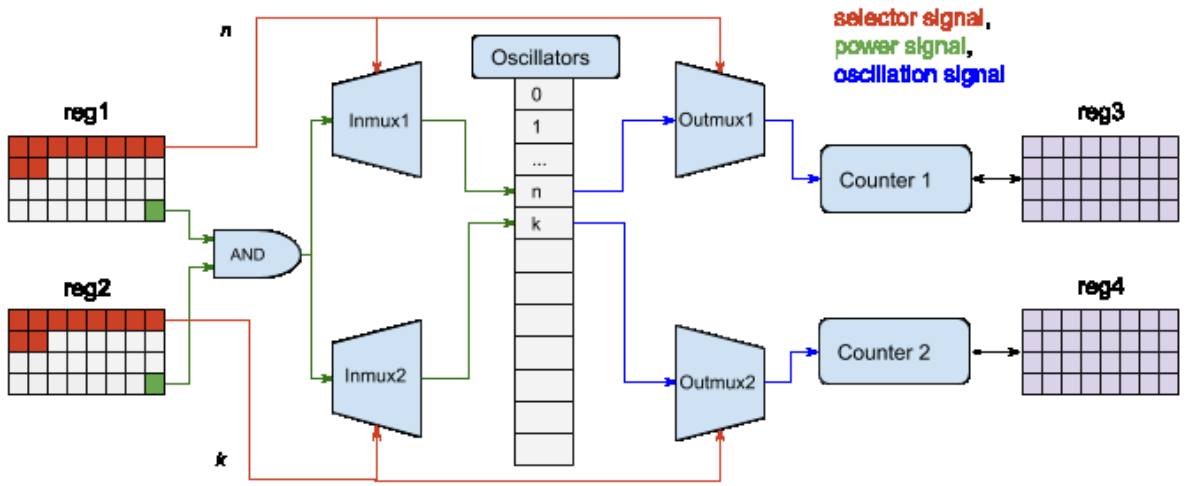


Figure 7: *Block design of our RO-PUF*

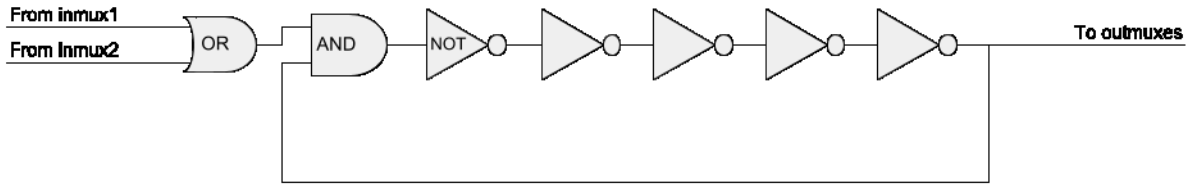


Figure 8: *Logic-level schematic of our ring oscillator. Each oscillator has an input from each inmux and an output to both outmuxes. A maintained signal on either input makes the output signal oscillate. Frequency has a unique component from manufacturing process variations.*

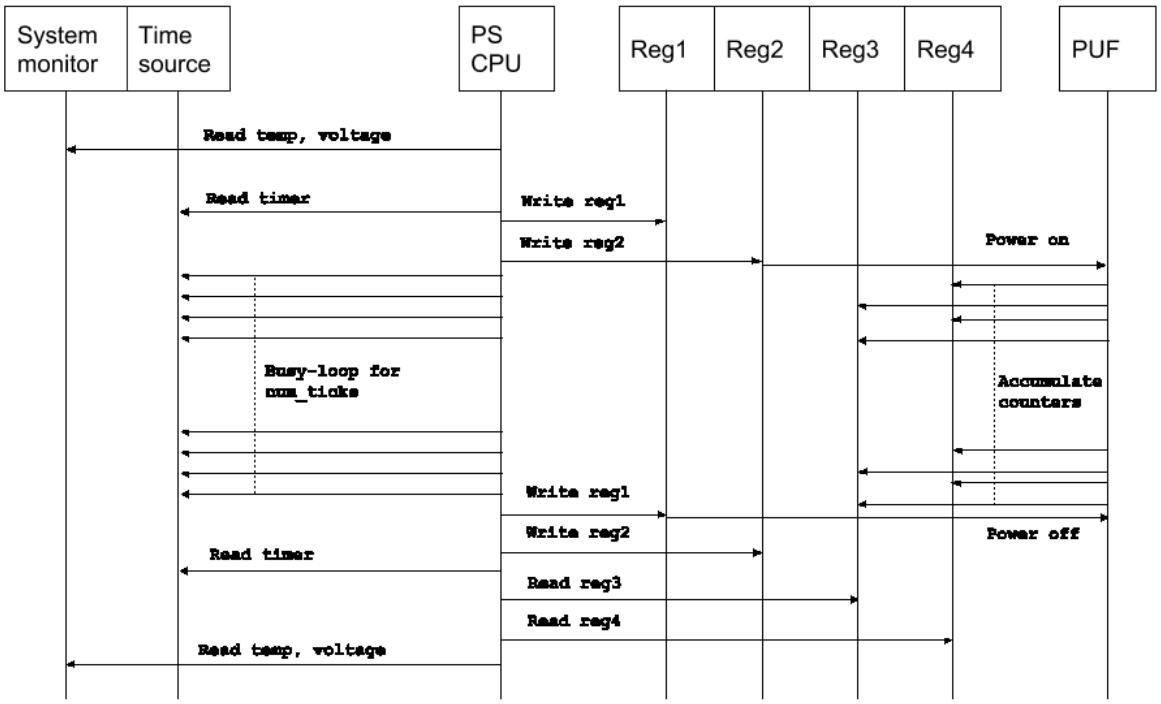


Figure 9: *Sequence diagram of a single RO pair measurement.*

The order of operations in a pair measurement is shown in figure 9. The measurement is initiated by the CPU by writing to the input registers (reg1 and reg2). While the measurement is running, the rising signal edge (oscillation) counts of the selected oscillators accumulate in the output registers (reg3 and reg4). Meanwhile the controller is continuously sampling the global timer available in the CPU, terminating the run when the elapsed time is longer than the `num_ticks` parameter by writing the power bits to zero. Temperature and voltage are read from the system monitor on both sides of the timed block and averaged. Controller program source code is included in the appendix.

Resource	# used	# available	% used
Look-up table (LUT)	11388	14400	79.1
LUTRAM	72	6000	1.2
Flip-flop (FF)	1558	28800	5.4

Table 3: *Total resource utilization, including selection and interface logic.*

Table 3 presents the FPGA resource utilization of our placed and routed design. The most prevalent components are Look-up tables, which are elements that can be programmed to act as any logical gate: they can map any logic input to any logic output. Flip-flops are essentially single-bit registers that hold any state assigned to them until they are reset. We can see that we have used almost all of the limiting resource (LUTs) on this chip. Further study would be required to achieve an accurate understanding of the scaling behavior of the selection logic (the inmuxes and outmuxes). The counters, the system monitor and the AXI interface do not need to scale and have constant resource requirements.

Modules

The RO units are divided into modules in order to reduce the effect of systemic variation on the oscillator comparisons (Maiti & Schaumont 2010). As the routing and placing is performed automatically by the Vivado software, we do not know (without performing manual examination after each placing) where exactly each oscillator is placed. The modules allow us, within a certain degree of accuracy depending on module size, to always know where on the chip an oscillator with a certain index (selector address) is located. See Figure 10 to see how the modules were placed on the FPGA.

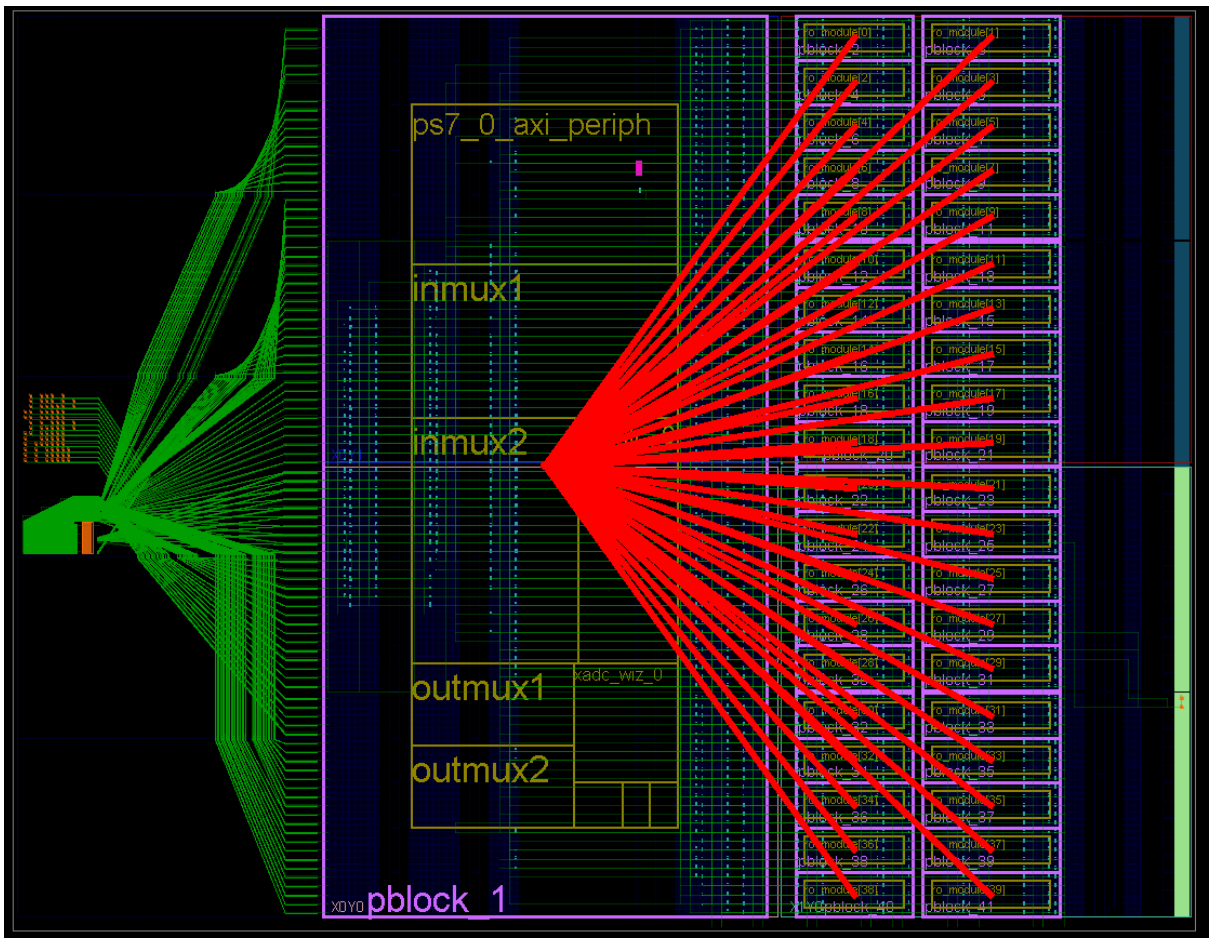


Figure 10: *Routing and placement of the design on the Zynq-7000 as represented by the Vivado hardware design suite. The red lines represent "bundle nets" between pblocks. Note the oscillators divided into modules on the right hand side.*

Oscillator selection

We have implemented three strategies for selecting which oscillators to include in the CRP: the *naive* strategy, the *filtered naive* strategy, and the *Longest Increasing Subsequence Algorithm* (LISA) of Yin & Qu (2010).

Naive selection

In the *naive* selection scheme, we always select oscillators *n* and *n+1*, and encode their relative speed into a single response bit. If oscillation count of *n* is larger than the oscillation count of *n+1*, the bit is 1, otherwise 0.

Pseudocode:

```
for n from 0 until count_of_oscillators-1; n=n+2
      oscs[n], oscs[n+1] = run_oscillators(n,n+1)
if oscs[n] > oscs[n+1]
      return 1
else
      return 0
```

Strengths:
In naive selection each bit will be independent, i.e. no oscillator is included in generating more than one response bit, and the relative order of two oscillators can not be inferred from other response bits. As the challenge bitstring is trivial, it reveals no information about the enrolling process. The naive method also avoids systemic chip variation, by 24/25 of the time selecting a pair within the same module and 1/25 of the time from a neighboring module (due to each module containing 25 ROs). If we would e.g. pair the oscillators with lowest absolute values with the largest absolute values, on a biased chip the winning oscillators would all be on one side of the chip, and the response would be biased and predictable.

Weaknesses:
The naive method is likely not an optimal way of selecting oscillators. We do not make any effort here to pick particularly good matches from the available population of oscillators, and as such some of the oscillators we select have frequencies inside the random error threshold, and have unstable response bits.

Filtered naive selection

The *filtered naive* selection scheme works on the same base idea as naive selection, but we filter out all potentially unstable pairs in the enrolling phase.

Pseudocode:

```
// do K measurement runs (e.g. with K from 5 to 10), ideally
// slightly outside the extremes of the desired operating
// temperature range; this should guarantee reliability within that
// range while keeping as many bits as possible.
for k from 0 until K; k=k+1
```

```
      For n from 0 until count_of_oscillators-1; n=n+2
            oscs[n][k], oscs[n+1][k] = run_oscillators(n,n+1)

// detect overlapping pairs in the enrolling data
if min oscs[n] < max oscs[n+1] or max oscs[n] > min oscs[n+1]
      // reject this pair
else if oscs[n] > oscs[n+1]
      return 1
else
      return 0
```

Strengths:
Filtered selection should be completely error-free within the environmental parameters of the data used for calibration, and thus achieve perfect reliability within those conditions.

Weaknesses:
Filtering eliminates bits from the CRP, likely losing entropy in the process. Filtering is also unpredictable, as we do not know beforehand how many pairs will survive the filter; there might be as many as 95% of bits remaining after the filter, or only 50%.

The result of the filtering is dependent on the calibration data used in the enrolling procedure. Temperature variation during the calibration run is the main factor affecting speed of oscillators in the enrolling data. A large temperature variation within the set of data implies larger variations and more bits will be filtered out, while a too small temperature variation will reduce the reliable operating temperature range of the PUF. Using a dataset with a somewhat larger temperature range than the expected operating temperatures should be used to produce a reliable PUF within the operating range.

As filtering eliminates pairs with the smallest differences, it likely to make the bits of the PUF more predictable, i.e. reduce the inter-distance across devices. This is likely detrimental to the average entropy gained per bit, as intuitively the surviving bits would be more likely to have across-devices bias.

LISA

LISA stands for Longest Increasing Subsequence Algorithm, presented by Yin & Qu (2010). Yin and Qu reported extracting a maximum of 480 reliable bits from 288 oscillators, thus having an efficiency of n*1.67. This is much more than the $n/2$ ratio of bits from normal RO pairing (like our naive selection), of which all bits might not be reliable.

The idea of LISA is to group the RO population into subsequences and encode the information embedded in their mutual order as bits. LISA uses a frequency threshold to define the minimum difference required between two oscillators for them to be permitted in the same subsequence. We use here a Lehmer encoding for denoting order within the subsequences as utilized by Maes et al (2012).

LISA requires the PUF to be controlled. The CRP challenge will also be more complicated, as the controller needs to know precisely which ROs were grouped in which subsequences in the enrollment phase. The mutual order within those subsequences is the secret included in the response. All oscillators will be run and their order evaluated and encoded by the controller.

LISA pseudocode (adapted from Yin & Qu 2010):

```
// "sorted" contains the ROs sorted by their oscillation minima;
// "f_th" is the threshold frequency, used as a safety margin
findReliableLIS(List sorted, Int f_th)
n = sorted.length
create stack ST_0, push sorted[0] to it
h <- 0
for j <- 1 to n
      top <- the top ring osc on stack ST_h;
      if ((sorted[j].fmin - top.fmin > f_th)
            && (sorted[j]. fmax - top.fmax > f_th) )

            h++
            push sorted[j] to new stack ST_h
      else

            find the stack ST_p with the largest index p that has its
            top element's fmax smaller than sorted[j].fmax - f_th
            and fmin smaller than sorted[j].fmin - f_th.

            if p ! = null
                  push sorted[j] to ST_p+1

return sequence <sorted[j_1], sorted[j_2] ... sorted[j_h]> where
      sorted[j_h] is the top element of ST_h
```

Strengths:
The response bitstring will no longer be of fixed length, but will vary based on the sequencing performed by LISA. It is also much harder to deconstruct the bit string as after joining the bit-encoded orders together, there is no obvious way of distinguishing which specific oscillator was used to derive which bit.

LISA allows us to extract many more bits from the oscillator frequencies, which likely but not necessarily improves the entropy strength of the response. To estimate the entropy carried, we sum over the set of subsequences

`H(Y) = sum(log2(b!))`

where *b* is the length of each subsequence in turn. This provides an entropy upper bound for a bit string built from combining the binary encodings of the relative order of each ring oscillator within its subsequence (Maes et al. 2012). However, this estimate has pitfalls.

Weaknesses:
As a single RO pair no longer produces a single bit, the amount of entropy carried by each bit is harder to estimate. Some bits in the binary encodings might be biased; also, while previously a pair yielded a single bit, every RO will produce at least one bit, no matter how biased. In a 2-RO sequence, encoding the order will now yield 1 for the faster RO and 0 for the slower RO. The information carried by a 2-RO sequence is thus exactly the same as previously carried by one pair; however, it is now encoded in two bits. A single-RO sequence always yields 0, and trivially carries no entropy if the attacker knows the sequence lengths, which would be contained as plain text in the challenge.

There are *n!* possible ways to order a sequence (Maes et al. 2012). For this to yield the estimated amount of entropy, the distribution of orderings should be uniformly distributed, which is not the case in reality, as the relative frequencies (at least in our implementation) have bit-specific bias across devices. As such, the formula used by Maes et al. (2012) will only estimate LISA's entropy against an attacker who knows we use LISA, but is ignorant of these biases.

The sequencing process is very sensitive to changes. Minuscule changes in the calibration data can alter the entire sequencing in ways that do not suggest any relation to the old one. This sensitivity will not be a problem in evaluations after the enrolling, as the threshold between frequencies will not be enforced when evaluating the order within the subsequences.

Choice of the threshold frequency will affect both the result of the sequencing and the reliability of the bits. A larger threshold frequency will lead to better reliability, but shorter subsequences and thus fewer bits, and vice versa. A too large threshold will prevent any subsequences from forming, thus collapsing the PUF's entropy output to zero.

Data collection

The main dataset was collected from 5 sequential "runs" on each of the 5 devices, with each "run" going through each *(n,n+1)* RO pair 4 times, for a total of 20 measurements for every oscillator in the dataset. The devices were kept in room temperature before powering on. The chips heated to the starting temperature during powering up and the programming of the FPGA. During the run the increasing chip temperatures in the main dataset are caused by the running PUF. Run length was configured at 10 million CPU cycles per pair, which translates to 15 milliseconds per pair with a CPU clock frequency of approximately 667 MHz (reported as 666 666 687 Hz). Temperature variation within this dataset is presented in figure 11; temperatures ranged from about 40°C to 58°C.

A secondary dataset with high temperature variations was also collected, with a single run performed at each of "cold" (boards pre-cooled in a freezer), "mild" (room temperature) and "hot" (heated with a 90°C hot air blower) environments. Results on this dataset are reported separately when evaluating the error from temperature. In addition, a shorter "jitter dataset" was collected for exploring gaussian random error. Many other runs were performed for controller program testing and other exploratory purposes, but not included in computing the reported statistics. Our controller program output the measurement data in a CSV format, the structure of which is presented in tables 4 and 5.

Header	Data type	Description
DEVICE	integer	ID of the test device, manually entered in the terminal before the test run
COUNTER	unsigned long	Indicates when in the run this RO pair was run
POSITION	unsigned	Indicates the position of this oscillator within the RO pair
RO_INDEX	unsigned long	The index (address) of the oscillator
INCREASE	unsigned long	The increase in the oscillation counter for this oscillator, read from the output registers
CPU_TICKS	unsigned long	Increase of the CPU clock during the run
TIME_DELTA	float	CPU_TICKS divided by the reported CPU clock frequency
TEMPERATURE	float	On-chip (XADC) temperature measurement
VCC_INT	float	On-chip (XADC) VCC_INT voltage measurement
VCC_AUX	float	On-chip (XADC) VCC_AUX voltage measurement

Table 4: *CSV output format produced by our controlling program.*

```
DEVICE; COUNTER; POSITION; RO_INDEX; INCREASE; CPU_TICKS; TIME_DELTA; TEMPERATURE; VCC_INT;
VCC_AUX;
0; 0; 1; 0; 3598563; 10001302; 0.015002; 46.645935; 1.799675; 0.993324;
0; 0; 2; 1; 3668059; 10001302; 0.015002; 46.645935; 1.799675; 0.993324;
0; 1; 1; 2; 3695296; 10001232; 0.015002; 46.680542; 1.799583; 0.993164;
0; 1; 2; 3; 3542482; 10001232; 0.015002; 46.680542; 1.799583; 0.993164;
0; 2; 1; 4; 3556774; 10001224; 0.015002; 46.649780; 1.799698; 0.993118;
0; 2; 2; 5; 3569668; 10001224; 0.015002; 46.649780; 1.799698; 0.993118;
0; 3; 1; 6; 3580311; 10001252; 0.015002; 46.661316; 1.799721; 0.992981;
0; 3; 2; 7; 3568520; 10001252; 0.015002; 46.661316; 1.799721; 0.992981;
0; 4; 1; 8; 3603311; 10001252; 0.015002; 46.669006; 1.800064; 0.993050;
0; 4; 2; 9; 3591443; 10001252; 0.015002; 46.669006; 1.800064; 0.993050;
0; 5; 1; 10; 3586012; 10001236; 0.015002; 46.695923; 1.799812; 0.993050;
0; 5; 2; 11; 3760374; 10001236; 0.015002; 46.695923; 1.799812; 0.993050;
0; 6; 1; 12; 3601267; 10001212; 0.015002; 46.576721; 1.799240; 0.992569;
```

Table 5: *Example CSV output from the beginning of a run, including the header row.*

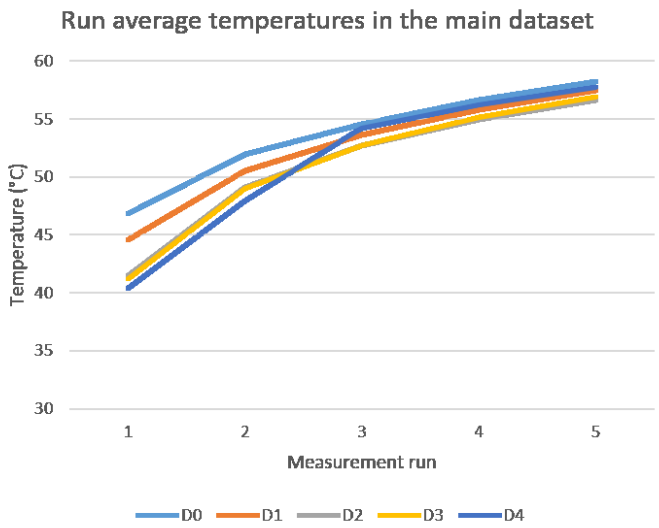


Figure 11: *Average temperatures for each device during the main dataset runs.*

Running order

As we aim to understand our error sources, we have reasons to vary the parameters for running each RO pair. In addition to position within the run (as temperature tends to rise towards the end), position within the pair might affect the oscillator frequency. For singling out different error sources, each RO was run four times in each test run, differently ordered:

```
For N from 0 to 999, run (N, N+1)
For N from 999 to 0, run (N, N+1)
For N from 0 to 999, run (N+1, N)
For N from 999 to 0, run (N+1, N)
```

As can be seen, the main dataset always has the ROs paired *(n*, *n+1)*. To detect if the index of the other RO affects the result, we ran some separate measurements with the configuration

```
For N from 0 to 499, run (N, N+500)
```

As any such effect was not found based on this run, we could stop considering this error.

Tools and languages

Free versions of Xilinx SDK and Vivado software were used for design and development. Verilog was used as the hardware definition language and C was used as the controller software language. Software was developed using a “standalone” hardware definition, i.e. without any operating system on the board, to minimize dependencies and complexity. Bitstreams (FPGA configuration data) and the PS programs were deployed on the board using Xilinx SDK’s “run using system debugger” mode.

The basic elements within each oscillator were placed using “hard macro“ constraints. Grouping the oscillators within physically restricted areas on the PL was done using the

pblock functionality in Vivado. The pblock areas were manually designated on the chip, and the design tool placed each oscillator unit within the pblock indicated by its index. Our implemented bitstream contains 40 pblocks containing 25 ring oscillators each for a total of 1000. The modules are as symmetrical as possible, with 5*5 configurable logic blocks each. We thus know the approximate physical location of each oscillator.

We communicated with the device using a serial connection over USB. The development machine was a Windows 7 PC, with a PuTTY serial communications terminal. Running the PUF was achieved by sending input to the controller program over the serial terminal. Output was collected from the controller program by printing CSV to the terminal, and logging the terminal sessions to file.

Data analysis was performed using Microsoft Excel. LISA was implemented in Python.

Analysis methods

There are two sides of analysing our PUF: determining how good our PUF is by performing measurements, and analysing the effect of the various sources of undesirable error.

Measurements

Intra-distance and inter-distance
Intra-distance represents what can be called reliability, reproducibility or stability (Suh & Devadas 2007, Maes & Verbauwhede 2010). Intra-distance is the difference within the same device using the same challenge, so it measures bit flips between different runs on the same PUF instances. The ideal intra-distance is 0%. This means no two runs of the same PUF instance with the same challenge flip any of the bits. Thus it is always possible to reproduce the expected response.

Inter-distance represents what has been called uniqueness or unpredictability (Suh & Devadas 2007, Maes & Verbauwhede 2010). Inter-distance is the difference between two PUF instances using the same challenge, so it measures bit flips between devices. The ideal inter-distance is 50%. This means that on average, any two runs on different PUF instances with the same challenge have half of their bits flipped. This indicates perfect unpredictability, on average. The distribution should be inspected to detect outliers.

Both distances are measured as the relative Hamming distance for pairs of responses to same challenge. The relative Hamming distance is simply the number of flipped bits divided by the total number of bits in the bit string.

To achieve better results, both measures are calculated between as many runs and devices as possible. Minimum, maximum and standard deviation are also to be reported (Maes 2013: 97). To estimate the efficiency of the PUF, we report both the raw number of bits, and the estimated entropy carried by those bits as extracted by the various methods.

Entropy
Shannon's entropy can be used to measure the "surprise" i.e. amount of randomness contained in random variables. Cryptography relies heavily on entropy as a measure of the strength of secrets. Entropy is a central measure also in PUF evaluation, as it takes into account the estimated quality of bits in addition to their quantity.

Estimating entropy comes with a number of pitfalls. Most importantly, it should be remembered that entropy calculations are always upper bounds. There might be predictable behavior that is not covered by our estimates, and knowing more about the PUF's behavior will lower the bound for a certain attacker. For an ignorant attacker, the entropy of a PUF response is always trivially equal to the length of the response bitstring (Maes 2013: 109). However, if an attacker can gain more information about the PUF's behavior, the responses can be better predicted and no longer contain as much effective entropy. As such, it is best to consider our entropy calculations to be optimistic estimates.

We will estimate the response entropy for an attacker who has information about bit-specific bias (i.e. measured expected value) across several devices (Maes 2013: 109). This knowledge could realistically be obtained by stealing and analysing a number of instances of our PUF. The entropy carried by a biased bit Y is estimated using the Bernoulli distribution ("unfair coin") formula

```
H(Y) = -log2(p)-log2(1-p)
```

where p is the probability of the bit being "0", as estimated from the observed expected value (Maes 2013: 109). Sample size affects the accuracy of the estimate.

Sources of error

Temperature
A higher temperature lowers the average frequency of an oscillator. Different RO units react differently and in an unpredictable way, and this might cause bits to flip (illustrated in figure 12). Temperature and voltage were sampled using the on-chip Xilinx Analog-Digital Converter (XADC), a built-in feature of the chip.

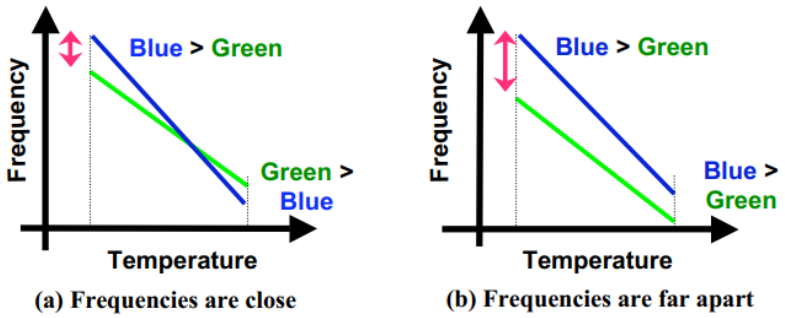


Figure 12: *Temperature changes can cause bit flips when the oscillator frequencies are close, due to the different amplitude of the temperature effect (Suh & Devadas 2007).*

The effect of temperature on the stability of our PUF was estimated by collecting a separate dataset, which contains from each device a "cold" measurement taken after cooling the devices in a freezer (on-chip temperatures starting from 6°C and reaching 20°C during the run), a "mild" measurement taken in room temperature (chip temperatures about 50°C), and a "hot" measurement taken with a 90°C a surface-mounted device (SMD) rework tool's hot-

air blower directed at the chip (chip temperatures reaching 90-92°C). The design temperature range of our device is 0-85°C, so extending this range further might cause damage to the device or artefacts resulting from device malfunction.

Voltage
We did not measure changes in voltage, due to resource constraints. We assume that voltage changes reasonably expected in a normal operating environment have a similar effect as temperature changes, and thus stability can be protected using the same mechanisms, such as filtering. We consider behavior under voltage changes a reliability issue, and not an attack vector.

Aging
Truly evaluating the effects of long-term operation and aging would require a long timeline and longitudinal study. Additionally, the FPGA fabric in our use-case might spend most of its lifetime in other than PUF use, so the components used for RO units might see wildly differing usage, and thus wear differently than in a lab experiment. We aim to cover our requirements related to aging by considering PUF control mechanisms aimed at early detection of aging deterioration.

Placing & routing differences
We will not be able to eliminate all routing differences from the design. Rather, we have designed the RO units to be small and modular, each fitting inside a single CLB, where we can reasonably trust that the routings are reasonably design-identical for most of the units. Every CLB has its own routing matrix where the signals of basic elements are connected according to the bitstream. While there is no reason for the synthesis step to make different routings within different CLBs, there are no guarantees with the current design.

The selection logic before the ring oscillators will have significantly varying routing delays. However, if it takes longer for a signal to reach the RO, the shutdown of power should take equally long. Routing of the counters and outmuxes should not be an issue, as these are completely combinatorial (asynchronous) components, and any signal sent by the oscillator will reach the counter register before the PS will be able to read the register.

Positional difference
Being selected as the "first" or "second" oscillator within the pair might cause effects on the oscillation average. Such effect might be caused by routing delays in the input selectors, and will be investigated.

Delays in the CPU
The PUF is controlled from the CPU via the input registers. According to the output of the controlling program, there is a non-zero, non-constant delay associated with reading and writing the PUF registers. This is expected to lead to small deviations in run lengths and thus oscillation counts.

Systemic manufacturing error
As suggested by Maiti & Schaumont (2010) and Zhang et al. (2014), there might exist significant systemic variations in RO speeds both within and between devices. To detect and counter this effect, we have placed the ROs in modules, making their indices to indicate their

approximate physical location on the chip. This allows us to validate the existence of any systemic effects and mitigate their detrimental effects.

Gaussian jitter

The output of ring oscillators contains a random component called Gaussian jitter (Valtchanov et al. 2008). Due to the existence of this jitter, there will always be unexplained noise in the RO frequencies, even if we account for all factors that we can influence with our design. A separate dataset was collected to evaluate this error source.

5 Evaluation

Measurements

The measurement results of the naive selection methods are presented in table 6. Results from LISA are not included in this table, as they are not meaningfully comparable. The number of bits and challenge structure in a LISA evaluation are dependent both on the device and the threshold frequency used in the enrolling procedure, making it difficult to compare behavior across devices using these same metrics.

	Unfiltered naive	Filtered naive
Relative intra-distance		
Average	1.97 %	0.00 %
Min (best case)	0.60 %	0.00 %
Max (worst case)	3.40 %	0.00 %
Relative inter-distance		
Average	34.79 %	32.60 %
Min (worst case)	31.80 %	26.69 %
Max (best case)	38.20 %	38.02 %
Average output bits	n/2 = 500 bits	n/2*85.76 % ~= 429 bits
Average entropy when attacker knows bit-specific bias	295 bits	295*85.76% ~= 253 bits (1
Average bits of entropy per output bit (entropy density)	0.59	0.51

(1: assumes all bits filtered out are non-zero-entropy bits, that provide the expected entropy of a non-zero-entropy bit; in other words, a reasonable worst case assumption.

Table 6: *Measures from our enrolling methods.*

We can consider our results quite good. Even with the most simplistic and naive method, where we simply compare each oscillator with the next, we have achieved a low figure of 2% average relative intra-distance, i.e. 98% reliability. In the worst case, where the attacker knows the average CRP bit behavior of the rest of the PUF population, our single CRP of 500 bits carries an estimated 295 bits of entropy. This inefficiency reflects the fact that our CRPs are somewhat predictable across devices, as represented by the inter-distance measure. With the filtered method, the worst-case inter-distance between two PUF instances is as low as 26.69%, meaning that 73.31% of bits did not flip in the comparison between those instances. An attacker can benefit from this by assuming that more than half of bits will remain the same. This will not cause a collapse of our PUF as this effect is accounted for in our entropy estimates, but reducing this effect would definitely have a positive impact. The

distributions of the distance measures and the effect of filtering can be seen in figures 13 and 14.

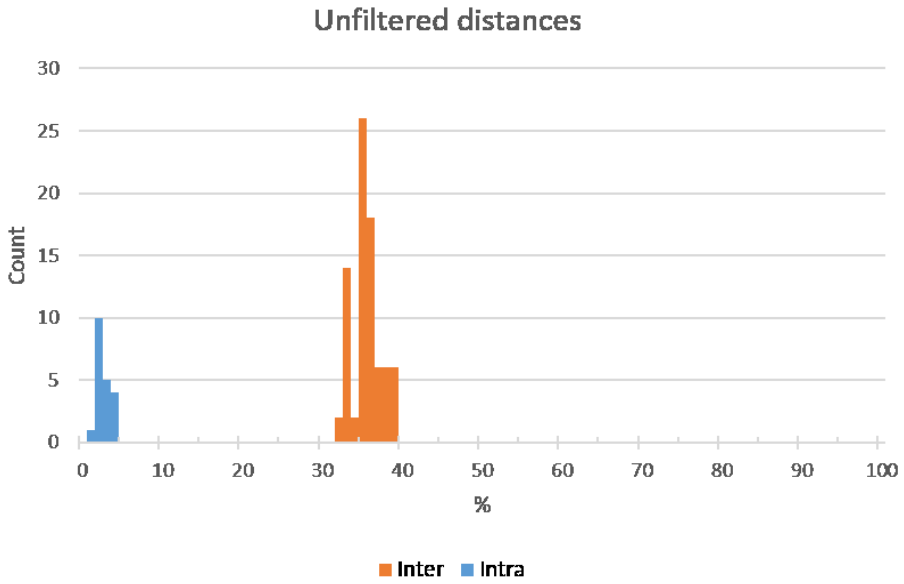


Figure 13: *Distribution of run intra- and inter-distances in the unfiltered case.*

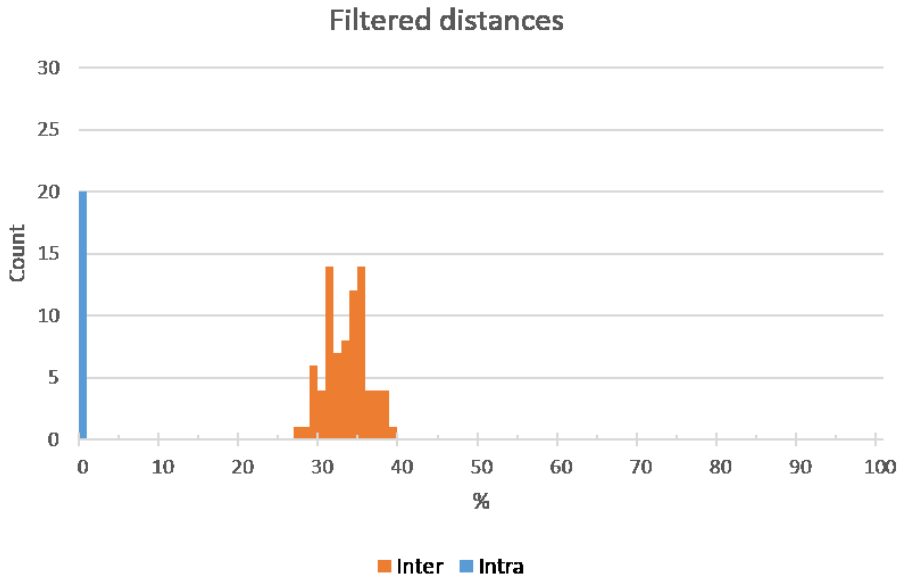


Figure 14: *Distribution of run intra- and inter-distances in the filtered case. Intra-distance is now ideal at 0%, but inter-distance has deteriorated.*

A “bit flip” is defined as a changed output bit value within a set of 5 measurements. Thus each intra-distance data point is calculated from 5 measurements of the same pair, instead of all 20. The reason for this is that each different run ordering (explained in the methodology section) is considered separately, resulting in 4 intra-distance values per device.

Inter-distance “flips” are calculated between comparable bits across devices. Each data point is a pairwise comparison. As there are 5 devices, each of the 20 intra-distance data points can be compared to 4 other devices, and so there are 80 inter-distance data points.

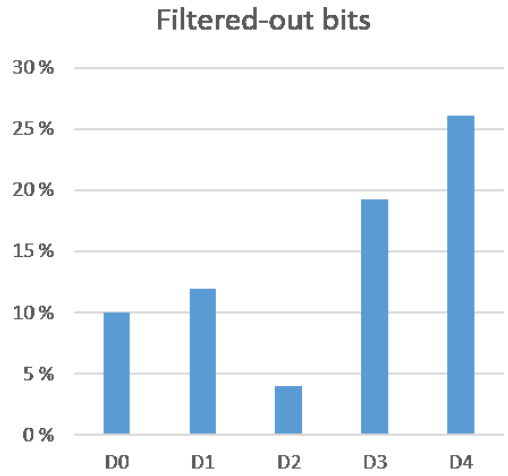


Figure 15: *The relative number of bits filtered out per device.*

As can be seen figure 15, the number of bits filtered out is very variable. While this means most of the devices are actually affected less than the average of about 15%, some devices can have their response length unexpectedly restricted.

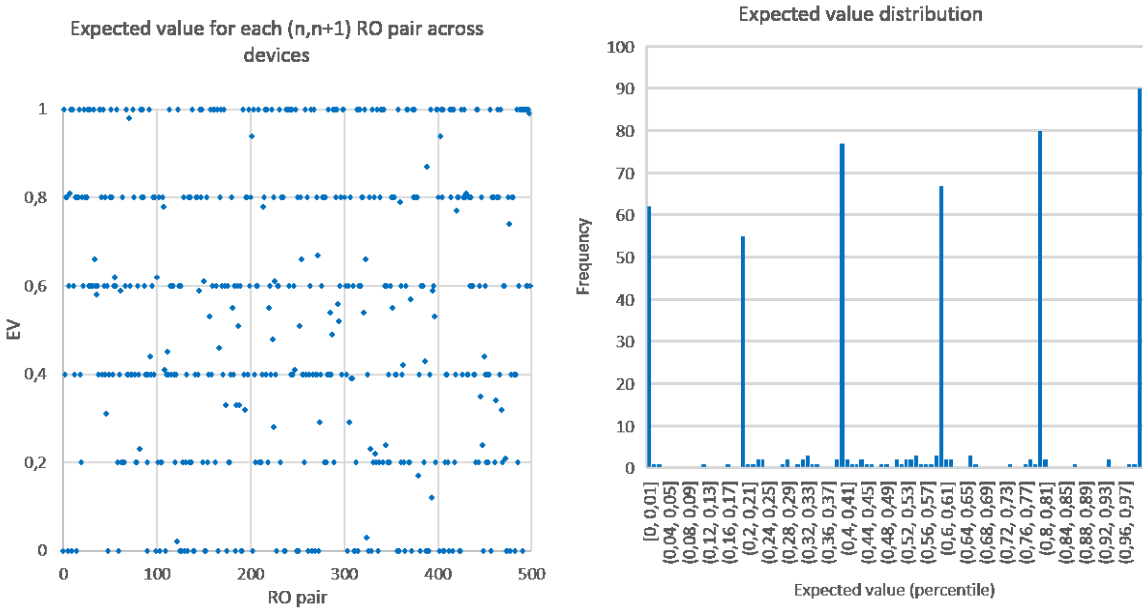


Figure 16: *Expected values per naively selected bit and their distribution of expected values.*

The clustering at 0.2 intervals in figure 16 is a result of the small number of tested devices, as any bit is likely to always be the same in the same device (low intra-distance). Bits with an expected value not one of these values has flipped within some device. Bits that have always been observed to return either 0 or 1 have that as their respective expected value, and these bits are useless from an entropy point of view. It is likely that evaluating more devices would reduce the number of these edge cases. In the absence of significant bias, with more data the distribution would start converging on the average. Figure 17 shows the effect of the Bernoulli distribution entropy formula. The same clustering is visible, with expected values close to 0.5 producing the highest entropy values.

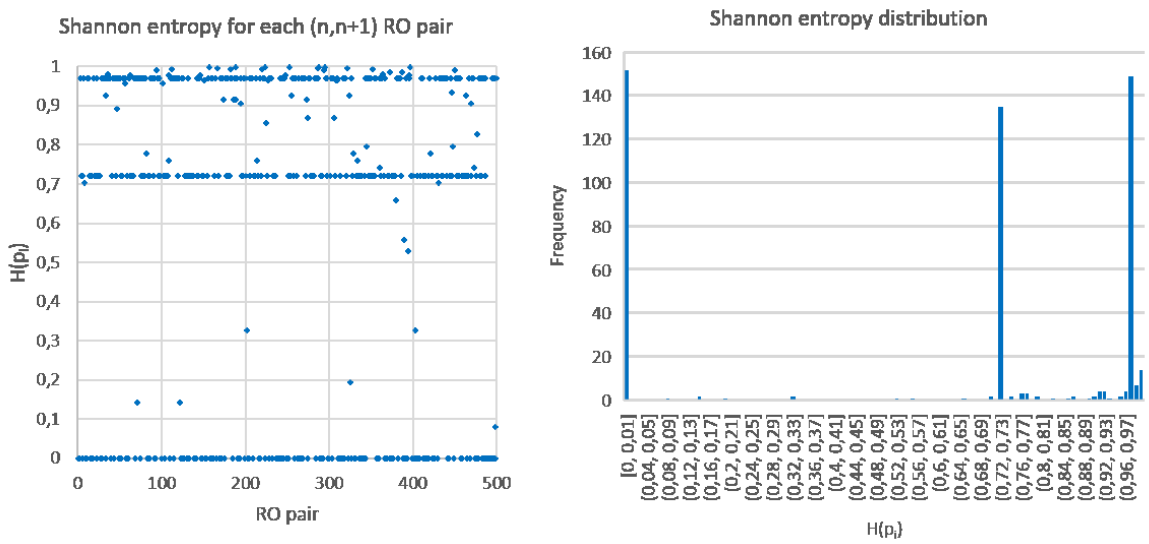


Figure 17: *Shannon entropy $H(p_i)$ for each naive bit estimated using observed bit-specific Bernoulli distributions.*

LISA

Evaluating LISA is not straightforward, as was already explained in the methodology section. We have implemented LISA and used it to sequence the ROs separately within each RO module (the module split is discussed in the Methodology section). This is done in order to not fall for systemic bias. Without restricting LISA to only perform sequencing separately within each module, device #2 produces an almost trivial result due to systemic bias (behavior of device #2 is visible in figure 27, where systemic bias is explored). All the oscillators on the slower half of the device had their response bits zero due to being the slowest ones in their subsequence. LISA is particularly vulnerable to bias, as the challenge reveals how the oscillators are split into sequences; this information may reveal systemic bias to an attacker.

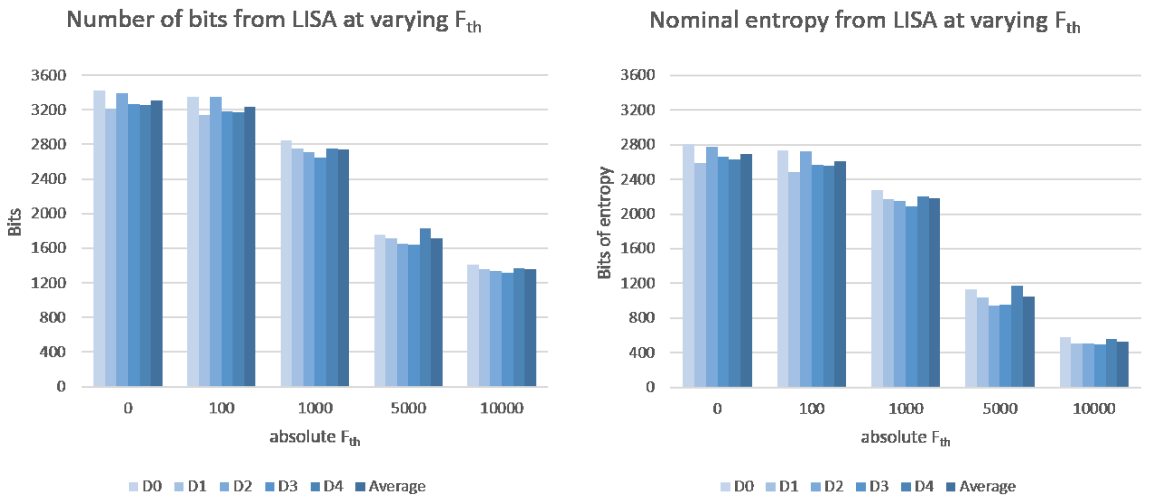


Figure 18: *Raw number of bits and the nominal Shannon entropy extracted from the main dataset using per-module LISA with different threshold frequencies and devices. Possible "raw bit" range is 1000 to 3800 bits, possible entropy range is 0 to 3347 bits.*

Figure 18 presents output lengths and nominal entropy estimates from LISA using different threshold frequencies. The entropy contained in a single sequence in calculated as *log2(b!)* where b is the length of the sequence (Yin & Qu 2010, Maes et al. 2012). As *log2(1!) = 0,* these calculations take trivial sequences into account. However, as discussed in the methodology section, this calculation only applies if we would have a perfectly uniform probability for each ordering within subsequences, or against an attacker who assumes uniform distribution for lack of better information. We are not confident in this estimate of the entropy generated by LISA, and more robust methods have not been found in the literature.

Sources of undesirable error

Temperature

Temperature variations and aging are the main threats to the reliability of our PUF. Since both of them manifest similarly, as a creep of the oscillator speeds, we pay special attention to temperature effect. Figure 19 compares the speeds of a set of oscillators across a 60-degree temperature range. The general trend in the oscillation counts is similar, but individual oscillators react in slightly different ways, causing bit flips. Note the flip of oscillators 5 and 7, when their relative order changes. Gaussian jitter (random noise) might also cause flips; it is discussed later in this section.

Figure 20 shows averages from the separate high temperature variation dataset (discussed in the Methodology section). Runs were performed at the extremes and middle of the operating temperature range, and these were classified into categories "cold", "mild" and "hot" as shown in figure 21.

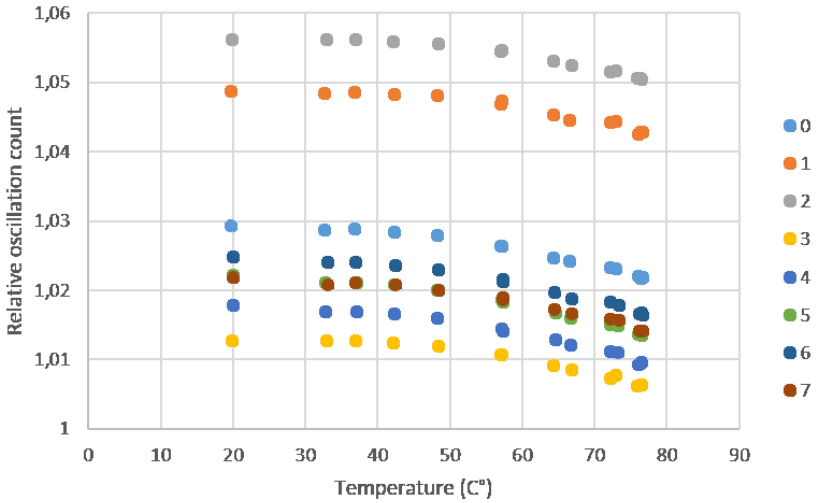


Figure 19: *The effect of temperature on the oscillation counts of a set of eight oscillators on device #0, across a 60°C chip temperature range.*

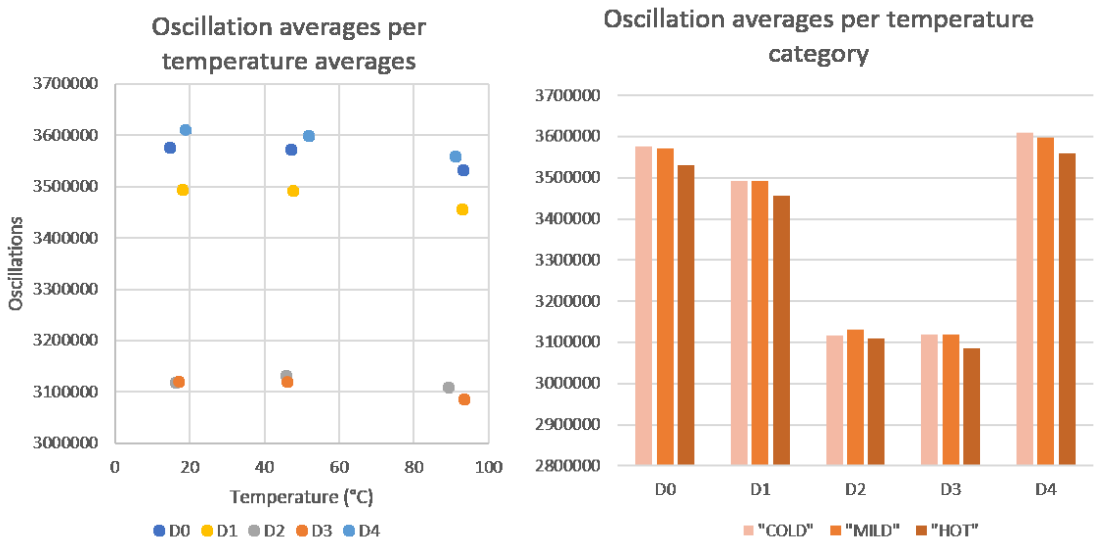


Figure 20 (left): *Run average temperatures in the high temperature variation dataset.*
Figure 21 (right): *Run averages temperatures grouped by category.*

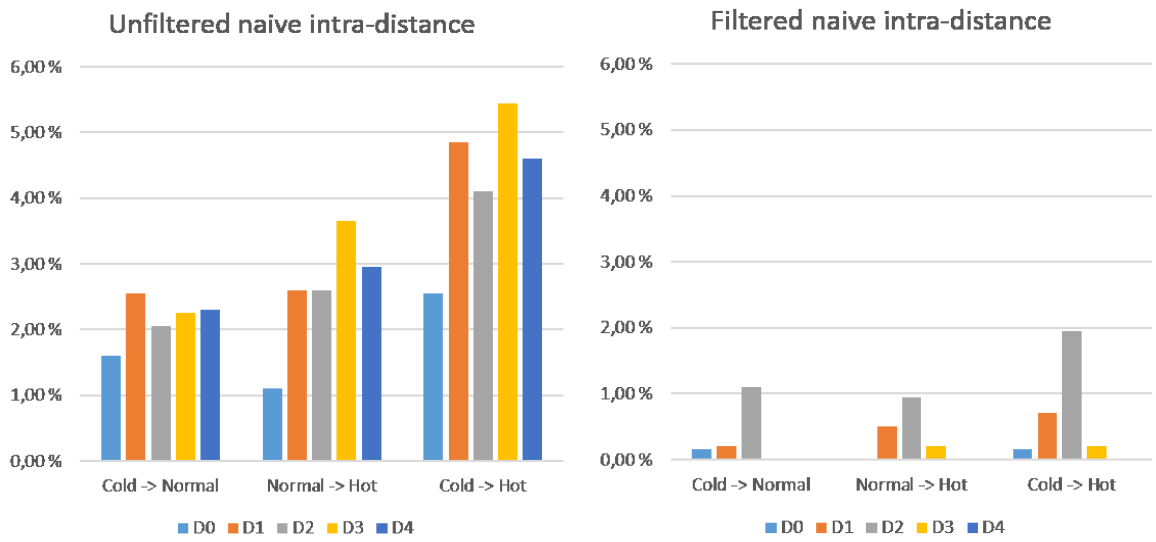


Figure 22 (left): *Raw intra-distances.*
Figure 23 (right): *Intra-distances when using the main dataset filter.*

In figures 22 and 23 we can see the effect of filtering on intra-distance at temperature extremes. Unfiltered, the bit flip rate is generally 4-5%, while the filtered error rates are generally very low. Notably, filtered device #4 has no bit flips at all in the entire 5 - 92°C range. The number of bits filtered thus seems to correlate with better stability, but we cannot draw robust conclusions from so small a sample.

Position
Positioning within the pair has been measured to have no effect, or at least the effect is irrelevant. Table 7 contains measures of the effect in the main dataset.

Average oscillation count difference between positions 1 and 2, across all devices	3.29
Largest difference between device averages between positions 1 and 2	14.42
Median oscillation count difference between oscillators *n* and *n+1*	25 425
Average oscillation count	3 567 454

Table 7: *Effect of position within the pair on RO speed.*

With each RO having been run equally many times in both positions, the difference was found to be 3.29 oscillations in favor of position 2 on average. The median difference between consecutive oscillators being 25 thousand oscillations and the absolute oscillation average being 3.5 million oscillations, we can dismiss the positioning within a pair as a non-significant source of error.

Routing error

Routing has the potential to be a major source of error. Figures 24 and 25 show hardware design tool screenshots of the implemented routing. However, it is unclear how accurately the design tool represents the exact physical details of the routing, as simplifications and approximations of normally irrelevant details might have been necessary to present the routing in a human-friendly way.

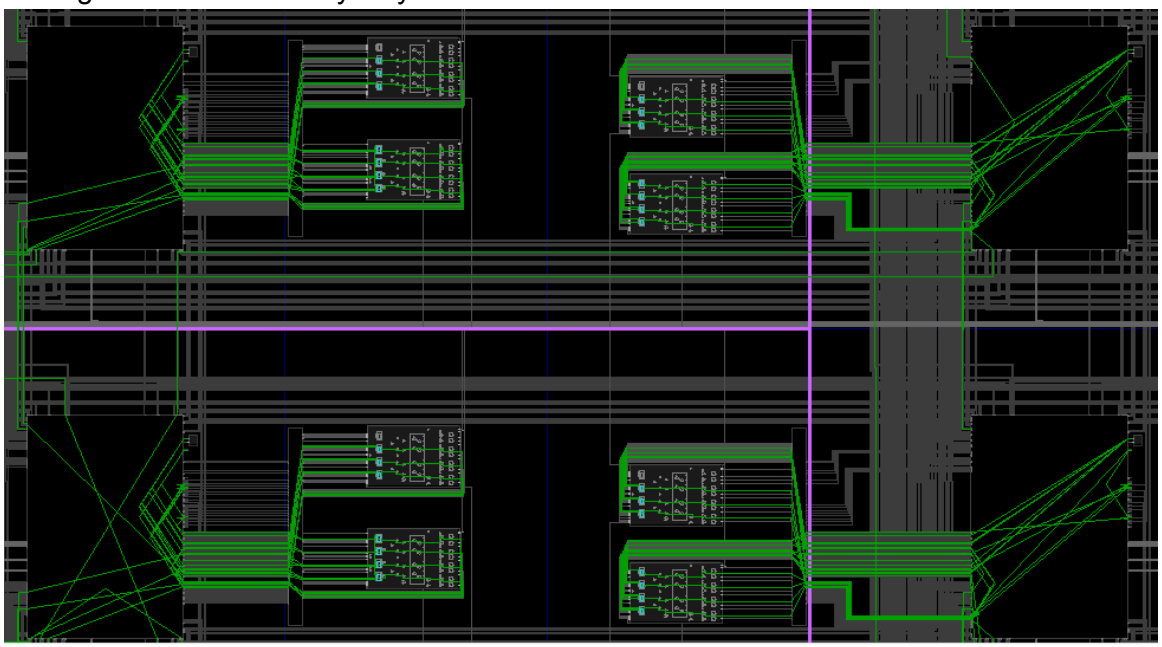


Figure 24: *Most oscillators have identical or very similar internal routings, at least within the same columns (routing matrix components are on the extreme left and right).*

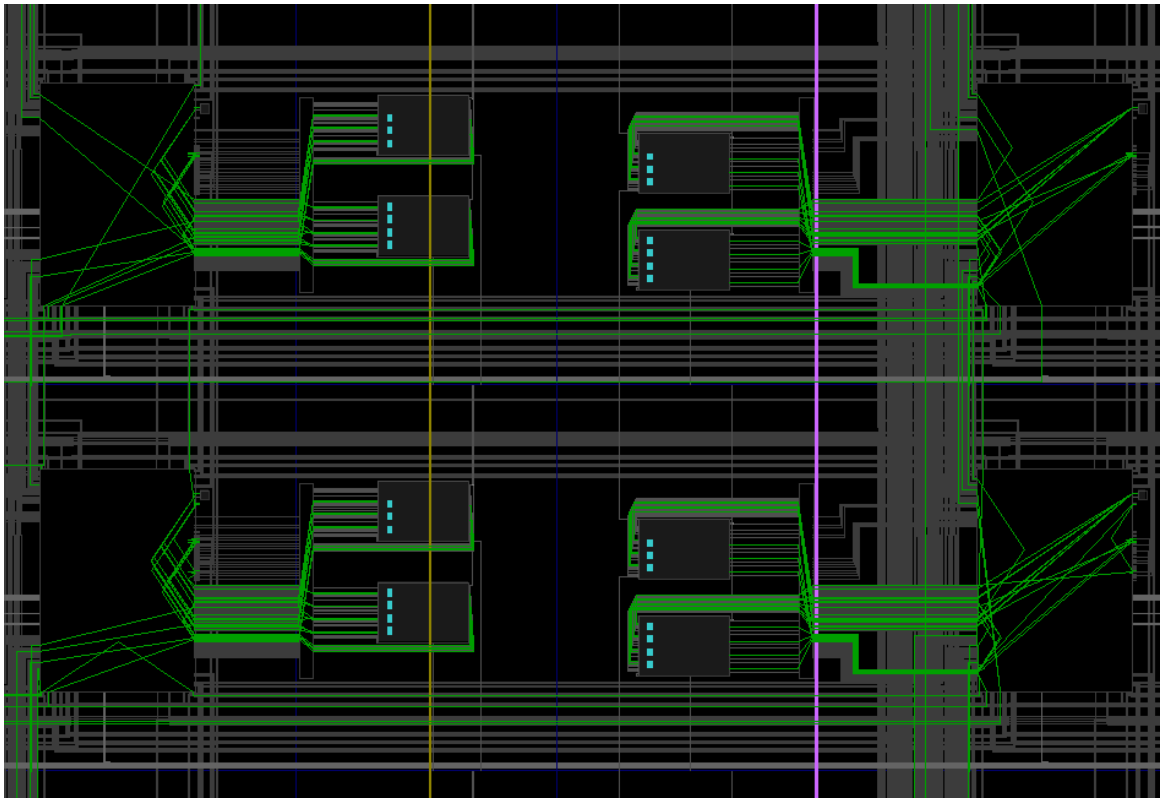


Figure 25: *On some rows, e.g. near the chip edges, the internal routing is no longer identical; compare the matrices.*

Delays in the PS
There is a small, varying delay in toggling power to the PUF circuit. In the main dataset, this delay varies from 1032 to 1114 CPU clock cycles. The variation (less than 100 cycles) within the delay is quite small as we have used 10M CPU cycles as the standard run length. The clock was sampled using the `XTime_GetTime` function, which samples the global timer register of the PS; the act of sampling the time also consumes CPU cycles. However, this delay should not have any effect, regardless of its size, whenever we compare the same oscillator pairs we run (as opposed to if we e.g. ran pairs as (n,n+500) and then compared pairs (n,n+1)). The units run simultaneously are powered by the same wire and any delay difference in powering up will compensate as a similar delay when powering down. Noise could be introduced to results if we ran and compared oscillators in different pairings.

Systemic error
The devices have three significant systemic characteristics.

- **Fast oscillators at the edges:** The most prominent systemic feature in the data is that there are much faster oscillators clustered at the extreme RO indices on all five devices (visible in figure 27). The cause for this effect is not clear as it might be an artefact introduced in the silicon chip manufacturing process, or caused by different routing within some oscillator units caused by the proximity of the chip edge. As the behavior of these oscillators seems to be dominated rather by predictable systemic variation rather than the desirable kind of random variation, they are likely to produce predictable bits that provide little entropy.

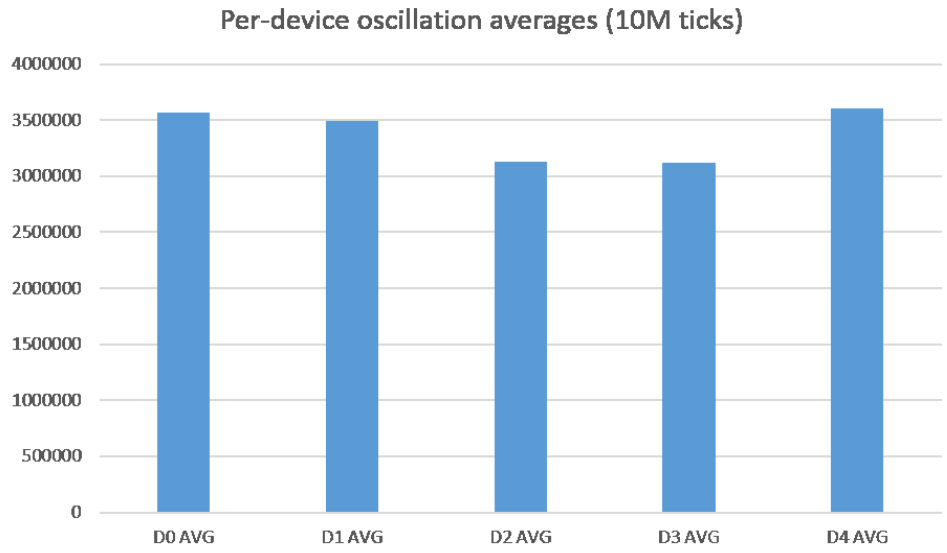


Figure 26: *Per-device oscillation averages across all measurements in the main dataset. Significant device-specific bias is visible even on absolute scale.*

- **Different average frequencies:** The second feature is the highly varying frequency averages between devices (figure 26). This does not affect our PUF, since all comparisons are relative comparisons within each device, but it reminds us of the fact that the devices are unique and their raw speeds are not comparable. The differences are not explained by temperature or voltage differences, so systemic manufacturing variation seems to be the only explanation.

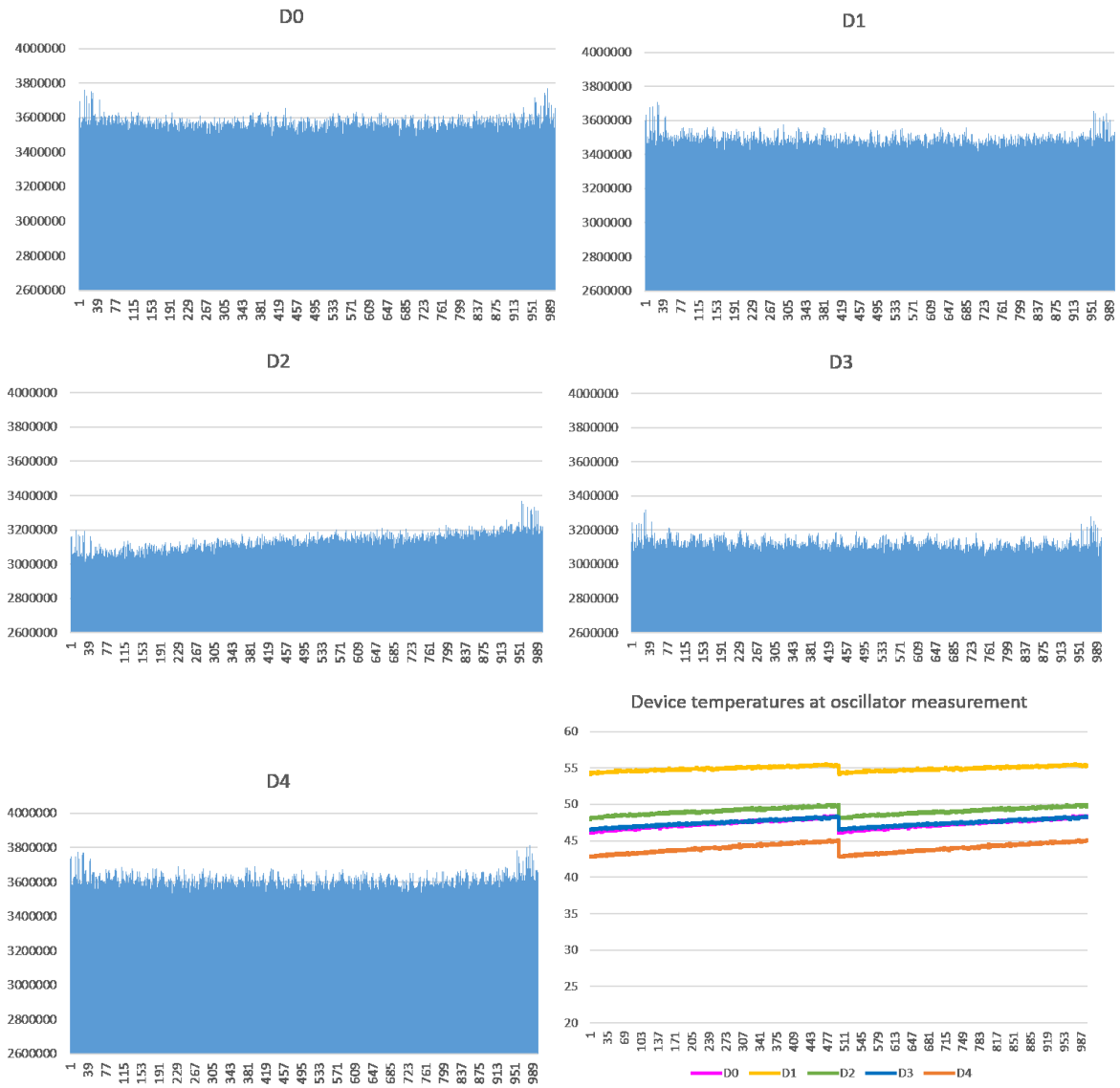


Figure 27: *Per-unit oscillation count charts on each device, based on a single measurement lasting 10M ticks. X-axis denotes oscillator index. On lower right are the device on-chip temperatures at each oscillator measurement. The temperature drop in the middle is caused by the use of the experimental oscillator pairing (n, n+500) in this run instead of the usual (n, n+1).*

- **Slopes:** The third interesting feature is the spatial bias in the frequency distributions of devices #2 and #3. Device #2 has a very uneven distribution of frequencies (see figure 27), with the average oscillator speed increasing with the oscillator index. Device #3 has a similar bias, but smaller and along a different axis, appearing as a sawtooth pattern in the chart. We believe these biases are caused by chip manufacturing irregularities. Finding these effects justifies the misgivings of Maiti & Schaumont (2010) about pairing oscillators that are not physically close to each other on the chip. Curiously, devices #2 and #3 also have lower average frequencies than the others, but we can't draw any conclusions from so few devices.

Grouping the oscillators in modules allows us to work around the problems caused by all of these three types of irregularities.

Gaussian jitter

A separate dataset was collected to detect and measure the amplitude of Gaussian jitter in the data. The set contains 250 measurements for each of the included 52 oscillators from device 0. Each RO measurement was only 1M CPU ticks long to allow more measurements be taken in shorter time. To eliminate the effect of the CPU tick variation, we used frequency instead of raw oscillation counts in these calculations.

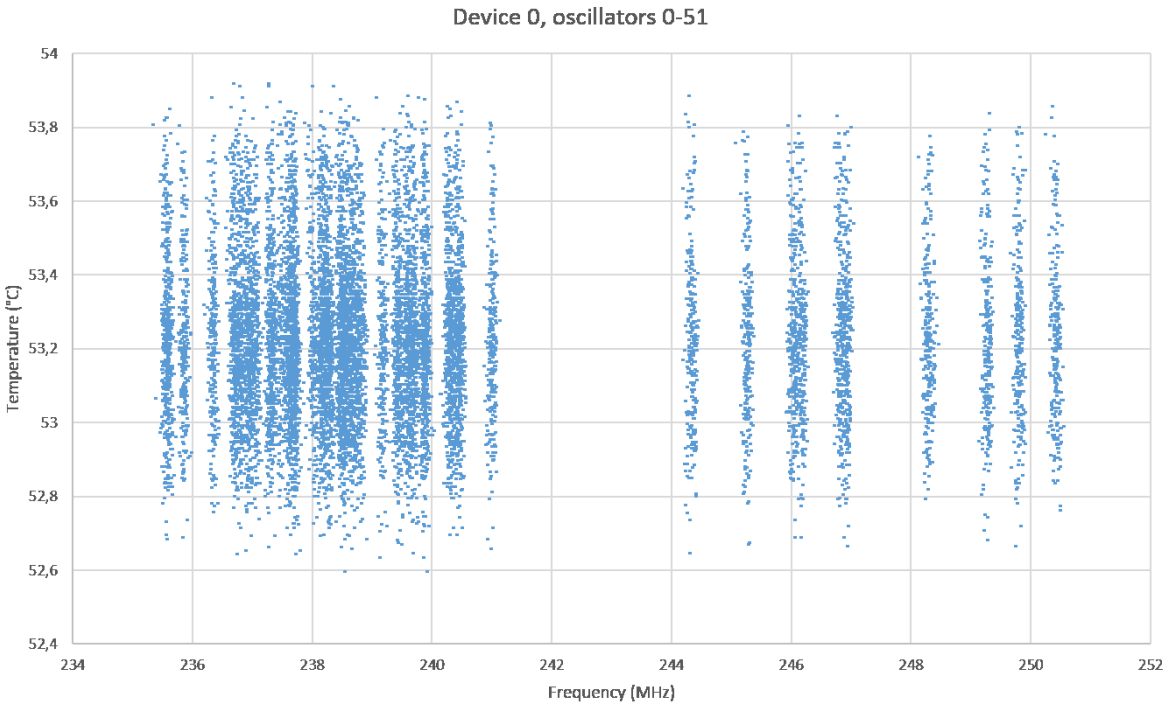


Figure 28: *A plot of measurement temperatures and frequency from the jitter dataset. Measurements from a single oscillator form a single "stripe" pattern.*

Stable temperature was maintained during the dataset measurements; variation between extremes is 1.32°C. As can be seen from figure 28, there is significant variation that is not explained by temperature. The effect of temperature would appear in figure 28 as a counterclockwise tilt of the "stripes". No such tilt is discernible when compared to the stripe widths; as the effect of run length variation has been normalized away, we can thus determine the frequency variations in this dataset to be predominantly caused by random Gaussian jitter. The information in table 9 and figure 29 can be used for optimizing filtering thresholds in later iterations.

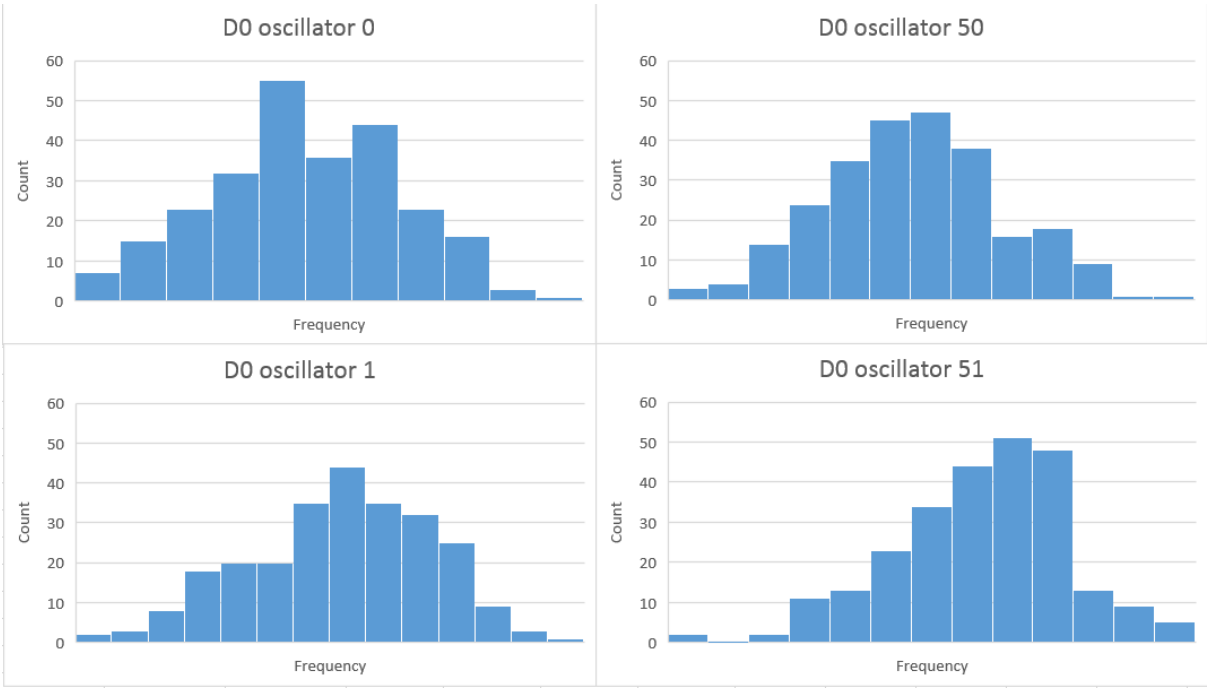


Figure 29: *Frequency distributions of some oscillators from the jitter dataset. Bin width is 0.02 MHz.*

	RO #0	RO #1	RO #50	RO #51
Max frequency	239.74 MHz	244.44 MHz	237.04 MHz	236.43 MHz
Min frequency	239.53 MHz	244.18 MHz	236.78 MHz	236.17 MHz
Variation	0.210 MHz	0.263 MHz	0.257 MHz	0.252 MHz
Sample standard deviation	0.041	0.051	0.044	0.043

Table 9: *Attributes of the selected RO distributions.*

Fulfillment of requirements

R1: Adequate entropy with no fundamental weaknesses
With the filtered naive selection method we have achieved an estimated upper bound of 253 bits of entropy. This is much more than 128 bits and very close to the ideal 256 bits. We consider it enough for a single very strong challenge-response pair, and this fulfills our requirement.

We are basing our calculations on the use of a single CRP using all the reliable RO pairs. There are security benefits in having many possible CRPs, most importantly the possibility to use a different challenge for each PUF evaluation. However, since we are on a SoC, we can use the PS to increase the effective number of CRPs by e.g. hashing the output in the PS immediately after acquiring a response bit, compensating for the reduction in the number of CRPs. We will discuss the possibilities of gaining more CRPs by using post-processing in the discussion section.

As a RO-PUF that produces only independent bits (no oscillator contributes to more than one bit), we avoid many fundamental risks. While our PUF deployed on a general-purpose FPGA is not proofed against physical attacks, any other design will be comparably vulnerable. Work remains in mitigating physical frequency measurement attacks.

R2: Maximum reliability and longevity within the expected operating environment
With the filtered naive selection method, accounting for temperature variation, we have achieved 0% intra-distance for the selected temperature range. We thus consider the maximum reliability requirement fulfilled, within these environmental conditions. The operating temperature range can be altered by using a dataset with larger temperature variations to perform the filtering in the enrolling phase.

We assume that aging deterioration is similar in nature to temperature variation, the effect being always of the same sign but different amplitude in different RO units. Threshold-based filtering should thus effectively mitigate both. It is naturally not possible to measure the real effect of aging in a short-term thesis project. Literature has usually ignored the effects of aging, with the exception of Rahman et al. (2014) who designed an aging-resistant “ARO-PUF”, but even they used simulated aging to evaluate their design.

While the work currently done does not decisively solve the aging issue, various methods could easily be developed to detect and control aging deterioration. The PUF could be enrolled using several overlapping CRPs with different filtering thresholds; if the lenient filtering fails, there would be more stringently filtered, more reliable CRPs to fall back to. Another option would be to enroll a variety of smaller CRPs that use only some regions of the chip, to allow working around deteriorated oscillators. Thus the PUF could be kept in use until the device can be replaced or re-enrolled, at the cost of some response entropy.

R3: Reasonable efficiency
According to the best estimate possible based on our data, our entropy density is about 0.5. We consider that efficiency reasonable, but far from perfect. However, the small sample size reduces our ability to generalize this result.

R4: Scalability and portability of design
The PUF is implemented using only basic configurable elements that should be available on any FPGA. It should be possible to adapt this design to some other device without altering the operating principles of the PUF. Deploying the PUF on a new device is only a matter of deploying the bitstream and performing the enrolling procedure. The number of oscillators can be increased without fundamentally altering the structure of the design. We consider this requirement fulfilled.

R5: Understanding the sources of error
We have measured and evaluated various sources of error, providing us with valuable understanding of their effects allowing us to concentrate further efforts on mitigating the ones that matter.

Voltage changes and aging were not evaluated in this study due to resource constraints, however, as these manifest as progressive frequency changes, they can be mitigated using adaptive CRPs or more aggressive filtering.

When estimating the entropy of naive RO pairings we have revealed significant bit-specific bias in our PUF. Since all of the devices were programmed using the same bitstream, they have identical placing and routing. As each unit is contained in a single CLB, each of which has its own routing matrix, the units were assumed to have identical routing of signals within each unit. Closer examination of the implemented routing later proved this to not necessarily be the case in all units. To prevent the implementation from placing non-identical routing, the routes within an oscillator unit should be specified manually. Since manually placing and routing at least a thousand oscillators is not a meaningful task, mechanisms like Fixed Routing constraints should be investigated to solve the problem (Xilinx 2014, 2015). Redoing the routing and placement would allow us to learn the exact effect, and evaluating a larger population of devices would lead to a more realistic estimate of this bias.

6 Discussion

Relevance and applicability of results

We have demonstrated that it is feasible to design and implement an adequate PUF even with limited resources. The results achieved are good enough to fulfill the original objective. We have a PUF that can produce approximately 256 reliable, physically unclonable bits of entropy.

There is obviously room for improvement. While our PUF can be declared good enough for our starting use-case, its performance is still short of the RO-PUF state-of-the-art in the literature (e.g. Maes et al. 2012; Yin & Qu 2010). Maes (2013: 95-98) achieved 1.53% intra-distance and 49.60% inter-distance on a 2048-bit RO-PUF using a simple pairwise comparison scheme. Using the Lehmer-Gray sequence encoding, the same PUF produced 12544 bits with 3.56% intra-distance and 46.86% inter-distance. Error-correcting post-processing (as in Maes et al. 2012) might allow us to tolerate smaller filter thresholds, or use no filtering at all. However, some forms of error correction might make us accept near-misses as correct, possibly lowering the effective entropy.

The central flaw of our design is predictability between devices, and a major reason for that is believed to be non-uniform routing within CLBs. Also, a form of bias that we have not evaluated is the effect of exact position on the chip on the oscillator frequency. Given the low average inter-distances, we might find RO speed bias based on fine-grained physical position or orientation. This would require manual or tightly constrained placing, in addition to routing. Maes (2013: 95-98) performs his measurements on 16 oscillators located in same position in different blocks at once, maybe making it more likely to pick a similarly routed and placed oscillator. Yin & Qu (2010) routed and placed their oscillators manually to achieve identical layout.

It is clear that our sample of 5 devices is too small for drawing statistically robust conclusions about the quality of our PUF. In particular, using more devices would make the entropy estimates more accurate and likely higher (as the number of bits with an expected value of 0 or 1 diminishes), while exposing worse worst-case distance measures. With a bigger sample we could also perform hypothesis testing on our claim of having a working PUF. Even with these shortcomings, we have a strong indication that our PUF has potential for future development.

Currently the most fundamental threat to our RO-PUF is considered to be the Lohrke et al. (2016) laser voltage measurement attack, which requires running each oscillator of the PUF in a laboratory with specialized equipment. In addition to taking the device to a laboratory, this either requires the original bitstream, or the attacker's version developed with detailed knowledge of how the original was built, to achieve the same frequency results. The attack presented in the paper was performed on a bitstream developed by the attackers. Having a non-optimal implementation could be formed into an another line of defense; With an imperfect implementation, it would be harder for an attacker to emulate the biased behavior of the original. While everything on the device, including encryption keys, is ultimately susceptible to physical attacks, measures like encrypting the bitstream can be used to place additional obstacles in the way of revealing design details to the attacker.

It should be noted that our entropy measures are estimated with harsh preconditions: in addition to the full-knowledge assumption, we are not reusing any oscillators in other CRPs, meaning that we can assume our bits to be independent and uncorrelated. Suh & Devadas (2007) originally estimated their entropy of a RO-PUF using the assumption that changing the order of RO evaluations in the challenge constitutes a new CRP. We consider a different ordering to count as a separate CRP only if the raw response is immediately obfuscated by hashing; this is discussed below in more detail.

Improving the simple CRP authentication scheme

While the raw response should be hashed to avoid revealing our CRPs, we can also hash each response bit as they come. Hashing in general will obfuscate the response by hiding the raw information about the relative speeds of the oscillators and making the order of bit evaluations matter; hashing in parts has the additional benefit of reducing the time any part of the response is kept as plaintext. Ideally, the hash function should be implemented in the FPGA, as in Maes et al (2012). They use a hash function as an entropy accumulator, where each partial response from the PUF is fed.

Example pseudocode:

```
ByteArray processed_response = []
For(pair in challenge)
      Bit bit = evaluate(pair)
      processed_response = Hash(append(processed_response, bit))
Return processed_response
```

Different ordering of the same pairs will now produce completely different output. Without hashing, the position of a pair evaluation in the CRP would not matter, allowing us only *n/2* ways of producing independent bits. Using response hashing and so not revealing the raw bitstring, we can reuse pairs in multiple CRPs, which will increase the number of potential pairs from *n/2* to *n(n-1)/2*. What is more, we can flip the positions within each pair, producing two versions of each pair, for *n(n-1)* different post-hash outputs. For our 1000 RO design this means 999 000 different CRPs up from the original single CRP. Taking this further, we can include non-secret salt values to increase the number of possible post-processed responses and use a standardized, slow function in the hash calculation, resulting in additional work. Performing this kind of post-processing will make the processed responses very unpredictable and potentially expendable, while also increasing the cost of brute-forcing the raw response from the processed response.

For the production version, different bitstreams could be used for enrolling and deployment. In the deployment bitstream the entire response processing could be performed in the PL, to make it impossible for malicious software on the PS to read raw responses and thus infer PUF behavior details.

A protocol could be devised in which the previous hashed response is also used as the next challenge, stored either in PL registers, or in persistent flash which would survive power cycling. This would leave proof of unauthorized evaluations by malicious software, but would not defend against theft and physical analysis. A related concept can be found in trusted platform modules (TPMs) that have Platform Configuration Registers (PCR). The PCRs

contain hashes of the platform software state that are kept updated by hashing the new state together with the previous hash (Abera et al. 2016).

Defending against LVP and FIB physical attacks on general-purpose FPGAs is hard, as any secrets on the FPGA will be vulnerable. Using tamper-resistant hardware or cases might be used with the PUF to provide defense in depth in high-risk deployments, but it still remains an open question. A specific solution might be to overlap several oscillators in the same CLB or physical area, or route random oscillating signals around the chip to distract readings.

PKI-based device authentication

Taken as-is, the original Suh & Devadas (2007) challenge-response authentication scheme has shortcomings. In addition to unobfuscated responses being transmitted over the wire, the stored CRP table presents a single point of failure, making the entire system brittle as a mechanism will be required to access the correct CRPs from the software authenticating the device. For this reason, the CRP table must be readable by internet-facing software. Various authentication protocols replacing the simple challenge-response protocol have been proposed in literature, including the BPV generator based protocol presented in Wallrabenstein (2015) and the PKI system in Afghah et al (2017). In figure 30 we present a simple example of how the plain challenge-response protocol can be replaced with a public-key infrastructure (PKI) based system.

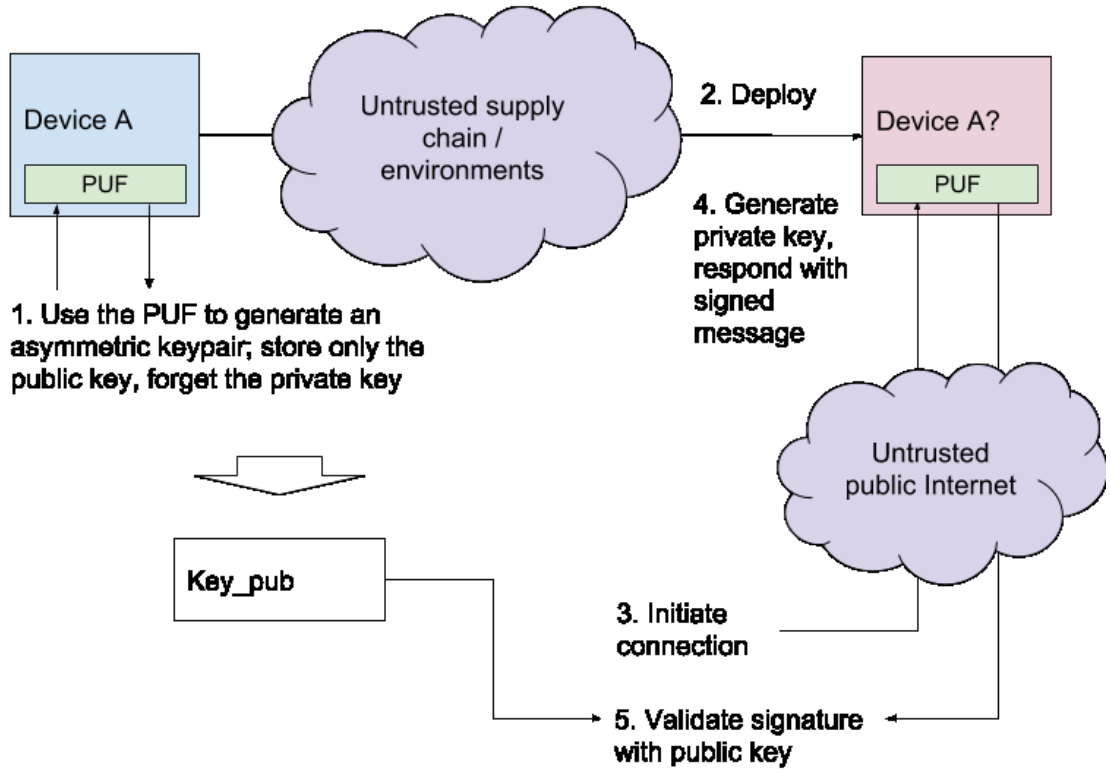


Figure 30: *a simple PUF-backed PKI device authentication scheme.*

By using the PUF to generate an asymmetric keypair and then storing only the public key, we can avoid the trouble of protecting a centralized CRP database. With the help of a carefully selected key-derivation function, the device will use the PUF-backed unclonable

randomness to regenerate the private key on the device every time it is needed. The public key can be used to verify the authenticity of messages sent from the device, while the private key never leaves the device, and is kept in memory only when actively used.

Most importantly, compromising a private key would compromise only that device, and not the entire fleet of deployed devices, as breaching a centralized database would. Symmetric cryptography could also be used, but would not remove the need for a centralized database. The private keys now become sensitive data, but as they are never transmitted or stored for longer than needed, they are protected to a reasonable degree.

PUF-based protocols can be built on to provide message encryption and other guarantees, e.g. timestamps and nonces to prevent replay attacks by a man-in-the-middle.

Lessons learned
It is determined to be possible and feasible to construct a good enough FPGA PUF even with constrained resources, a small target device and no previous hardware design experience. Work nevertheless remains if state-of-the-art results are to be achieved. Further, a larger set of devices needs to be evaluated before this or any other PUF implementation can be trusted for real-world use.

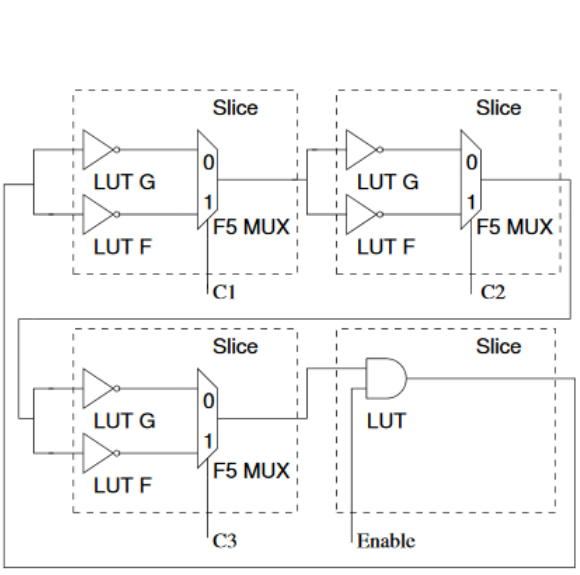
7 Related Work

In this section, we describe and compare three RO-PUF designs based on the original of Suh & Devadas (2007) that provide various improvements.

RO-PUF designs

Configurable RO-PUF

The main contributions of Maiti & Schaumont's (2009, 2010) Configurable RO-PUF ("CRO-PUF") design are considering systemic chip variation, the addition of an assisting CPU to perform reliable enrolling by analysing the raw oscillation data, and an increase in bit generation efficiency by making challenges configure RO internal routing. Using their design (figure 31), they can produce 8 reliable response bits using 2 ROs of 4 Slices each, thus producing *n* bits using *n* Slices. A straightforward pairwise RO comparison, where one oscillator takes 2 Slices (like in our design) needs 4 Slices to produce 1 bit, for a ratio of *n/4*. Thus, their design seems to provide a fourfold increase in resource efficiency, while also providing best possible reliability that can be achieved by comparing the paths within each oscillator pair. However, those 8 bits will not be independent of each other, as all of them will partly depend on the same physical variations (Xin et al. 2011). They claim about 40-45 % inter-distance, and 0% intra-distance within a 40-degree temperature range. Entropy yield is not discussed, and cannot be estimated using our previous entropy formulae due to dependent bits.



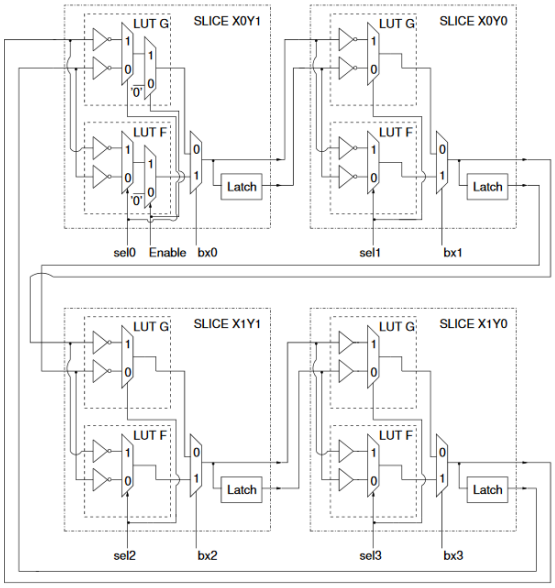


Figure 31 (left): CRO-PUF (Maiti and Schaumont (2009); figure from Xin et al. (2011)).
Figure 32 (right): Improved CRO-PUF (Xin et al. 2011)

Improved CRO-PUF

The unnamed design of Xin et al. (2011) is explicitly stated to improve on top of Maiti and Schaumont's CRO-PUF. They control the internal routing of oscillators in greater detail; see figure 32. Each oscillator now has 256 possible internal configurations. This way, the same oscillator pair can be used for an even larger number of comparisons. Their idea is to use the same configuration (challenge) in both oscillators being compared. Figure 33 presents

results from comparing the two ROs in all the 256 possible states. It can also be seen that some changes cause large changes in frequency, highlighting the fact that it would not bring benefits to configure the pair differently; the result would be predictable, and not bring much benefit. They claim about 1% intra-distance and about 40% inter-distance. Entropy is not discussed in their paper.

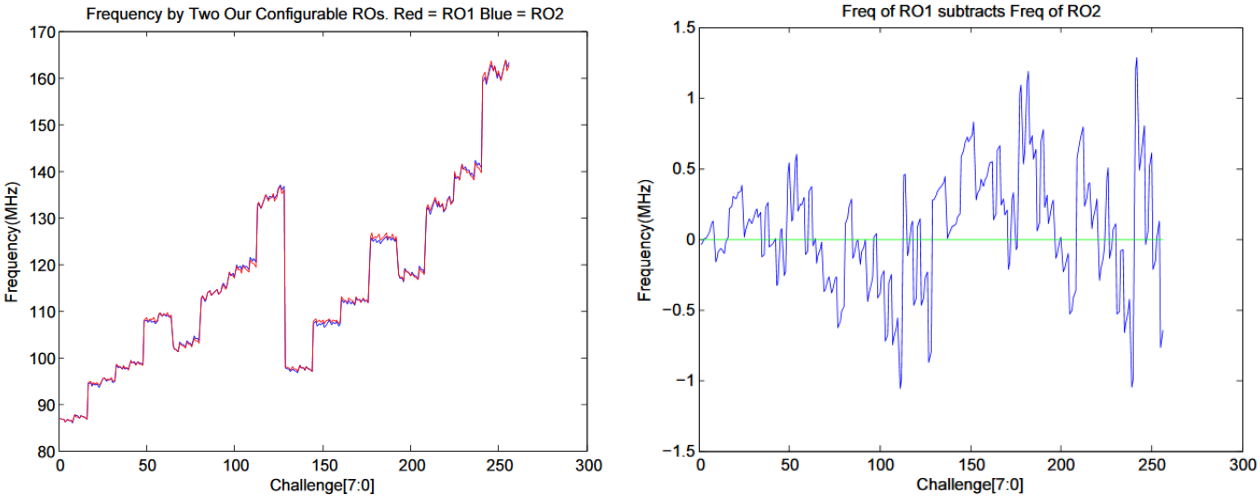


Figure 33: *Changing the configurable internal routing causes minuscule changes in the relative frequencies of a pair of ROs (Xin et al. 2011). Absolute frequency displayed on the left, frequency difference on the right.*

PUFKY

Perhaps the most advanced PUF-based solution published in the literature is the PUFKY of Maes et al (2012), some aspects of which have already been discussed in this thesis. Based on a sequence encoding idea like LISA (Yin & Qu 2010), it encodes the subsequence orders in Gray code to reduce the effect of changing order on the sequence response bitstring. Then it performs error-correcting postprocessing based on repetition code and BCH code. The corrected results are fed into a hash function. The error correction uses public syndromes that do not reveal information about the data to be corrected (Maes et al. 2012).

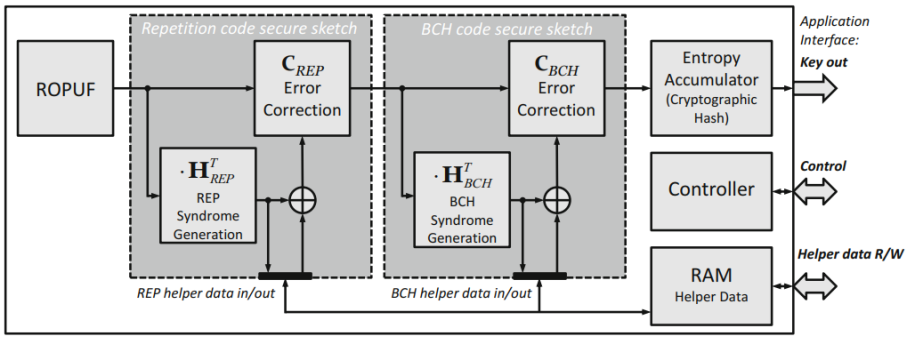


Figure 34: *Block design of PUFKY, showing the error correction blocks (Maes et al. 2012).*

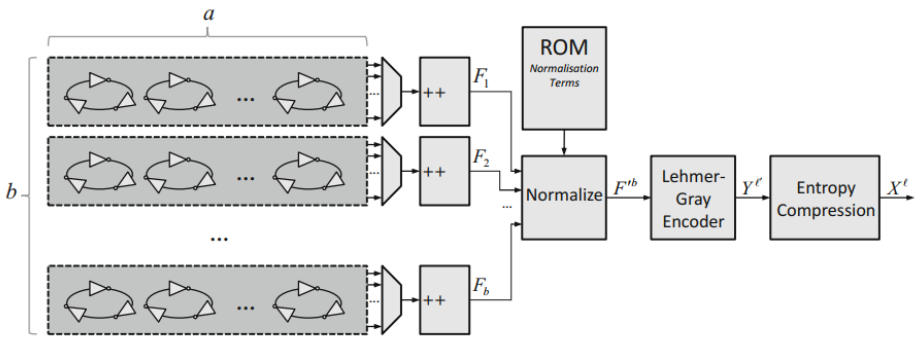


Figure 35: *Block design of the RO-PUF used in PUFKY (Maes et al. 2012). The challenge selects the same oscillator from each block to feed that block's counter. "Entropy Compression" consists of a XOR operation.*

Figure 34 contains a block-level schematic of PUFKY, while figure 35 expands the "ROPUF" block. While PUFKY does compare multiple oscillators, it does not contain any sequencing algorithm. Rather, the same fixed oscillators are always compared and reliability against bit flips is provided by the postprocessing. They claim 2.0% intra-distance and 48.4% inter-distance, and pay much attention to entropy generation. 90% entropy density is claimed after post-processing without entropy compression. With compression, the density can be further increased, but at the cost of a greater error rate (intra-distance).

The distance metrics reported by these related designs have been summarised in Table 10. While our intra-distance is not bad even when not filtered, our inter-distance is lagging behind, giving us reason to suspect fine-grained positional bias, possibly caused by routing asymmetries.

	Avg. intra-distance	Avg. inter-distance	Entropy density
CRO-PUF	**0%**	41.3%-45.5%	N/A
Improved CRO-PUF	1%	40%	N/A
PUFKY	2.0%	**48.4%**	**90 %**
Our PUF, naive	2.0 %	34.8 %	59 %
Our PUF, filtered	**0.0%**	32.6 %	51 %

Table 10: *Comparison of our results to those reported by related work. Intra-distances do not include extreme temperature tests. Values closest to the ideal are emphasized.*

References

Abera, T., Asokan, N., Davi, L., Koushanfar, F., Paverd, A., Sadeghi, A. R., & Tsudik, G. (2016). Things, trouble, trust: on building trust in IoT systems. In *Proceedings of the 53rd Annual Design Automation Conference* (p. 121). ACM.

Afghah, F., Cambou, B., Abedini, M., & Zeadally, S. (2017). A ReRAM Physically Unclonable Function (ReRAM PUF)-based Approach to Enhance Authentication Security in Software Defined Wireless Networ*ks. International Journal of Wireless Information Networks, 1-13.*

Anderson, J. H. (2010). A PUF design for secure FPGA-based embedded systems. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference* (pp. 1-6). IEEE Press.

Avnet (n.d.) *MiniZed*. Retrieved 2018-05-20 from http://zedboard.org/product/minized

Aysu, A., Gulcan, E., Moriyama, D., Schaumont, P., & Yung, M. (2015). End-to-end design of a PUF-based privacy preserving authentication protocol. In *International Workshop on Cryptographic Hardware and Embedded Systems* (pp. 556-576). Springer, Berlin, Heidelberg.

Boyko, V., Peinado, M., & Venkatesan, R. (1998). Speeding up discrete log and factoring based schemes via precomputations. In *International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 221-235). Springer, Berlin, Heidelberg.

Gassend, B., Clarke, D., Van Dijk, M., & Devadas, S. (2002). Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security* (pp. 148-160). ACM.

Gu, C., & O'Neill, M. (2015). Ultra-compact and robust FPGA-based PUF identification generator. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on* (pp. 934-937). IEEE.

Helfmeier, C., Boit, C., Nedospasov, D., & Seifert, J. P. (2013). Cloning physically unclonable functions. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on* (pp. 1-6). IEEE.

Herder, C., Yu, M. D., Koushanfar, F., & Devadas, S. (2014). Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, *102*(8), 1126-1141.

IETF (2010). *Trust anchor format* (RFC 5914). Retrieved 2018-05-20 from https://tools.ietf.org/html/rfc5914

IETF (2016). *DNSSEC Trust Anchor Publication for the Root Zone* (RFC 7958). Retrieved 2018-05-21 from https://tools.ietf.org/html/rfc7958

Kumar, S. S., Guajardo, J., Maes, R., Schrijen, G. J., & Tuyls, P. (2008). The butterfly PUF protecting IP on every FPGA. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on* (pp. 67-70). IEEE.

Lim, D., Lee, J. W., Gassend, B., Suh, G. E., Van Dijk, M., & Devadas, S. (2005). Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *13*(10), 1200-1205.

Lohrke, H., Tajik, S., Boit, C. and Seifert, J.P., (2016). No place to hide: Contactless probing of secret data on FPGAs. In *International Conference on Cryptographic Hardware and Embedded Systems* (pp. 147-167). Springer, Berlin, Heidelberg.

Maes, R. (2013). *Physically unclonable functions: Constructions, properties and applications*. Springer Science & Business Media.

Maes, R., & Verbauwhede, I. (2010). Physically unclonable functions: A study on the state of the art and future research directions. In *Towards Hardware-Intrinsic Security* (pp. 3-37). Springer Berlin Heidelberg.

Maes, R., Van Herrewege, A. and Verbauwhede, I. (2012). PUFKY: A fully functional PUF-based cryptographic key generator. In *International Workshop on Cryptographic Hardware and Embedded Systems* (pp. 302-319). Springer, Berlin, Heidelberg.

Maiti, A., & Schaumont, P. (2009). Improving the quality of a physical unclonable function using configurable ring oscillators. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on* (pp. 703-707). IEEE.

Maiti, A., & Schaumont, P. (2010). Improved ring oscillator PUF: an FPGA-friendly secure primitive. *Journal of cryptology*, *24*(2), 375-397.

Mazady, A., Rahman, M.T., Forte, D. and Anwar, M., (2015). Memristor PUF—a security primitive: Theory and experiment. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, *5*(2), pp.222-229.

Morozov, S., Maiti, A., & Schaumont, P. (2010). An analysis of delay based PUF implementations on FPGA. In *International Symposium on Applied Reconfigurable Computing* (pp. 382-387). Springer, Berlin, Heidelberg.

Pappu, R., Recht, B., Taylor, J., & Gershenfeld, N. (2002). Physical one-way functions. *Science*, *297*(5589), 2026-2030.

Rahman, T., Forte, D., Fahrny, J., & Tehranipoor, M. (2014). ARO-PUF: An aging-resistant ring oscillator PUF design. In *Proceedings of the conference on Design, Automation & Test in Europe* (p. 69). European Design and Automation Association.

Rührmair, U., Sölter, J., Sehnke, F., Xu, X., Mahmoud, A., Stoyanova, V., ... & Devadas, S. (2013). PUF modeling attacks on simulated and silicon data. *IEEE Transactions on Information Forensics and Security*, *8*(11), 1876-1891.

Schmidt, A. U., Kuntze, N., & Kasper, M. (2008). On the deployment of mobile trusted modules. In *Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE* (pp. 3169-3174). IEEE.

Schneier, B (1999). *Crypto-Gram, February 15th*. Retrieved 10.5.2018 from https://www.schneier.com/crypto-gram/archives/1999/0215.html.

Schneier, B (2013). *The NSA's Cryptographic Capabilities*. Retrieved 17.4.2018 from https://www.schneier.com/blog/archives/2013/09/the_nsas_crypto_1.html.

Suh, G. E., & Devadas, S. (2007). Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual design automation conference* (pp. 9-14). ACM.

Valtchanov, B., Aubert, A., Bernard, F., & Fischer, V. (2008). Modeling and observing the jitter in ring oscillators implemented in FPGAs. In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on* (pp. 1-6). IEEE.

Wallrabenstein, J. R. (2015). Implementing authentication systems based on physical unclonable functions. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (Vol. 1, pp. 790-796). IEEE.

Xilinx (2014). *Answer Record # 54683*. Retrieved 2018-03-02 from https://www.xilinx.com/support/answers/54683.html

Xilinx (2015). *Vivado: User Guide 903.* Retrieved 2018-03-02 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug903-vivado-using-constraints.pdf

Xin, X., Kaps, J. P., & Gaj, K. (2011). A configurable ring-oscillator-based PUF for xilinx FPGAs. In *Digital System Design (DSD), 2011 14th Euromicro Conference on* (pp. 651-657). IEEE.

Yin, C.E.D. and Qu, G. (2010). LISA: Maximizing RO PUF's secret extraction. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on* (pp. 100-105). IEEE.

Zhang, J. L., Qu, G., Lv, Y. Q., & Zhou, Q. (2014). A survey on silicon PUFs and recent advances in ring oscillator PUFs. *Journal of computer science and technology*, *29*(4), 664-678

Appendix

1. LISA

The LISA pseudocode was hard to discern. The first attempt did not work as expected, and so some edits were included. In particular, some counters were omitted and an error-correcting check was added at the end. Numbered lines refer to lines of pseudocode in Yin & Qu (2010).

lisa.py:

```
def LIS(sorted, n, f_threshold):
    # 1. create a stack ST_1, and push the first
    # ring oscillator sorted[1] to it
    stacks = []
    stack_0 = []
    stacks.append(stack_0)

    stacks[0].append(sorted[0])

    # 2. h <- 1
    # commented out as unused
    #h = 0

    # 3. for j <- 2 to n
    for j in range(1, n-1):

        # 4. top <- the top ring osc on stack ST_h;
        top = stacks[-1][-1]

        # 5. if ((sorted[j]. fmin - top. fmin > fth) && (sorted[j]. fmax - top. fmax > fth) )
        if(sorted[j]["fmin"] - top["fmin"] > f_threshold and
            sorted[j]["fmax"] - top["fmax"] > f_threshold):

                # 6. h++
                # commented out as unused
                #h = h+1

                # 7. new a stack ST_h (sic), push sorted[j] to it.
                stack_h = []
                stacks.append(stack_h)
                stacks[-1].append(sorted[j])

                # 8. sorted[j].PRE <- top
                #sorted[j]["PRE"] = top

        # 9. else
        else:

                # 10. find the stack ST_p with the largest index p that
                # has its top element's fmax smaller than sorted[j].fmax - fth
                # and fmin smaller than sorted[j].fmin - fth.
                p = False
                for i in range(0,len(stacks)):
                        if sorted[j]["fmin"] - stacks[i][-1]["fmin"] > f_threshold and
                            sorted[j]["fmax"] - stacks[i][-1]["fmax"] > f_threshold:
                                p = i

                # 11. if p ! = null
                if not p == False:
                        # 12. push sorted[j] to ST_p+1

                        stacks[p+1].append(sorted[j])
```

```
                    # 13. sorted[j].PRE <- top element of ST_p
                    #sorted[j]["PRE"] = stacks[p][-1]

            # 14. end if
    # 15. end if
# 16. end for

# 17. return sequence <sorted[j_1], sorted[j_2] ... sorted[j_h]> where sorted[j_h]
# is the top element of ST_h and sorted[j_q-1] = sorted[j_q].PRE, q from 2 to h

output = []

# We encountered occasional errors where sometimes the sequences would not be in order.
# Below is a stopgap solution.

prevelem = False
for stack in stacks:
    elem = stack[-1]
    # if we're about to hit an error, chop the sequence here
    if prevelem:
            if elem["fmin"] - prevelem["fmin"] <
               f_threshold or elem["fmax"] - prevelem["fmax"] < f_threshold:
                    print("ERROR: chopped the sequence")
                    return output
    prevelem = elem
    output.append(elem)

return output
```

2. Controlling program

This is the program that was run on the Zynq CPU, with obsolete parts and non-essential system monitor code removed.

controller.c:

```
#include "stdio.h"
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xtime_l.h"
#include <time.h>
#include <limits.h>
#include <inttypes.h>

#include "xsysmon.h"
#include "xstatus.h"

<verbose system monitor code retracted from here>

u32 reg1_address = XPAR_PUF_5_V1_0_0_BASEADDR;
u32 reg2_address = XPAR_PUF_5_V1_0_0_BASEADDR + 4;
u32 reg3_address = XPAR_PUF_5_V1_0_0_BASEADDR + 8;
u32 reg4_address = XPAR_PUF_5_V1_0_0_BASEADDR + 12;

// how many ROs the hardware has.
const u32 NUMBER_OF_OSCILLATORS = 1000;

static inline void turn_on_puf(u32 reg1, u32 reg2){

        // ORs the words with 100000[...]
        reg1 |= 1UL << 31;
        Xil_Out32(reg1_address, reg1);

        reg2 |= 1UL << 31;
        Xil_Out32(reg2_address, reg2);

}

static inline void turn_off_puf(u32 reg1, u32 reg2){

        // ANDs the words with 011111[...]
        reg1 &= ~(1UL << 31);
        Xil_Out32(reg1_address, reg1);

        reg2 &= ~(1UL << 31);
        Xil_Out32(reg2_address, reg2);

}

static inline void sleep_ticks(u32 ticks){

        // XTime is u64
        XTime start;
        XTime end;
        XTime_GetTime(&start);
        XTime_GetTime(&end);

        while(end - start < ticks)
        {
                XTime_GetTime(&end);
        }
}
```

```
void run_pair(u32* oscillator_1, u32* oscillator_2, u32* reg3_last_read, u32* reg4_last_read,
u32* reg3_increase, u32* reg4_increase, u32* cpu_ticks_target, u32* cpu_ticks_actual, int
debug){

    if(*oscillator_1 == *oscillator_2){
        printf("#ABORT; TRYING TO RUN SAME OSCILLATOR\n\r");
        return;
    }

    else if(*oscillator_1 >= NUMBER_OF_OSCILLATORS || *oscillator_2 >= NUMBER_OF_OSCILLATORS){
        printf("#ABORT; OSCILLATOR INDEX OUT OF RANGE (%lu)\n\r", NUMBER_OF_OSCILLATORS);
        return;
    }

    Xil_Out32(reg1_address, *oscillator_1);
    Xil_Out32(reg2_address, *oscillator_2);


    // "Global timer is clocked at half the CPU frequency" -xtime_l.h
    // ie. the number of real CPU ticks is double the number of these timer ticks.
    // cpu_ticks should ideally be an even number.
    u32 timer_ticks = (*cpu_ticks_target)/2;

    XTime start;
    XTime end;

    // this is the measurement loop; need as few distractions as possible here
    XTime_GetTime(&start);
    turn_on_puf(*oscillator_1, *oscillator_2);
    sleep_ticks(timer_ticks);
    turn_off_puf(*oscillator_1, *oscillator_2);
    XTime_GetTime(&end);

    *cpu_ticks_actual = (end-start)*2;

    u32 reg3_now_read = Xil_In32(reg3_address);
    u32 reg4_now_read = Xil_In32(reg4_address);

    //if smaller: overflow happened
    if(reg3_now_read < *reg3_last_read){
        *reg3_increase = reg3_now_read + (UINT32_MAX - *reg3_last_read);

    }else{
        *reg3_increase = reg3_now_read - *reg3_last_read;
    }
    if(reg4_now_read < *reg4_last_read){
        *reg4_increase = reg4_now_read + (UINT32_MAX - *reg4_last_read);

    }else{
        *reg4_increase = reg4_now_read - *reg4_last_read;
    }

    *reg3_last_read = reg3_now_read;
    *reg4_last_read = reg4_now_read;
}

int main()
{
    init_platform();

    int keeprunning = 1;
    u8 input;

    int number_input_buffer_index = 0;
    u8 number_input_buffer[32];

    int interpret_as_binary = 0;
```

```
    int multirun = 0;
    int which_device = -1;

    u8 which_oscillator = 0;
    u32 oscillator_1 = 0;
    u32 oscillator_2 = 0;
    u32 reg3_last_read = 0;
    u32 reg4_last_read = 0;

    u32 cpu_ticks_target = 10000000; // default: 10M

    int status = conf_sysmon();
    if (status != XST_SUCCESS) {
         print("SYSMON CONF FAILED; QUIT\n\r");
        cleanup_platform();
         return XST_FAILURE;
    }

        print("#Listening...\n\r");
        while(keeprunning)
        {
         // read the serial connection
         input = XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR);

         printf("#Input char was %c\n\r", input);

         // covers (most) non-printable ascii chars; we ignore them
         if(input < 32){
                continue;
         }

        // set device id
        if(input == 'C'){
                which_device = 0;
                printf("#INFO: DEVICE ID SET TO %d \n\r",which_device);
        }
        if(input == 'V'){
                which_device = 1;
                printf("#INFO: DEVICE ID SET TO %d \n\r",which_device);
        }
        if(input == 'B'){
                which_device = 2;
                printf("#INFO: DEVICE ID SET TO %d \n\r",which_device);
        }
        if(input == 'N'){
                which_device = 3;
                printf("#INFO: DEVICE ID SET TO %d \n\r",which_device);
        }
        if(input == 'M'){
                 which_device = 4;
                 printf("#INFO: DEVICE ID SET TO %d \n\r",which_device);
        }

        // toggle varying run orders
        if(input == 'I'){
                if(multirun){multirun = 0; } else { multirun = 1; }
                 printf("#INFO: multirun SET TO %d \n\r",multirun);
        }

        // automated full run

        if(input == 'G')
        {
                printf("#INFO: START FULL RUN\n\r");
                printf("#INFO: CPU FREQUENCY IS %d HZ\n\r",
XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ);
```

```
		if(which_device == -1){
			printf("#ABORT: NO DEVICE NUMBER SET (use C-M)\n\r"); continue;
		}

		// ro indices
		oscillator_1 = 0;
		oscillator_2 = 1;

		u32 counter = 0;
		u32 counter_exclusive_upper_limit = NUMBER_OF_OSCILLATORS/2;

		if(multirun){
			printf("#INFO: DOING A MULTI-RUN\n\r");
		}

		u32 reg3_increase = 0;
		u32 reg4_increase = 0;

		u32 cpu_ticks_actual;
		double real_time_delta;
		int debug = 0;

		float temp1, vcc_int1, vcc_aux1, temp2, vcc_int2, vcc_aux2;

	printf("DEVICE; COUNTER; POSITION; RO_INDEX; INCREASE; CPU_TICKS; TIME_DELTA;
TEMPERATURE; VCC_INT; VCC_AUX;\n\r");

	int keep_running = 1;
	while(keep_running)
		{
			poll_sysmon(&temp1, &vcc_int1, &vcc_aux1);
			run_pair(&oscillator_1, &oscillator_2, &reg3_last_read,
				&reg4_last_read, &reg3_increase, &reg4_increase,
				&cpu_ticks_target, &cpu_ticks_actual, debug);

			poll_sysmon(&temp2, &vcc_int2, &vcc_aux2);

			// sampling does averaging already but we do once more
			float temp_avg = (temp1+temp2)/2;
			float vcc_int_avg = (vcc_int1+vcc_int2)/2;
			float vcc_aux_avg = (vcc_aux1+vcc_aux2)/2;

			real_time_delta   =
(1.0*cpu_ticks_actual)/XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ;

			printf("%d; %lu; %u; %lu; %lu; %lu; %f; %f; %f; %f;\n\r",
				which_device, counter, 1, oscillator_1, reg3_increase,
				cpu_ticks_actual, real_time_delta, temp_avg,
				vcc_int_avg, vcc_aux_avg);

			printf("%d; %lu; %u; %lu; %lu; %lu; %f; %f; %f; %f;\n\r",
				which_device, counter, 2, oscillator_2,
				reg4_increase, cpu_ticks_actual, real_time_delta,
				temp_avg, vcc_int_avg, vcc_aux_avg);
			counter++;

			if(multirun)
			{
				// 1st->2nd
				if(counter == counter_exclusive_upper_limit)
				{
					oscillator_1 = NUMBER_OF_OSCILLATORS-1;
					oscillator_2 = NUMBER_OF_OSCILLATORS-2;
					printf("#Sub run done\n\r");
				}
				// 2nd->3rd
```

```
                        else if(counter == 2*counter_exclusive_upper_limit)
                        {
                                oscillator_1 = 1;
                                oscillator_2 = 0;
                                printf("#Sub run done\n\r");
                        }
                        // 3rd->4th
                        else if(counter == 3*counter_exclusive_upper_limit)
                        {
                                oscillator_1 = NUMBER_OF_OSCILLATORS-2;
                                oscillator_2 = NUMBER_OF_OSCILLATORS-1;
                                printf("#Sub run done\n\r");
                        }
                        // 4th->stop
                        else if(counter == 4*counter_exclusive_upper_limit)
                        {
                                keep_running = 0;
                        }
                        // 4th
                        else if(counter >= 3*counter_exclusive_upper_limit)
                        {
                                oscillator_1 = oscillator_1 - 2;
                                oscillator_2 = oscillator_2 - 2;
                        }
                        // 3rd
                        else if(counter >= 2*counter_exclusive_upper_limit)
                        {
                                oscillator_1 = oscillator_1 + 2;
                                oscillator_2 = oscillator_2 + 2;
                        }
                        // 2nd
                        else if(counter >= counter_exclusive_upper_limit)
                        {
                                oscillator_1 = oscillator_1 - 2;
                                oscillator_2 = oscillator_2 - 2;
                        }
                        // 1st
                        else
                        {
                                oscillator_1 = oscillator_1 + 2;
                                oscillator_2 = oscillator_2 + 2;
                        }
                }
                else
                {
                        oscillator_1 = oscillator_1 + 2;
                        oscillator_2 = oscillator_2 + 2;

                        if(counter >= counter_exclusive_upper_limit)
                        {
                                keep_running = 0;
                        }
                }
        }
        printf("#INFO: Done\n\r");
    }
    if(input == 'Q')
    {
             printf("Quit!");
             keeprunning = 0;
    }
    } // end of the main while loop
    cleanup_platform();
    return 0;
}
```