ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/228615879

Intrinsic PUFs from flip-flops on reconfigurable devices

ARTICLE · JANUARY 2008

CITATIONS
64

READS
239

3 AUTHORS, INCLUDING:

Roel Maes
Intrinsic-ID
30 PUBLICATIONS **678** CITATIONS
SEE PROFILE

Pim Tuyls
115 PUBLICATIONS **3,290** CITATIONS
SEE PROFILE

Available from: Roel Maes
Retrieved on: 11 April 2016

Intrinsic PUFs from Flip-flops on Reconfigurable Devices

Roel Maes, Pim Tuyls, Ingrid Verbauwhede

Katholieke Universiteit Leuven: ESAT-COSIC
{roel.maes, pim.tuyls, ingrid.verbauwhede}@esat.kuleuven.be

Abstract. Intrinsic Physical Unclonable Functions or PUFs have been introduced as a highly secure method to extract digital identifiers and keys from integrated circuits. In the setting of reconfigurable devices, like FPGAs, they can be used to protect the IP contained in the design in a cost-effective and tamper-evident way. In this work, a new type of PUF construction for reconfigurable devices is introduced, based on flip-flop powerup values. Given experimental data shows that flip-flop PUFs behave similarly to earlier described SRAM PUFs. However, they have some additional advantages with regard to the use in reconfigurable devices.

Keywords: Physical Unclonable Functions, Reconfigurable Devices, Intellectual Property Protection

1 Introduction

The NRE costs for a new chip design have increased drastically and have to be compensated for by selling many items of an innovative product at a fair price. Cloning of integrated circuits has therefore become a profitable bussiness for counterfeiters. If they get their hands on the chip's plans, *e.g.* through reverse-engineering, industrial espionage or even outsourcing, they can produce and sell the product below the market price and possibly gain a lot of money at the expense of the original developper. The implementation of countermeasures to avoid cloning of ICs is therefore a topic of ongoing research [1–3]. Moreover, the cloner can even add malicious content like a backdoor or a trojan to the original design, making the use of counterfeited hardware a potential security risk [4, 5]. It is therefore nowadays a major concern for many circuit developpers to protect their intellectual property from being copied or tampered with by external parties.

For reconfigurable devices like Field-Programmable Gate Arrays or FPGAs, cloning is an even larger problem because the act of copying or tampering with the design is particulary easy. Since many types of FPGAs are volatile, their

This work was supported in part by IBBT (Flemish Government) and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy). The first author's research is funded by IWT-Vlaanderen under grant number 71369.

configuration has to be reloaded on every powerup. The used design, contained in a so called *bitfile* is therefore often stored on a non-volatile memory on the same PCB as the FPGA. An adversary who has physical access to the device can eavesdrop on the bus loading the configuration onto the FPGA. The bitfile can easily be copied and loaded onto other FPGAs, which is equivalent to cloning the original design. Moreover, since the bitfile is in general not encrypted, the adversary can attempt to reverse-engineer it, in order to obtain the netlist [6]. He then has the possibility to make malicious modifications.

FPGA manufacturers address these problems by providing the possibility to use encrypted bitfiles. A unique private key has to be stored on-chip to be able to decrypt the bitfile. This brings us back to the problem that many FPGAs do not contain non-volatile memory to store such a key. Non-volatile storage like EEPROM has to be added [7], or a small battery has to be included to keep a small amount of the FPGA's volatile memory always powered [8]. Both solutions however increase the size and/or the cost of the product. Even more, for sensitive designs, a permanently stored secret key might not be safe from determined attackers with sophisticated equipment [9].

Recent research results suggest using PUFs for protecting the intellectual property of reconfigurable devices like FPGAs [1–3, 10–13]. PUFs have the possibility to securily store secret data in a non-volatile way without the explicit need for non-volatile storage. They moreover provide additional security against sophisticated attackers that physically invade the chip to try to learn the secret key.

1.1 Intrinsic Physical Unclonable Functions

Physical Unclonable Functions or PUFs have been introduced as challenge-response entities that are undetachably embedded in a physical system [14], possibly a silicon integrated circuit [15]. The physical system is, explicitly or implicitly, produced in such a way that it contains uncontrollable random elements. An applied challenge reacts with these elements in a complex and unpredictable way to produce a certain random response. The presence of random elements, and the impossibility of controlling them at manufacturing time, makes each PUF unique and physically unclonable, hence it's name.

Because of the usefull practical applications and their low cost, there is much attention for PUFs contained in integrated circuits. A major advantage of this approach is the fact that ICs implicitly contain random elements introduced at manufacturing time [16, 17]. Due to random variations in the production process, no two identically produced chips will be physically exactly identical. In a regular circuit on a chip, there has to be some margin built in the behavior to cope with these process variations. For the concept of PUFs, it might however seem useful to construct circuits that measure the unique variations contained within each IC. Such circuits have been suggested based on accurate delay measurements of digital paths [18, 15, 19]. Further research demonstrated that even regular circuits like SRAM-cells have the possibility to behave as PUFs under certain conditions [1, 20]. This is particulary interesting since SRAM-cells are often already present on a chip, and using them as PUFs induces no extra cost. These types of physical unclonable functions have been labelled *Intrinsic PUFs* in [1].

1.2 Overview and Contributions

This work concentrates on Intrinsic PUFs on reconfigurable devices, specifically FPGAs. In Section 2 an overview of existing (Intrinsic) PUFs on reconfigurable devices is given. Further on, the Flip-flop PUF is introduced. Section 3 first describes the configuration process of a common FPGA and Section 4 demonstrates how some configuration commands can be creatively combined to readout flip-flop powerup values. Post-processing steps that ameliorate the response statistics and prepare them for use as cryptographic keys are proposed in Section 5. Section 6 discusses the practical use and this work is concluded in Section 7.

2 Related Work: PUFs on Reconfigurable Devices

Because of the particular need of IP-protection on reconfigurable devices, we will consider PUF implementations on FPGAs. However, because of the strong similarity with regular integrated circuits, the mentioned results on PUFs on FPGAs can be generalized for all silicon chips. We will briefly describe three types of PUFs that are available in literature and discuss the advantages and disadvantages with respect to reconfigurable devices.

2.1 Delay PUFs

The first PUFs embedded in silicon devices that were suggested are based on delay measurements of digital paths. Because of process variations, two identically designed digital paths on two ICs, or on two distinct places on the same IC, will not have exactly the same delay from their input to their output. Two methods have been proposed to measure the random delay variations for use as a physical unclonable function.

Arbiter PUF [18, 21]. A race condition is introduced by feeding two symmetrical paths into an *arbiter*. Two simultaneous pulses on the inputs of the paths will in general not arrive at the outputs simultaneously due to small random delay variations. The arbiter decides which of the two paths was the fastest. Due to the manufacturing variations, the results will be random for a given instantiation of the PUF. The symmetrical delay circuits are laid out in such a way that an applied challenge can choose a different routing of the paths, with a possibly different response.

Ring-oscillator PUF [15, 19]. A ring oscillator is constructed by feeding the inverted output of a delay path back to it's input. An edge detector and a digital counter are used to determine the frequency, and hence the period, of the oscillation. Again due to process variations, the measured delays will be random upto some extent and can be used as PUF responses. The exact route through the delay circuit can be choosen by setting the challenge.

Ring-oscillator PUFs as well as Arbiter PUFs have been successfully implemented on FPGAs [15, 21]. It is however clear from their description that they are not intrinsic to a regular chip. Specialized, possibly large dedicated circuits have to be placed on the IC to implement these PUFs.

2.2 SRAM PUFs

More recently the first intrinsic PUFs were proposed, based on SRAM-cells [1, 20]. Static Random Access Memory or SRAM is a very common type of volatile storage in integrated circuits. A typical SRAM-cell consists of six transistors in CMOS-technology. The actual storage element comprising four transistors implements two crosscoupled inverters. Two additional transistors are used for read/write-access. The crosscoupled storage element is bistable, meaning it has two stable states, and hence can store one bit of information by residing in one of both states. When the cell is powered on, it will quickly converge into one of both stable states and it was experimentally verified that most cells have an explicit preference of one state over the other. However, which power-up state a cell prefers is random and not known a priori. The reason for this is that every cell contains some amount of *mismatch* between the two halfs of the crosscoupled circuit, due to random deviations during manufacturing. This inbalance causes the cell to fall more easily into one state, but which of both states this is, is determined by the sign of the mismatch and not known in advance. The power-up state of an SRAM memory will hence be random for a given device and can be used as a PUF response.

Since SRAM is abundantly used in many digital devices, the feasibility of this type of PUF could be verified easily by reading out the initial memory contents of these devices. It was demonstrated that for most devices the power-up state is indeed random and often even close to a uniform distribution [1]. However, for FPGAs some problems were encountered. The majority of FPGAs are SRAM-based which means that the configuration of the FPGA's logic is stored in volatile SRAM. This ensures that FPGAs contain a lot of SRAM, but the random power-up values are problematic for their functioning. The random initial configuration of the FPGA can *e.g.* cause shorts in its reconfigurable circuitry and damage the device. This should be avoided at all cost, and therefore FPGA manufacturers make sure the entire SRAM configuration is immediately resetted at power-up (*e.g.* [22, p.293]). Apart from some rare cases where some parts of the SRAM are not resetted[1], SRAM PUFs cannot be used on FPGAs.

2.3 Butterfly PUFs [11]

Butterfly PUFs were introduced as a method to emulate SRAM-behavior on FPGAs where all SRAM is resetted on power-up. An SRAM-cell consisting of crosscoupled inverters is replaced by a so-called butterfly-cell consisting of cross-coupled latches. Latches are asynchronous memory elements that can be resetted to '0' or presetted to '1'. In most FPGAs, latches can also be configured to work as synchronous flip-flops and they are moderatly present, spread over the whole FPGA area. Because of the crosscoupling, the butterfly-cell also has two stable states. However, by resetting one of the latches and simultaneously presetting the other one, the cell can momentarily be brought into an unstable condition. The butterfly-cell will converge quickly back to one of both stable states, again with a certain preference depending on the manufacturing mismatch between the two halfs.

The Butterfly PUF construction has been demonstrated to posses a PUF-behavior equivalent to SRAM PUFs [11]. They have also been implemented on FPGAs. Great care should however be placed in the exact placing and routing of the butterfly-cells. This is because the preferred state does not only depend on the mismatch in the latches, but also on the possibly deterministic mismatch in the routing. It is also obvious that a Butterfly PUF is not entirely intrinsic. Latches are present on most FPGAs, but they must be used in dedicated and carefully placed circuits to act as a PUF.

2.4 Why yet another PUF on FPGA?

SRAM PUFs seem to have a clear advantage over other PUF types on reconfigurable devices, since they are truly intrinsic and do not use up any resources. Unfortunately, on most FPGAs, all SRAM is immediately and automatically resetted at power-up and can hence not be used anymore to serve as a PUF response. Delay PUFs and Butterfly PUFs on the other hand do not suffer from this initial resetting, but they are build using dedicated circuitry and occupy an amount of FPGA resources. It would be convenient to have a truly intrinsic PUF that can be used on any SRAM-based FPGA. The Flip-flop PUF introduced in this work fullfills both these conditions.

3 Preliminaries: The FPGA configuration process

Firstly, some information is provided about the way an FPGA is initialized and how a design can be programmed on it. The used methods and terminology originate from the Xilinx FPGAs on which the tests were performed.

[1] *E.g.* on certain Altera platforms, some dedicated SRAM blocks retain their powerup state.

3.1 Initialization steps.

Besides a large amount of configurable logic, an FPGA also contains a small configuration controller that regulates the initialization and configuration of the device. In Figure 1 a simplified overview of the initialization procedure of a typical Xilinx FPGA is given, omitting some details that are not important for this work. The consecutive steps are:

1. When power is applied to the FPGA, the supply voltage needs some short time to rise. The configuration controller halts until the supply voltage has reached an appropriate level.
2. When the required supply voltage is reached, the FPGA configuration controller immediately clears the whole SRAM configuration memory, including the portions that are used as regular SRAM in the reconfigurable logic. This erases any random values that could have been used in an SRAM PUF. Clearing the configuration memory is absolutely necessary to avoid the accidental creation of shorts in the reconfigurable logic. Moreover, it provides a fixed and known initial device state, which is convenient for designers and design tools.
3. The FPGA is now ready for configuration and waits until a bitfile is loaded through one of its configuration ports. The bitfile contains certain commands for the configuration controller and a large portion of configuration data to set the reconfigurable logic.
4. When the configuration has finished successfully, the configured design is ready to start. Among other things, the FPGA I/O-pins are activated and the design's clock signal is applied.

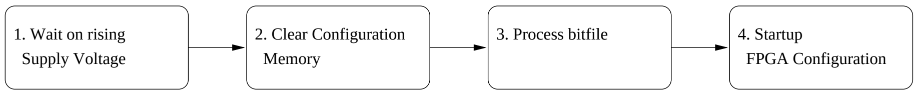
1. Wait on rising Supply Voltage
2. Clear Configuration Memory
3. Process bitfile
4. Startup FPGA Configuration

Fig. 1. Initialization steps of a Xilinx FPGA when powered up.

3.2 Configuration.

A lot of things happen in the actual configuration part of the FPGA initialization (step 3 in Figure 1). We focus on some of them that are important to this work.

The major part of the FPGA bitfile consists of configuration data. This data is written into the volatile configuration memory that fully controls all the settings of the reconfigurable logic to which it is connected. An illustration of the configuration memory on top of the reconfigurable logic can be seen in Figure 2. Besides configuration data, the bitfile also contains some additional commands for the configuration controller, *e.g.* to set some control options, read out some status registers or to specify an address in the configuration memory.

The state of the configuration memory completely determines all the settings of the reconfigurable logic. One of the settings that is contained in the configuration data is the initial value of all the flip-flops in the logic. For every reconfigurable flip-flop, there is a memory cell in the configuration memory that is set to the flip-flop's initial value during configuration. The initial value is however not immediately stored in the flip-flop itself! When all configuration data is loaded into the memory, the *global restore line* (`GRESTORE`) is asserted, (p)resetting all flip-flops to their initial value. This is done by issuing the `GRESTORE` command to the configuration controller at the end of the bitfile. The operation of the `GRESTORE` line is shown in Figure 2.

3.3 Readback.

Besides loading configuration data into memory, the configuration controller offers some other functionality. It can equivalently be used to readback the configuration memory at any point in time through a configuration port[2]. This can be usefull, *e.g.* to verify the validity of the loaded configuration or to detect Single-Event-Upsets (SEU). Additionally, the state of the flip-flops at any moment during operation can be captured and stored in the configuration memory, overwriting the initial value of the flip-flop. This is done by applying a pulse on the *global capture line* (`GCAPTURE`). One way of doing this is by issuing the `GCAPTURE` command to the configuration controller. The operation of the `GCAPTURE` line is also shown in Figure 2. If a readback is done after a capture, the current values of the flip-flops, rather than the initial values, can be inspected.

4 The Flip-flop PUF

The flip-flop PUF uses the powerup values of the flip-flops present on the FPGA in the same way as an SRAM PUF would use the powerup values of SRAM-cells. With the information provided in the previous section, it is possible to devise methods for retaining and reading out the power-up values of flip-flops in the FPGA.

4.1 Retaining the flip-flop power-up values.

A default bitfile, as produced by the Xilinx compilation tools, will always contain a `GRESTORE` command at the end. This will (p)reset all flip-flops to their initial value[3], erasing any random power-up values that could be used as a PUF response. However, it is possible to locate and remove the `GRESTORE` command from the bitfile, leaving the flip-flops in their random power-up state.

[2] On Xilinx FPGAs, possible configuration ports are *e.g.* the JTAG interface or the SelectMAP interface.

[3] If no initial value is specified in the design, the flip-flop will be resetted to '0'.

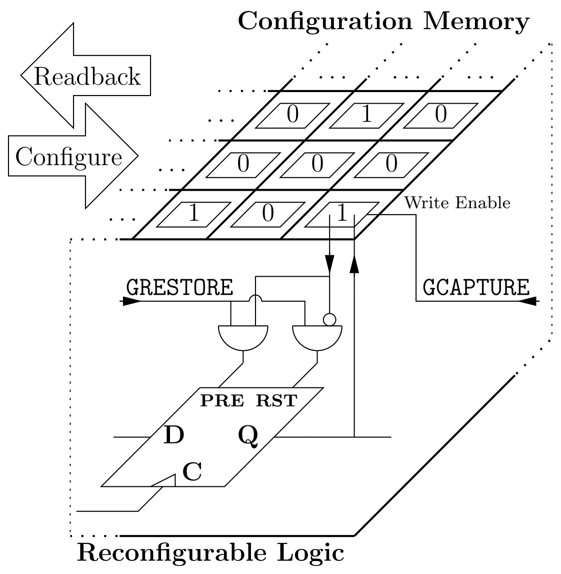

Fig. 2. Illustration of the FPGA's configuration memory and reconfigurable logic. Also shown is how these two layers are connected for a configurable flip-flop. When `GRESTORE` is pulsed, the flip-flop is asynchronously resetted (to '0') or presetted (to '1') depending on the value in the configuration memory. When `GCAPTURE` is pulsed, the flip-flop's current value is written back to the configuration memory.

4.2 Reading out the flip-flop power-up values.

The random power-up values of the flip-flops now can be read and used as a PUF in the same way as an SRAM PUF. Two methods to read the random flip-flop values were tried:

– A circuit can be configured into the FPGA that reads out a number of flip-flop power-up values, processes them if necessary and outputs them. There is however a pitfall to this method. Since no flip-flops are initialized, the read-out circuit itself will also start in an undetermined state. The read-out circuit should be designed such that it can recover to a known state, possibly needing an external reset signal.
– Using the capture and readback functionality of the configuration controller, the initial (random) flip-flop values can be outputted directly through a configuration port without loading a design. This is done by making a bitfile that contains a `GCAPTURE`-command and the necessary commands to perform a readback of the portions of the configuration memory that contain the flip-flop values.

4.3 Measurement results

Using the methods described in Subsections 4.1 and 4.2, the power-up values of flip-flops were experimentally read out. The measurement was done on three different Virtex-II Pro FPGAs. The power-up value of 4096 flip-flops (32 columns $\times$ 128 rows) were measured after 101 consecutive power-ups[4]. The result is shown in Figure 3.

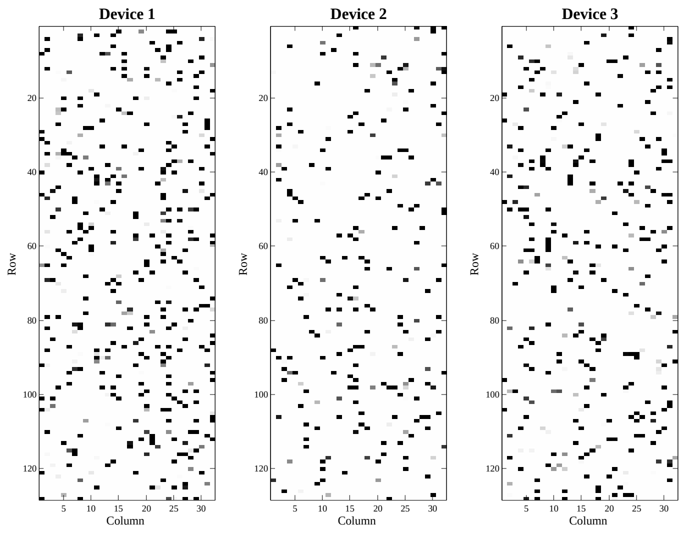

Fig. 3. Measurement results of flip-flop power-up values on three distinct but equal devices. A white square signifies that the flip-flop at that position has a '0' as power-up value, a black square means it starts up as '1'. Some flip-flops powered up as '0' some of the 101 measurements, and as '1' the other times. These are depicted as gray squares, with the gray scale signifying the preference towards '0' or '1'.

From the measurement results in Figure 3, it is clear that there is a pronounced favor of '0' over '1' as power-up value in all three devices. This can be explained by some deterministic inbalance in the memory cell of the flip-flop, either by construction, or by a deterministic offset in the mismatch. This inbalance also becomes clear in Figure 4, showing the histogram plot of the ratio of flip-flops that power up at '0' versus the ratio that power up at '1'.

5 Post-processing

From the measurement results in Subsection 4.3 it is clear that the amount of randomness present in the power-up values of the flip-flops is limited. This is due to the strong favor of most flip-flops to start up in '0', rather than a more or less fifty-fifty chance. Because of this strong non-uniformity, the power-up

[4] The same flip-flops were measured on all devices.

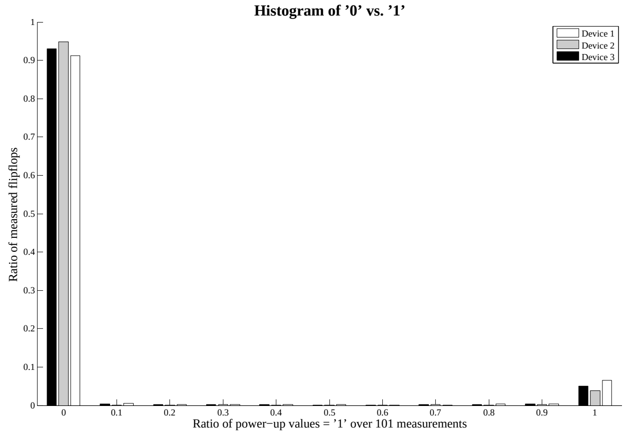

Fig. 4. Histogram of the ratio of flip-flops that prefer '1' as power-up value. This plot shows that more than 90% of the measured flip-flops always power-up as '0' versus little less than 10% which always start up as '1'. Less than 1% will start up as '0' some of the times and '1' the other times.

bits cannot be used directly, *e.g.* as key bits. Some processing has to be done to extract uniform randomness.

There is also another reason why postprocessing is necessary. In Subsection 4.3 it was shown that around 1% of the flip-flops do not have a strong preference to neither of both power-up values, and start up as '0' some of the times and '1' the other times. These flip-flops introduce an amount of *noise* in the response. To use the power-up values in cryptographic algorithms, *e.g.* as key bits, all this noise should be removed!

5.1 Majority Voting

Majority Voting is the process of collecting multiple measurements, either through repeated measuring (temporal) or by considering multiple flip-flops at once (spatial), and determining one output value based on the individual values or *votes* of the single measurements. When the number of postive votes exceeds a certain threshold or *majority*, the output is said to be positive ('1'), otherwise the output is negative ('0').

Temporal Majority Voting. When doing temporal majority voting or TMV, a number N_T of consecutive measurements of a single cell are considered. The threshold or majority M_T of the voting is generally set to be $M_T = \frac{N_T - 1}{2}$ when N_T is odd. This is a good way to decrease the noise on flip-flops that do not always power up to the same value, but still have a distinct favor of one state over the other. Assume *e.g.* a flip-flop that powers up to '0' 90% of the times and to '1' 10% of the times. If a TMV over $N_T = 5$ measurements is done with

$M_T = 2$, the output of the TMV on this flip-flop will give a '0' 99.1% of the times and '1' 0.9% of the times. If powering up to '1' is considered erroneous for this flip-flop, the probability of this error occuring has decreased from 10% to less than 1% by performing the TMV. Applying TMV requires N_T times more time, so it exchanges running time of the PUF against error probability of the output.

Spatial Majority Voting. Spatial majority voting or SMV considers a number N_S of simultaneously measured values, generally neighbouring flip-flops are considered. SMV also can decrease the noise, but more importantly, it is capable of producing an output that is closer to uniformity. This is achieved by choosing a majority M_S that is not exactly half the number of votes, but with a certain offset to compensate for the inbalance in the original values. Assume *e.g.* $N_S = 9$ flip-flops and each will either always vote '0' or always vote '1'. The probability that a flip-flop is an always-'0'-voter is 90%, and an always-'1'-voter is 10%. The majority of the voting is set to $M_S = 0$, this means that in order to get a negative outcome, the vote would have to be unanimously negative. This SMV produces an output '0' with probability 39% and a '1' with probability 61%. The SMV hence transforms the strongly non-uniform $90\% - 10\%$-distribution into a more uniform $39\% - 61\%$-distribution. Applying SMV will require N_S times more flip-flops, so it exchanges area of the PUF against distance to uniformity of the output.

Results To demonstrate the usefulness of majority voting on the measurements obtained in Subsection 4.3, two measures are considered:

- **Inter-device distance $\boldsymbol{\delta_{inter}}$.** This is the fractional Hamming distance between the power-up values of the same flip-flops on two distinct devices. This measure hence expresses how easy it is is to distinguish two devices based on their powerup values. In general, it measures the uniqueness of the PUF responses with respect to the entire device population and gives a notion of the uniform randomness contained in the power-up values. For practical use of the PUF, δ_{inter} should be as close to 50% as possible.
- **Intra-device distance $\boldsymbol{\delta_{intra}}$.** This is the fractional Hamming distance between the power-up values of the same flip-flops on one device at two distinct measurements. Two distinct measurements on the same device will differ due to noise and δ_{intra} gives a clear notion of the amount of noise disturbing the PUF responses. For practical use of the PUF, the intra-device distance should be as close to 0% as possible.

The inter- and intra-device distances[5] are key parameters that describe the practical usefulness of a PUF construction. They are often used to interpret the experimental data of PUF measurements [1, 15]. We show how TMV and SMV improve δ_{inter} and δ_{intra} substantially.

The inter- and intra-device distances are calculated for the obtained measurements before majority voting, and after combined temporal and spatial majority

[5] Synonymous terms are *within-class* and *between-class* distance.

voting with $(\langle N_T = 5, M_T = 2\rangle, \langle N_S = 9, M_S = 0\rangle)$. The resulting histograms of both measures before and after the majority voting are shown in Figure 5. The SMV transforms the poorly random distribution of Figure 5(a) into the near-uniform distribution with inter-device distances very close to 50% for all pairs of devices shown in Figure 5(b). The SMV however also slightly increases the noise, since a single biterror in one of nine cells can cause a biterror in the output. Additional TMV controls the noise and keeps the biterror probability below 5%.

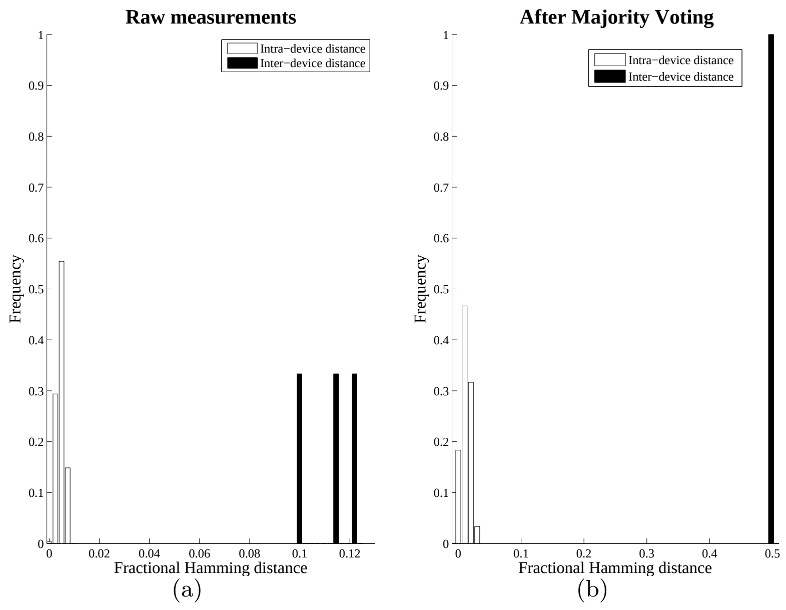

Fig. 5. Inter- and intra-device distances, before majority voting, and after combined temporal and spatial majority voting with $(\langle N_T = 5, M_T = 2\rangle, \langle N_S = 9, M_S = 0\rangle)$.

5.2 Fuzzy Extractor

Majority voting is a convenient method to transform poorly uniform and noisy measurements into more random distributions with less noise. It is however not perfect! Due to its discrete nature, SMV will in general not be able to reach a perfect uniform distribution[6]. The number of repeated measurements which TMV needs to reduce the noise by a constant factor rises exponentially[7]. This

[6] The nice result of Subsection 5.1 with the inter-device distance very close to 50% is considered exceptional.

[7] This is due to the Chernoff Bound.

means TMV is very good at correcting cells with a low error probability, but quickly becomes inefficient as the error probability increases. However, for cryptographic applications, a completely noise-free response with a perfect uniform distribution is often required. To achieve this, another postprocessing step is required.

A Fuzzy Extractor [23, 24] generally comprises two major steps. Firstly, possible biterrors are corrected. In a second step, uniform random bits are extracted from the corrected non-uniform random PUF responses. It will however do this at the expense of an amount of response bits. Majority Voting before using a Fuzzy Extractor is therefore recommended to keep the losses in the Fuzzy Extractor as low as possible. Since the Majority Voting also induces a certain loss, an optimization step is needed to achieve optimal efficiency.

Information Reconciliation = Error Correction. The error correction step of a Fuzzy Extractor is generally called Information Reconciliation. It makes use of error correcting codes in a special way to correct possible biterrors in the PUF response. The losses due to the error correction are a function of the average biterror probability in the PUF responses.

A straight-forward example is given. We want to derive a 128 bit key from the PUF responses after majority voting has taken place, so we start from the results as given by Figure 5(b). A biterror probability of at most 5% needs to be corrected. When using a BCH(255,47,42) error-correcting blockcode, the probability of uncorrected biterrors after correction is smaller than 10^{-11}. Three 255-bit blocks or $3 \times 255 = 765$ flip-flop powerup values are used to extract $3 \times 47 = 147$ error-free bits.

Privacy Amplification = Randomness Extraction. In the Privacy Amplification step, a so called Strong Extractor is used to extract random bits that are very close to uniformity from random responses that are often not uniformly distributed. Strong Extractors can be implemented using 2-universal hashfunctions with a uniform random seed. The closest possible distance[8] from uniformity that a Strong Extractor can achieve is a function of the min-entropy present in the PUF responses, and the compression ratio of the hashfunction.

Figure 5(b) shows that the response bits after majority voting are already closely uniformly distributed. However, a privacy amplification step is still recommended to increase the security of the keybits. The amount of compression that is needed should be estimated from elaborate measurements on a large number of devices.

Phases of a Fuzzy Extractor. A Fuzzy Extractor works in two phases, as explained in [23]. In the *Generation phase*, a robust and uniform random bitstring is extracted from the PUF response. Additionally, an amount of *Helper Data* is produced in both steps of the Fuzzy Extractor. In the second phase

[8] Here, the statistical distance between distributions of random variables is meant.

or *Reproduction phase*, the previously produced Helper Data is combined with the same, possibly noisy, PUF responses to extract the identical bitstring as in the first phase. The Helper Data should hence be stored and made available in the Reproduction phase. If the noise is small enough, the Fuzzy Extractor construction guarantuees a perfect reconstruction of the extracted random bitstring. Moreover, the Helper Data does not contain any information about the PUF response, nor about the extracted bitstring, hence it does not need to be kept confidential! The extracted bitstring is uniform random, even if the Helper Data is known, and can hence be used in cryptographic applications, *e.g.* as a private key.

6 Practical use

6.1 General security advantages of a PUF

In Section 5 it has been described how some post-processing steps can turn a PUF response into a secret key. The use of a PUF as source of the key material offers some security advantages over simply storing the binary digits of the key in a memory:

- **Cost.** In order to preprogram a secret key in a chip, some kind of non-volatile memory has to be present, which could induce an extra cost. Using a PUF to extract a key requires no non-volatile memory. This work demonstrates how a PUF can extract keybits from volatile memory, like flip-flops, which are often abundantly present.
- **Trust.** In order to assign a unique identification key to a chip, it must be hard-programmed in a one-time-programmable or OTP memory. This is often done by the chip manufacturer. The user hence has to place it's trust in an external party which might (accidentally) compromise the private key. Since a PUF is intrinsically present in the chip's physical structure and beyond the control of the chip manufacturer, no externalized trust is necessary.
- **Tamper evident.** Well-equiped attackers will attack a hard-programmed binary key by opening up the chip's package and uncovering the cells storing the keybits in the non-volatile memory [9], *e.g.* by using Focused Ion Beams, lasercutters, microprobes, *etc.*... Similar attacks could be done trying to learn the parameters that determine the unique PUF responses. However, since these attacks are invasive, they often change these unique physical parameters and consequently alter the PUF's challenge-response-behavior. PUFs have the ability to make a chip tamper evident.

6.2 IP-protection on reconfigurable devices

The Intrinsic Flip-flop PUFs, as described in this work, can be used to produce secret keys for protocols to protect the configuration of a reconfigurable device from cloning and tampering. Examples of such protocols are given in [1, 12]. The advantage is that such keys are bound to a specific instantiation of an FPGA. Bitfiles can hence be encrypted on a per-device basis, and cloning of encrypted bitstreams becomes useless.

However, an additional problem concerning PUFs on reconfigurable devices exists. If an adversary is able to learn the implementation details of the PUF, he can customize the design containing the PUF implementation to output the PUF responses that need to be kept secret. Because of the flexibility of the FPGA, the adversary can easily load his customized design and learn the PUF responses. This problem holds for all types of PUFs on reconfigurable devices that were proposed, including the Flip-flop PUF, and can only be overcome by making hardware changes to the FPGA chip that prevent the output of PUF responses. Since such hardware changes induce a large cost for FPGA manufacturers, they will not be made lightly. We believe however that the changes needed for the secure use of Flip-flop PUFs would be minimal. The Flip-flop PUF can be read out by issuing *software* commands to the configuration controller. Additionally, after reading out the necessary responses, all the flip-flops can easily be resetted to avoid further readout of the secret responses by unauthorized parties. If the configuration controller would perform these steps automatically, before the bitfile is loaded, the FPGA can extract its own secret key needed to decrypt the coming bitfile. That key would only be present in volatile memory before and during the loading of the bitfile. To achieve this, no substantial changes in the controller's hardware are believed to be necessary.

7 Conclusion

In this work, the possibility of using the powerup values of flip-flops on FPGAs as PUF responses is explored. A proof-of-concept implementation was made and the provided experimental data shows that this Flip-flop PUF achieves results similar to the previously described SRAM PUF. Moreover, Flip-flop PUFs can be implemented in common FPGAs, whereas this was seldomly the case for SRAM PUFs due to the unavoidable initial reset of SRAM-cells.

To confirm the strength of Flip-flop PUFs on reconfigurable devices, more experimental measurements on a number of different platforms are needed. Moreover, the sensitivity to changing environmental conditions (temperature, supply voltage, . . .) should be studied. This will all be considered in future work.

References

1. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In Paillier, P., Verbauwhede, I., eds.: CHES. Volume 4727 of Lecture Notes in Computer Science., Springer (2007) 63–80
2. Roy, J.A., Koushanfar, F., Markov, I.L.: Epic: Ending piracy of integrated circuits. In: DATE, IEEE (2008) 1069–1074

3. Alkabani, Y.M., Koushanfar, F.: Active hardware metering for intellectual property protection and security. In: SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2007) 1–16
4. King, S.T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., Zhou, Y.: Designing and implementing malicious hardware. In: LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats, Berkeley, CA, USA, USENIX Association (2008) 1–8
5. Markoff, J.: F.B.I. Says the Military Had Bogus Computer Gear. The New York Times (May 9, 2008) Available online (`http://www.nytimes.com/2008/05/09/technology/09cisco.html`).
6. Note, J.B., Éric Rannaud: From the bitstream to the netlist. In: FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, New York, NY, USA, ACM (2008) 264–264
7. Baetoniu, C., Sheth, S.: XAPP780: FPGA IFF copy protection using Dallas Semiconductor/Maxim DS2432 Secure EEPROM. Xilinx Inc. (August 2005)
8. Lesea, A.: IP Security in FPGAs. Xilinx White paper (February 16, 2007) Available online (`http://www.xilinx.com/support/documentation/white_papers/wp261.pdf`).
9. Witteman, M.: Advances in Smartcard Security. Information Security Bulletin (July 2002) 11–22
10. Hammouri, G., Birand, B., Sunar, B.: IP protection using PUF-based stream cipher. submitted
11. Kumar, S.S., Guajardo, J., Maes, R., Schrijen, G.J., Tuyls, P.: The Butterfly PUF: Protecting IP on every FPGA. In: IEEE International Workshop on Hardware-Oriented Security and Trust – HOST 2008, IEEE (June 9th, 2008)
12. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: Physical unclonable functions and public-key crypto for FPGA IP protection. In: Field Programmable Logic and Applications, 2007. (August 2007) 189–195
13. Schellekens, D., Tuyls, P., Preneel, B.: Embedded trusted computing with authenticated non-volatile memory. In Koch, K.M., Lipp, P., Sadeghi, A.R., eds.: TRUST 2008. Volume 4968 of Lecture Notes in Computer Science., Villach,Austria, Springer-Verlag (2008) 60–74
14. Pappu, R.S.: Physical one-way functions. PhD thesis, Massachusetts Institute of Technology (March 2001) Available at `http://pubs.media.mit.edu/pubs/papers/01.03.pappuphd.powf.pdf`.
15. Gassend, B., Clarke, D., van Dijk, M., Devadas, S.: Silicon physical random functions. In: CCS '02: Proceedings of the 9th ACM conference on Computer and communications security, New York, NY, USA, ACM (2002) 148–160
16. Veendrick, H.: Deep-submicron CMOS IC. second edn. Kluwer academic publishers (2000)
17. Boning, D., Nassif, S.: Models of process variations in device and interconnect. In Chandrakasan, A., Bowhill, B., eds.: Design of High Performance Microprocessor Circuit. IEEE Press (2000)
18. Lee, J., Daihyun, L., Gassend, B., Suh, G., van Dijk, M., Devadas, S.: A technique to build a secret key in integrated circuits for identification and authentication applications. VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on (17-19 June 2004) 176–179
19. Suh, G.E., Devadas, S.: Physical Unclonable Functions for Device Authentication and Secret Key Generation. In: Proceedings of the 44th Design Automation Con-

ference, DAC 2007, San Diego, CA, USA, June 4-8, 2007, New York, NY, USA, ACM (2007) 9–14

20. Su, Y., Holleman, J., Otis, B.: A 1.6J/bit 96% stable chip ID generating circuit using process variations. In: ISSCC. (2007) 406–611
21. Ozturk, E., Hammouri, G., Sunar, B.: Physical unclonable function with tristate buffers. Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on (May 2008) 3194–3197
22. Xilinx: VirtexII Pro User Guide (v4.2) (November 2007) Available online (`http://www.xilinx.com/support/documentation/user_guides/ug012.pdf`).
23. Dodis, Y., Reyzin, L., Smith, A.: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In: EUROCRYPT. (2004) 523–540
24. Linnartz, J.P., Tuyls, P.: New shielding functions to enhance privacy and prevent misuse of biometric templates. In: In AVBPA 2003. (2003) 393–402