Design Space Exploration and Automation of Multi-Personality PUFs Using FPGA

Abstract—**Physical Unclonable Function (PUF) is widely used as a hardware security primitive. It provides a secure, robust, and dynamic approach for various security purposes such as secret key storage, random number generator, identification, and authentication etc. One of the important gaps that remains in designing secure distributed systems is the enabling of multi-identity encoding schemes to facilitate anonymous IC authentication. In this paper we propose a multi-personality PUFs (MP-PUF) design that meets such needs. This design is able to convert both strong and weak PUFs into MP-PUFs. In addition, we will explore the key critical details of the MP-PUF's FPGA implementation. The discussion of these details will 1) help with increasing the uniqueness and reproduction of the proposed MP-PUF, 2) inspire improvements to the existing popular PUF designs on FPGA, such as the ones based on Ring Oscillator and Arbiter PUFs. Moreover, we have also developed an open-source design automation software tool to generate PUFs using user specified parameters to enable easier design exploration of FPGA-based PUFs.**

Index terms — ***PUF, FPGA, multi-personality, design automation, authentication.***

I. Introduction

There is always a need for authenticating and identifying hardware devices. The conventional approach is usually to attach a secret key or ID to the device and verify its validity. However, this static approach suffers from several vulnerabilities. Firstly, it takes an expensive non-volatile memory to store such a piece of secret. Secondly, usually keys or IDs (such as RFIDs or barcodes) are separately produced apart from the devices they attach to. Thus a server or user is unable to distinguish between a genuine product and a counterfeit with a genuine ID. Finally, since under most occasions an ID or a key is essentially a string of numbers, if this static string (or even just part of it, such as the last four digits of a credit card number or social security number) is acquired by another party, the key's owner can easily be spoofed.

Therefore, there is a demand of a new approach for authentication and identification, which is inexpensive, naturally integrated in the device, unpredictable and unclonable, and supportive of dynamic verification. Under this demand the physical unclonable function (PUF) [1] has been proposed as a fitting solution. A PUF is a piece of hardware that produces unpredictable responses upon challenges due to their manufacturing variations. Each PUF's output (response) is a non-linear function of the outside stimulation (challenge) and the PUFs own physical, intrinsic, and unique diversity, in another word, "Silicon Fingerprints". To apply PUF in the security field, it also needs to have the property of easy to manufacture but hard to duplicate, even under exact the same circuit layout and manufacturing procedures. Because of its attributions of randomness and uniqueness, PUFs can be used for random number (i.e. secret keys for cryptographic purposes) generation and storage, authentication (i.e. IC, user, product authentication), and identification etc.

It is notable that the same challenge should receive different responses from different PUFs, even though those PUFs are built from identical design on identical circuits. Also ideally different responses should have as large a Hamming distance as possible. Usually non-cryptographic hash functions are used to randomize the responses to amplify their variations in order to increase each PUF's uniqueness.

Besides the ordinary use of PUF to authenticate a single device as a single entity, there is also a need to have a single device authenticated as multiple identities for multiple applications. For example, an individual might not want to be identified as the same person when he logs into a regular bank account, trades over a Bitcoin platform, and plays Texas Hold'em on a gambling website etc. Thus for this purpose there is another type of PUF invented called the controlled PUF (CPUF) [2] or *multi-personality PUF* (MP-PUF).

The previously proposed designs of MP-PUFs would work the best with strong PUFs, as the variety of multiple personalities can be restrained by weak PUFs' limited responses. In addition, in the previous works although the high level architecture of the MP-PUF is given, few details were provided in terms of their FPGA implementation. In this paper we propose a new design of MP-PUF which is able to upgrade both strong and weak PUFs with multi-personality. We also include sufficient details of its FPGA implementation to help (i) improve the PUF's uniqueness and (ii) enable reproduction.

The major contributions of this paper are:

1) The new MP-PUF provides distinct behaviors across different personalities. It also enhances weak PUFs with higher diversity and larger size of their responses;
2) It explores the FPGA implementation problems and solutions on Xilinx's Vivado IDE;
3) Besides MP-PUF, it suggests a few improvements to the designs of several popular delay PUFs;
4) It provides a PUF automation tool to generate PUFs with given parameters, so that even researches who are not familiar with FPGAs can have access to PUFs.

The rest of the paper is organized as follows. Sections II gives a brief introduction of the related works on PUFs. Section III introduces the proposed new MP-PUF on a high level. Section IV dives into its FPGA implementation problems and proposes solutions in the context of the latest Vivado IDE. Section V continues the discussion on MP-PUF's FPGA implementation by exploring the routing and placement approaches. Section VI evaluates the proposed MP-PUF and introduces our PUF automation tool. Section VII conlcudes the paper.

II. Related Works on PUFs

In this section the PUF's authentication protocol and several popular PUF designs will be introduced. We focus more on the PUFs with intrinsic randomness rather than the ones with explicitly introduced randomness (such as the optical PUF or coating PUF), because they can be included in a design without modifications to the manufacturing process. To better facilitate the presentation, we introduce the following notations:

- CHL_i: the i^{th} challenge to a PUF;
- RSP_i: the i^{th} response of a PUF;
- PSN_k: the k^{th} personality of a MP-PUF;
- c: the number of bits in a PUF's CHL;
- r: the number of bits in a PUF's RSP;
- m: the number of ROs in a RO group which produces 1 bit of response;
- $|PSN|$: the personality set size of a PUF;
- $|CRP|$: the CRP set size of a PUF.

The authentication procedure of a PUF is as follows:

i Before a PUF is released to its owner, the server or manufacturer will challenge it with multiple challenges $\{CHL_0, CHL_1, \cdots, CHL_i, \cdots, CHL_n\}$ and store their responses $\{RSP_0, RSP_1, \cdots, RSP_i, \cdots, RSP_n\}$;
ii When a released PUF needs to be authenticated, the server will arbitrarily send a pre-stored CHL_i to the PUF;
iii When the PUF returns a response to the server, this response will be compared with the server's pre-stored RSP_i to verify its validity.

Based on the size of the CRPs, PUFs can be categorized as weak and strong PUFs:

1) Weak PUF: The $|CRP|$ size grows linearly with the PUF size, mostly used for secret key generation and storage. The weak PUFs usually only has one CRP (such as the memory PUF), and so its CRP should be secured and not accessible by opponents;

2) Strong PUF: The $|CRP|$ size grows exponentially with the PUF size, mostly used for dynamic authentication. For strong PUFs there can be a large set of CRP publicly accessible. However with this large public set, it should still be impossible for an opponent to predicate any unknown CRPs.

Based on the source of randomness, there are mixed-signal (Analog), memory, and the delay PUFs etc. The latter two are the most commonly seen PUFs nowadays.

The SRAM PUFs, the DRAM PUFs, and the commercialized Butterfly PUFs [3] are representatives of the memory PUFs. On these PUFs, usually the reading of part of a memory serves as the challenge, and the values read from those cells are the response. Memory PUFs are mostly used for secret key generation and storage. Having usually only one challenge-response pair, memory PUFs are considered as weak PUFs and so the reading of the memory is supposed to be restricted. Otherwise this type of PUF can be easily cloned (in laboratory environment) through either standard on-chip channels or laser stimulations [4], [5].

Comparing with the memory PUF, we are more interested in the delay PUFs since they usually generate much larger sets of CRPs than the memory based PUFs. Adversaries with a limited amount of CRPs can hardly spoof the character of a genuine PUF with a large $|CRP|$. Also there can be a new challenge for every round of authentication, so that replay attacks will never apply. In addition, most secure PUFs are also based on the basic delay PUFs.

The delay PUFs take the advantage of the random variations in delays in the basic circuit elements (gates, LUTs, etc.). Given an input (challenge), a race condition will appear in the circuit to generate a delay difference (response). Different chips with the same circuit will have different delay patterns, providing the uniqueness of those PUFs. The commonly seen delay PUFs are: Ring Oscillator (RO) PUF, Arbiter PUF, Glitch PUF, HELP and etc. Among them the RO and Arbiter PUFs are the most popular.

A. Ring Oscillator (RO) PUF

A RO PUF [6], [7] consists of multiple inverter chains, and in each chain an odd number of inverters are connected in a loop. Due to the manufacturing variation resulting in each inverter's delay, each oscillator has a different and unpredicted frequency from others. At the end of this circuit there are two counters of the same size whose increment speed depends on the frequencies of the corresponding chains, which serve as the clocks. A 1-bit RO PUF is shown in Fig. 1. The challenge will be the input to the decoder selecting two ROs from the group of m ROs, and the response is the comparison of the values of the two counters clocked by the selected ROs.

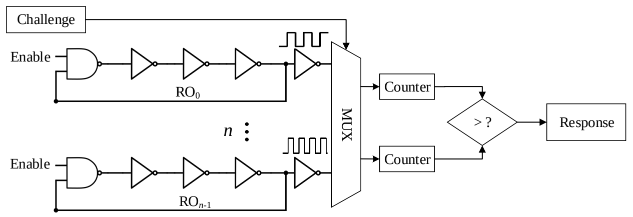

Fig. 1: The circuit above produces 1 bit of PUF response. To generate more response bits, more RO groups and counters can be added.

Suppose for a RO PUF, there are r RO groups and in each group there are m ROs. If the first and second half of the challenge CHL is used to select two different ROs in this group, then the challenge size c will be:

$$c = 2 \cdot \lceil log_2 m \rceil. \quad (1)$$

The total number of CRP is:

$$|CRP| = \binom{m}{2}. \quad (2)$$

While the Arbiter PUF requires highly symmetric implementation, the RO PUF does not, which makes it easier to be implemented on FPGAs, although it still requires identical routing and placement across the ROs.

B. Arbiter PUF

Arbiter PUF [8], [9] is multiplexer (MUX) and D-flip-flop (DFF)-based. It takes two sets of parallelly connected MUXes and the output is latched by a DFF as shown in Fig. 2. As a positive edged signal is applied to the first two MUXes, the binary challenge vector determines which two paths the signal will go through. Because of the unique intrinsic delay in each MUX, the positive edged signal will create a racing condition between the two paths to the clock port and data port of the DFF. The output of the DFF is then a random response bit as shown in Fig. 2. In this case the two parallel MUX chains' wiring should be completely symmetric, so that the only factor that determines the response will be the MUXes' manufacturing variations.

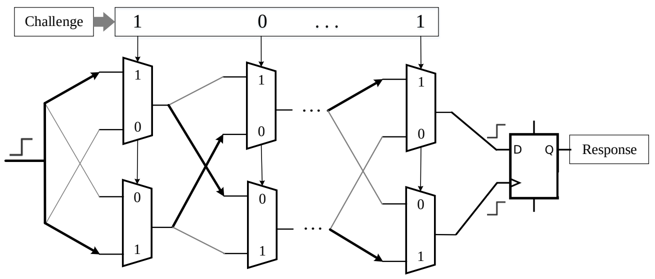

Fig. 2: The challenge vector determines whether the rising signal will reach the DFF data port or the clock port. In the former case 1 will be the response, and 0 will be the latter case. The circuit above produces 1 bit of PUF response.

For an Arbiter PUF, if there are r MUX chain pairs, and the challenge has c bits, then the total number of CRP is:

$$|CRP| = 2^c. \tag{3}$$

The Arbiter PUF is considered as a strong PUF because of [Eq. 3] and thus RO PUF is a weaker PUF. Our proposed multi-personality PUF (MP-PUF) is based on both of them.

III. Multi-personality PUF (MP-PUF)

In this subsection we will propose our new multi-personality PUF (MP-PUF) based on RO and Arbiter PUFs. Firstly we will briefly describe the high level architecture and functionality of the MP-PUF. Sections IV through VI expound on the implementation details.

Unlike ordinary PUFs which only have one input known as the challenge, the MP-PUF has "personality" as another input. For an ordinary PUF, no matter how many challenge and response pairs (CRPs) it has, it can only be identified and authenticated for one identity. However, an MP-PUF can be identified and authenticated for as many identities as the personalities it owns. Explicitly, given the same challenges, with different personalties, an MP-PUF should behave differently by its responses.

PUFs work in the form of CRPs. The MP-PUF's mechanism is similar, except the MP-PUF's owner is able to select different personalities upon different applications' authentication requests:

i. An application indexed by k holds a set of CRPs of a MP-PUF without knowing its multi-personality nature;
ii. In need of authentication, the application sends out a pre-stored CHL_i to the MP-PUF's owner;
iii. The owner of the MP-PUF applies both CHL_i and the personality PSN_k to the MP-PUF, and generates a response;
iv. The application compares the response with the stored RSP_i to verify its identify, namely the k^{th} personality.

The Controlled PUF (CPUF) or Multi-personality PUF (MP-PUF) was originally proposed by [1], [2]. The variety of a CPUF under various personalities are achieved by hashing a personality and the original challenge into a new challenge. In this way the same challenge will be mapped to different responses under different personalities. This design would work the best with strong PUFs. As for the weak PUFs with very limited CRPs, the control in fact only increases the ways of mapping between CHL and RSP, but not the number of actual responses. Thus the RSP set stored at any two different applications can have a large overlap. This can be leveraged by man-in-the-middle adversaries or malicious applications.

Therefore in this paper we propose a new design of MP-PUF which is able to increase not only the personalities, but also the number of responses for both strong and weak PUFs. The proposed new MP-PUF's circuit will be altered under any change of personality. In this way each personality will have its own unique PUF circuit so that even if one personality's full response set is acquired by an adversary or a malicious application, the response set of another is still beyond prediction. The comparison between the design concepts of conventional controlled PUFs and the proposed MP-PUF is shown in Fig. 3.

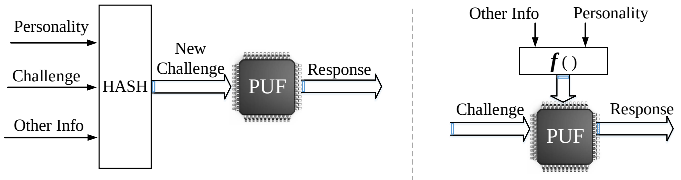

Fig. 3: The conventional CPUF (left) uses the same circuit for all personalities, while in the proposed MP-PUF (right), the personality alters the internal circuit of the PUF. For simplicity we have omitted the fuzzy extractor and the output hash in these diagrams.

We propose two MP-PUFs based on two classic primitives: the oscillator ring (RO) PUF, and the Arbiter PUF. In the basic elements of these two PUFs, three functional blocks are integrated: the personality (PSN) box, strict avalanche criterion (SAC) network, and first order Reed-Muller (FORM) encoder.

Fig. 4 illustrates a basic element (a single Ring Oscillator) of the RO PUF based MP-PUFs. The PSN Box is installed between every two neighboring inverters.

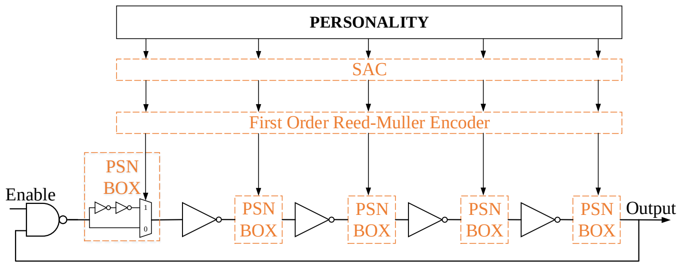

Fig. 4: The upgraded RO for the RO PUF-based MP-PUF. A group of such element and two counters will form a single bit of response.

Fig. 5 illustrates a basic element (a pair of MUX chains) of the Arbiter PUF based MP-PUFs. The PSN Box is introduced between every two stages of MUX pairs.

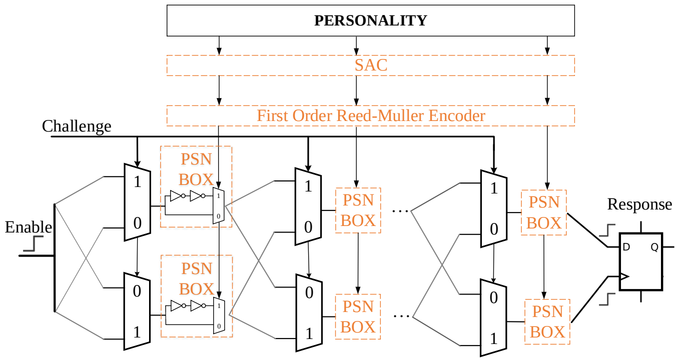

Fig. 5: The upgraded MUX chain for the Arbiter PUF-based MP-PUF which forms a single bit of response.

A. PSN Box

The PSN box manifests the personality variation of MP-PUF by altering the PUF circuit. It consists of two inverters and one 2-to-1 MUX as shown in Fig. 4 and 5. On FPGAs, each inverter is instantiated by a 1-to-1 lookup-table (LUT), and the MUX by a 2-to-1 LUT. Each PSN box takes in one bit of the transformed personality input, and will affect the timing but not the value of the propagated signal. With different inputs to the PSN Box, the RO and Arbiter PUF-based MP-PUFs will function under various timing characteristics, thus behaving differently under the change of personalities.

For a five-stage RO PUF based MP-PUF in Fig. 4, assume there are m ROs in a RO group, and there are r such groups in the MP-PUF. Since the number of personalities will be $|PSN| = 2^5$, the total number of CRPs is:

$$|CRP| = 2^5 \cdot \binom{m}{2}, \tag{4}$$

which is also the total number of unique responses.

For an Arbiter PUF based MP-PUF in Fig. 4, assume there are c MUX pairs in a MUX chain, and there are r such chains in the MP-PUF. Since the number of personalities will be $|PSN| = 2^c$, the total number of CRPs and unique responses are:

$$|CRP| = 2^c \cdot 2^c = 2^{2c}, \tag{5}$$

In contrast, comparing with the ordinary PUF with single personality and the CPUF in [1], [2], the proposed MP-PUF has its advantages in both the $|CRP|$ set size and the unique responses set size $|RSP|$:

TABLE I: $|CRP|$ and Unique Response Set $|RSP|$ Comparison

PUF Type	Proposed MP-PUF $\lvert CRP\rvert$	Proposed MP-PUF $\lvert RSP\rvert$	Ordinary PUF $\lvert CRP\rvert$	Ordinary PUF $\lvert RSP\rvert$	CPUF $\lvert CRP\rvert$	CPUF $\lvert RSP\rvert$
RO	$2^5 \cdot \binom{m}{2}$	$2^5 \cdot \binom{m}{2}$	$\binom{m}{2}$	$\binom{m}{2}$	$\lvert PSN\rvert \cdot \binom{m}{2}$	$\binom{m}{2}$
Arbiter	2^{2c}	2^{2c}	2^c	2^c	$\lvert PSN\rvert \cdot 2^c$	2^c

With more CRPs and unique responses, the proposed MP-PUF especially its RO PUF based version will be more resilient against replay attacks from the adversaries who have acquired the responses of one or more personalities.

Remark III.1. If more diversity in personality is needed (especially for the weak PUFs such as RO PUF), the PSN box can further evolve in at least two ways as shown in Fig. 6: 1) by fitting in multiple PSN boxes between two stages; 2) by adding more choices of timing routes to the MUX. Both are able to provide a great number of additional personalities/delay circuit variations to the MP-PUF.

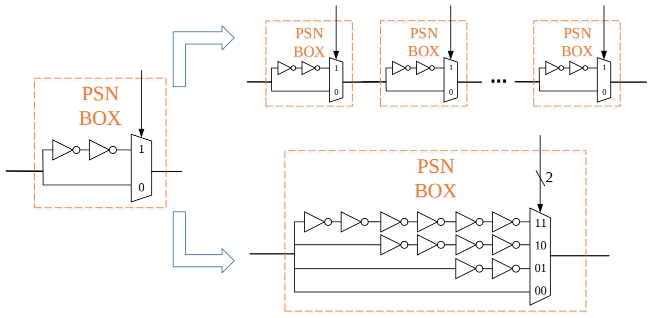

Fig. 6: Suppose h 2-to-1 PSN boxes or h timing choices of a h-to-1 PSN box are added between each two stages/inverters of a RO, then the number of personalities grows from 2^5 to 2^{5h}.

B. Strict Avalanche Criterion Network

On receiving an input of a personality, it will be firstly transformed by a Strict Avalanche Criterion (SAC) network [10], which is also used in the design of lightweight secure PUFs [11]. In a SAC network, whenever a single input bit is flipped, each output bit should have a probability of 0.5 to flip. The introduction of the SAC network is to increase the unpredictability and diversity an MP-PUF circuit generated by a given personality. Thus every two personalities with a small Hamming distance will not result in similar circuits. This helps to prevent learning attacks across different personalities.

C. First Order Reed-Muller Encoder

Side channel attack is also a powerful tool to acquire information of a PUF when used together with learning attacks [12], [13]. Therefore after the SAC network, we will also encode the output with the first order Reed-Muller (FORM) encoder. A b-bit FORM codeword can be generated by check matrix $M = [\frac{M_1}{M_0}]$, where M_1 is a single row of all 1's, and the columns of M_0 consist of all different vectors of $\lceil \log_2 b \rceil$ bits. One important attribution of FORM codes is that, all its codewords will be equal weights, and the number of 1's and 0's in any codeword are the same (the vectors of all 1's and all 0's are excluded from codewords). In this way there are always half of PSN boxes turned on to the slower route (port 1 of the MUX), and half of them to the faster route (port 0 of the MUX), which makes the power analysis on different personalities harder for attackers.

IV. Implementation Problems and Solutions

In this section we will dive into the details of the MP-PUF's implementation on FPGA, which the previous works on MP-PUFs have not provided. PUF's low level implementation is non-trivial. Without a proper implementation, regardless how elegant the high level architecture is, the PUFs will lose their uniqueness. In addition, we will also propose a few improvements in the design in order to increase the MP-PUF's uniqueness. These improvements can also be beneficial to other delay PUF's design and implementation.

Previously, most researchers prefer to use the combination of Xilinx ISE + Xilinx FPGA because ISE allows users to configure the placement and route of the circuits using its FPGA Editor [14], [15], [16]. However, from October 2013 ISE is no longer updated and most researchers in the FPGA field have moved to the Vivado IDE. Nevertheless, since then there are few works being done with clear presentation of PUF design methodology with this new IDE. The possible major reasons are 1) Vivado does not have the convenient FPGA Editor anymore and users need to find the alternatives such as XDC macro or hard macro; 2) comparing to its predecessor, the new Vivado IDE has many new and unique features and regulations that need to be addressed during the PUF implementation.

The following sections are organized as follows. Sections IV-A and IV-B are about how to avoid the common obstacles designers could encounter in implementing PUF on Vivado. Section IV-C is to propose an improved design for the ring oscillator. Section V explores the fixed relative routing and placement for MP-PUFs in Vivado.

It is notable that the techniques introduced in these sections apply to both the proposed MP-PUF and other existing delay PUFs.

A. Avoiding Design Rule Violations

The basic elements of a PUF or MP-PUF circuit are not ordinary circuits one could use in most FPGA designs. They should be implemented with caution otherwise design rule violations can be easily triggered.

1) Combinatorial Loops: A single ring oscillator (RO) consists of odd number of inverters as shown in Fig. 1 & 4. Since a RO's output is fed back to its input, it forms a combinatorial loop, which needs special permissions in Vivado. To allow a RO combinatorial loop in the synthesis, the following attribute should be added before the instantiation of each RO module:

```
(* ALLOW_COMBINATORIAL_LOOPS = "TRUE" *)
```

To further secure this unit in the *.bit* file generation, a *.tcl* file should be loaded prior to bitstream writing with the following severity downgrade command:

```
set_property SEVERITY {Warning}
    [get_drc_checks LUTLP-1]
```

2) Gated-clock: As shown in Fig. 1 & 4, when "Enable" is set to 1, this RO produces a $1 \to 0 \to 1 \cdots$ sequence similar to a clock signal, which drives the counter. Since this is not a real FPGA clock but a combinational logic, it forms a so-called "gated-clock". Similar situation also exists in the arbiter (DFF) in Fig. 2 & 5. Unwanted gated-clock can cause glitches, increased clock delay, and clock skew. However in MP-PUFs, since the counter and arbiter are designed to be clocked by combinational logic intentionally, it to be given a special permission to pass the synthesis procedure.

To allow this premeditated gated-clock, in the *.tcl* file mentioned previously, the following severity downgrade command should also be added:

```
set_property SEVERITY {Warning}
    [get_drc_checks NSTD-1]
```

B. Preventing Logic Trim

As shown in Fig. 1 that a RO consists of odd number of inverters. However, functionally speaking, this circuit is equivalent to a single inverter in its logic. This in fact is the result of the logic trim in Vivado's post-synthesis optimization.

As it is known a ring oscillator is designed intentionally with multiple inverters. To preserve them from being optimized away, usually the attribute of "KEEP" is suggested. However in Vivado this attribute only saves the redundant inverters in the RTL view, but not in the technology view of post-optimization. Therefore instead, the following attribute should be used before the declaration of each inverter and wire, as well as the RO module instantiation [17]:

```
(* DONT_TOUCH = "TRUE" *)
```

Then in both RTL and technology schematics, every component of the ring is kept.

C. Improving the RO Element in MP-PUF

In this subsection we propose a new architecture of RO elements, in order to improve the accuracy and uniqueness of a RO based MP-PUF.

Although the RO architectures in Fig. 1 & 4 are not of great complexity, there is a critical procedure that the two counters will be compared against each other at a certain point. Yet, it cannot be any moment. Otherwise a counter's value at comparison point might not represent its driving RO's frequency correctly.

Therefore there needs to be a stopwatch to fire the comparison signal at the right timing, namely before overflow occurs to any of the two counters. In other words, the stopwatch's period needs to be smaller than either counter's overflow period, which is determined by the driving RO's frequency. Otherwise before the comparison an overflow might happen to the counter connected with the faster RO. Then when the comparison is made, the overflown counter could end up with a smaller reading than the one connected with the slower RO.

Thus on one hand, the stopwatch's period needs to be set smaller than any of the RO frequencies. However, on the other hand, the stopwatch's period also needs to be large enough to reveal the ROs' frequency difference.

1) The Synchronous Stopwatch: Denote the frequency of the 1^{st} counter's driving RO as f_{RO_0}, the 2^{nd} RO as f_{RO_1}, the FPGA's clock used as the stopwatch as f_{FPGA}, and the size of the counters b-bits. Assume the stopwatch fires the comparison signal at the k^{th} clock cycle, to have the counter readings reflect the RO frequency difference properly, the following statement needs to be true:

$$\left(\frac{1}{f_{RO\_0}} \ll \frac{k}{f_{FPGA}} < \frac{2^b-1}{f_{RO\_0}}\right) \&\& \left(\frac{1}{f_{RO\_1}} \ll \frac{k}{f_{FPGA}} < \frac{2^b-1}{f_{RO\_1}}\right). \quad (6)$$

Note: f_{FPGA} can be acquired from the FPGA manual.

This approach requires extra experiments to find out the RO frequencies, which is possible with proper FPGA programming and an oscilloscope. However it takes unnecessary extra effort. Thus we propose the following second solution.

2) The Asynchronous Stopwatch: Another better approach is to use an asynchronous stopwatch. A MAX value can be preset where $MAX \le 2^b - 1$. The counters will stop incrementing and compare their output when either of them reaches the MAX value. The advantage of this approach is the avoidance of using the FPGA's clock, as well as the complication of measuring and calculating Eq. 6 and RO frequency. The only criterion is that b should be a large enough number to reveal the RO's frequency difference. Then the comparison timing graph will be:

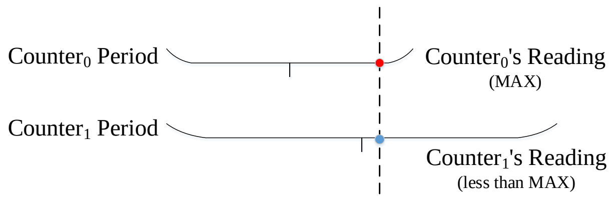

Fig. 7: If Counter$_0$ is driven by a faster RO, then it has a smaller overflow period. It will reach the MAX first, then both counters stop and the comparison is made. Vice versa for Counter$_1$'s case.

With the above improvement, overflow of the counters will never be an issue. In addition, it saves the power of using the system clock and the hassle of identifying each RO's frequency. This design improvement can be applied to both the RO based MP-PUF and ordinary RO PUF.

V. Fixed Placement, Pin, and Relative Routing for MP-PUF Implementation in Vivado

This section still belongs to the discussion of MP-PUF's FPGA implementation problems. However due to its importance and complexity, it is presented as a separate section.

Given a personality and a challenge, a properly functional MP-PUF design should generate different responses on different FPGA boards. This is due to the delay difference among the LUTs on a FPGA. However, if the routing and placement are automatically carried out by the design tools, then the delay bias produced by automated routing will dominate over the LUTs' intrinsic delay. This will result in having the same CRPs on different FPGA boards all the time. Namely the PUFs have lost their uniqueness.

For example, the two ROs (without the PSN box) below are placed and routed automatically by Vivado. The RO on the left obviously has a shorter routing path than the one on the right. Therefore the frequency difference of the two ROs in Fig. 8 is determined essentially by their routing difference which are identical on every FPGA, rather than the LUTs' intrinsic manufacture differences, which are unique on every FPGA. That is to say, this automatic place and route will cause the PUF to lose its uniqueness.

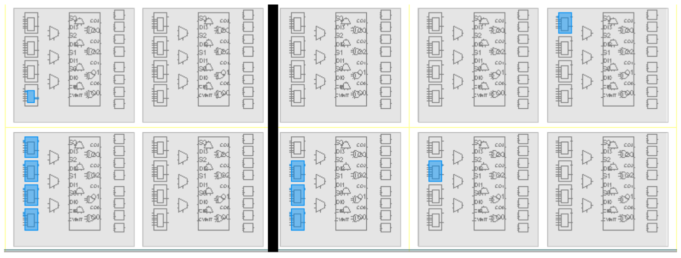

Fig. 8: The RO on the left occupies two adjacent slices, while the RO on the right three slices relatively far away from each other. Thus in all FPGA boards the counter driven by the left RO will always result in a larger reading.

To eliminate the timing difference caused by biased automatic placement and route, all the critical elements of a PUF need to be manually placed and routed identically, so that the LUTs' intrinsic delays will be the only factor affecting the PUF's responses.

We define the term "stage" in Fig. 4 & 5 as:

- RO based MP-PUF: one inverter and its following PSN box;
- Arbiter based MP-PUF: two parallel (upper and lower) MUXes and their following PSN boxes.

To manually place and route the MP-PUF, there are three parameters to configure:

1) Placement: which SLICE (logic slice) and BEL (basic element) a LUT will be placed into;
2) Pins: which pins a LUT will be using;
3) Route: which path two or more LUTs are connected.

It is easier to configure the RO based MP-PUF since it only requires identical place and route on each RO as a whole. While for the Arbiter based MP-PUF, it is required between any two adjacent stages within a MUX chain.

In Vivado, although there is no longer the FPGA Editor, designers can use the *.xdc* file to specify those placements and routes. They can also use XDC macros to generalize the configurations of one RO or MUX chain to others.

A. Fixing the Placement and Pins

A PUF's MUXes and inverters (NOT gates) are all instantiated by the LUTs and referred to as cells in the implementation process. A 2-to-1 MUX or a NAND gate is instantiated by LUT2, and an inverter by LUT1. Knowing that each Xilinx FPGA's SLICE (logic slice) has 4 LUT6s indexed by $\{A, B, C, D\}$ and 8 flip-flops as shown in Fig. 8, we can plan the placement accordingly.

The relative placement can be set through the set_property LOC and BEL keyword in the *.xdc* constraints file [18]. The former locates a LUT in a certain SLICE, and the latter places it in a certain basic element.

Secondly, according to the Vivado constraints manual, different pins of a LUT have different speeds. Especially, LUT pins A6 and A5 are faster than A1 $\sim$ A4. Therefore the pins of a given LUT in all ROs should be locked identically by the set_property LOCK_PINS command.

For example in a RO group of a MP-PUF, for the first and ninth ROs, to place their NAND gates and lock the pins identically, the *.xdc* file can have the following constraints:

```
# Placing each cell into a fixed SLICE
set_property LOC SLICE_X0Y0 [get_cells
    RO_0/NAND]
set_property LOC SLICE_X0Y8 [get_cells
    RO_8/NAND]

# Placing each cell into a relative LUT in
    that SLICE
set_property BEL A5LUT [get_cells RO_0/NAND]
set_property BEL A5LUT [get_cells RO_8/NAND]

# Lock the input pins
set_property LOCK_PINS {I0:A1 I1:A3}
    [get_cells RO_0/NAND]
set_property LOCK_PINS {I0:A1 I1:A3}
    [get_cells RO_8/NAND]
```

Although the NAND gates of these two RO are placed into different SLICEs (X0Y0 and X0Y8), they are fixed into the same basic element: A5 LUT. Their pins are also identical: A1 and A3. This eliminates the delay bias caused by random place and pins. Fig. 9 shows two ROs of a MP-PUF with the same relative placement and pin locking. Each RO has five stages and each stage takes up four BELs (1 for the inverter, and 3 for the PSN box as in Fig. 4).

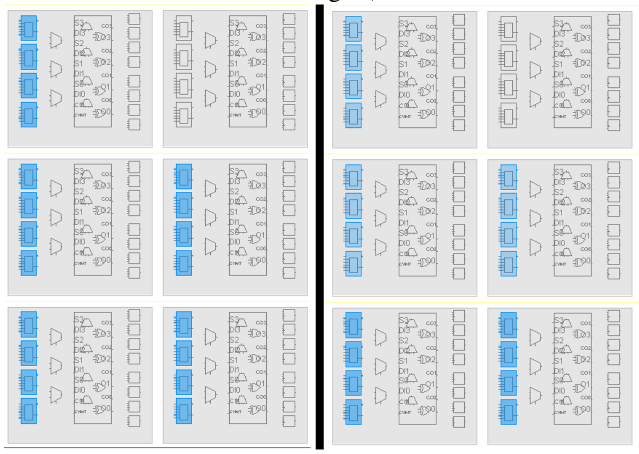

Fig. 9: Each specific LUT of the two ROs not only has the same relative placement, but also uses the same pins.

B. *Identical Relative Routing*

Finally, each LUT's route to the next needs to be fixed identically through the set_property FIXED_ROUTE command in the *.xdc* constraints file. Although one existing route can be read out and copied to another by XDC commands, a proper route that best helps with identical relative routing needs to be manually figured out.

The RO based MP-PUF is easier to implement, since each RO is more of a standalone system and it does not require symmetric design. However, the routing of an Arbiter based MP-PUF is more complicated, since each stage is connected to the next in a staggered manner and so the two routes needs to have as much symmetry as possible. Therefore we will take the Arbiter based MP-PUF for the illustration.

We will use the notations in Fig. 10 to assist the illustration. It shows the connection from a previous stage to the next in a MUX chain of a MP-PUF from Fig. 5.

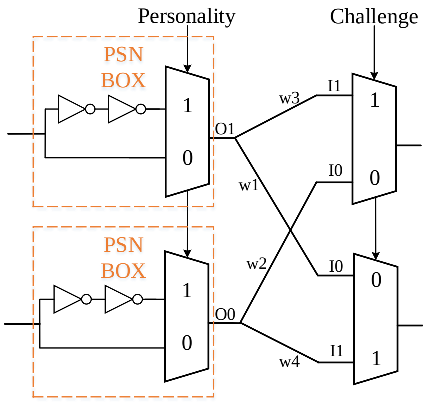

Fig. 10: The routing of w1 should be the same as w2, and w3 the same as w4. I1 of the upper MUX should be the same pin as the I1 of the lower MUX, similarly to I0.

Although an FPGA provides many routing options from one BEL to another, the routing path cannot be "invented" by simply editing the xdc constraints. It has to be explored using the "Assign Routing Mode" [19]. The "Assign Routing Mode" is a FPGA editing mode which allows users to apply their own configurations in post implementation. Since from one cell to another there are only limited resources (paths) for routing, designers will need to manually discover the best BEL, pin, and route for a cell within this limited resource.

Instruction V.1. The 7-step procedure can be followed to manually explore the desired routing of a MP-PUF on Vivado:

1) Firstly, the placement of two consecutive stages of an Arbiter based MP-PUF can be fixed symmetrically by the set_property BEL and set_property LOC commands using xdc constraints. For now the pins will be left unfixed to allow the routing with more flexibility.
2) After running the implementation, in the "Device" tab by selecting "Unroute" on the wires sprung from pin O1, their current automatic routing will be canceled. Then by selecting "Enter assign routing mode" on O1, the designer is able to start manual routing of the wires from O1.
3) The designer will be asked by Vivado if he/she wants to set the routing target/destination or not. In Fig. 10 it refers to pin I0 of the lower MUX and pin I1 of the upper MUX in the next stage. For more routing flexibility, "No load" should be chosen to explore all possible options of routing to any cell and any pin.
4) Then the "Routing Assignment" menu will be offered with the available routing resources for O1 in the form of connection nodes. By selecting a node under "Assigned Node", the next possible nodes and their connections will show up in dotted lines, from which we can set the routing path node by node, and finally to our target cell. If a node and its route is not preferred, one can remove it from the "Assigned Nodes" window to start over.
5) When the routing of w1 from O1 to I0 is accomplished, w2 should be routed in a similar way to have as symmetric a path to w1 as possible, in order to minimize the delay bias caused by the difference of w1 and w2.
6) In addition, once the routing of w1 is determined, the pin for I0 is also set. Therefore w2 should also be routed to the same pin of the upper MUX's I0 as in Fig. 10.
7) After w1, w2, w3, w4 are all routed, their routing path can be read out by:

```
        get_property ROUTE [get_nets
            wirename]
```

which can be generalized to other stages in this PUF by:

```
        set_property FIXED_ROUTE [get_nets
            wirename]
```

Fig. 11 shows that $\{w1, w3\}$ and $\{w2, w4\}$ are symmetrically routed to the next stage.

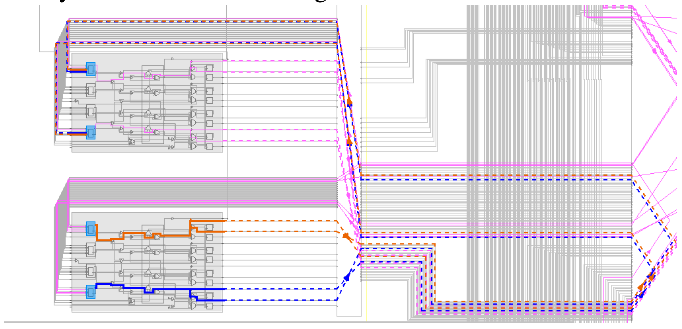

Fig. 11: w1 and w3 are in blue, and w2, w4 in orange.

Fig. 12 shows two ROs of a MP-PUF identically routed.

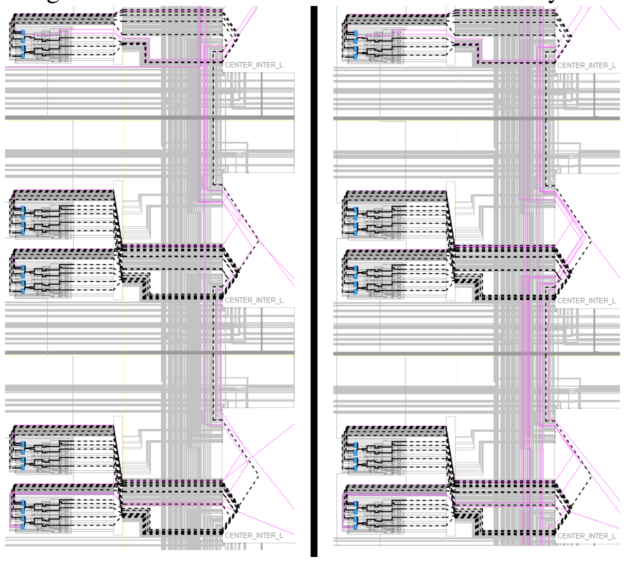

Fig. 12: Both RO's routings (left & right) are in dotted black lines and they are completely identical.

VI. Design Evaluation and Automation

A. Uniqueness Evaluation

Sections IV and V are dedicated to improving the uniqueness of the proposed MP-PUF. Here we provide two sets of data: 1) the MP-PUF PUF uniqueness across FPGA boards, and 2) the MP-PUF personality uniqueness within one FPGA. Both are calculated by dividing the average Hamming distance of their responses by r under the same challenge, except that the former is on different FPGA boards under the same personality, and the latter on the same FPGA but under different personalities.

TABLE II: Uniqueness Evaluation

	PUF Uniqueness		Personality Uniqueness	
	RO based	Arbiter based	RO based	Arbiter based
Without Hash	33.81%	19.13%	25.40 %	23.09 %
With Hash	49.27%	49.31%	49.30%	49.29%

I It can be seen that even with hashing the MP-PUF response, the identically placed and routed MP-PUFs already achieve a satisfying uniqueness.

II For further improvement of uniqueness, the counters of the RO based MP-PUF are also identically placed and routed.

B. MP-PUF Design Automation

Although one can manually make a PUF or MP-PUF on FPGA as most researchers did, it still involves a good amount of work: writing the HDL code, adding the attributes, avoiding design rule violations, finding the best placement, route, and fixing each basic elements of the PUF etc. Therefore we propose an automation tool which takes user's inputs of four parameters: challenge size c, response size r, MP-PUF type (RO or Arbiter based), and FPGA board model, to generate synthesizable Vivado project files automatically.

The automation tool's flowchart is shown below.

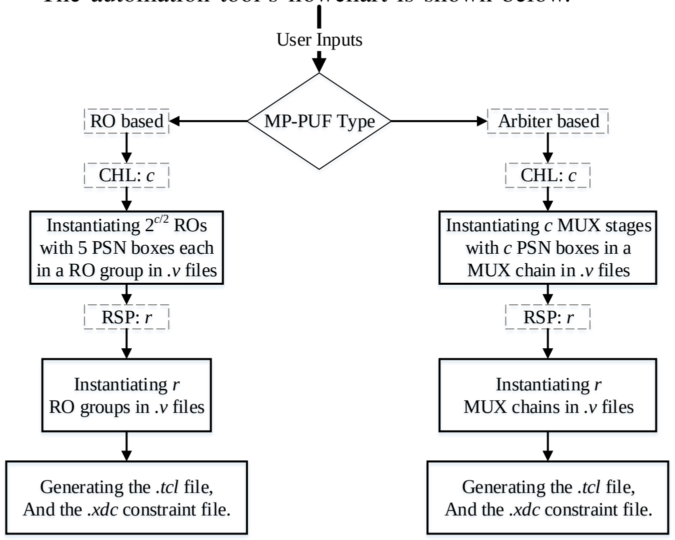
User Inputs
MP-PUF Type
RO based
Arbiter based
CHL: c
CHL: c
Instantiating $2^{c/2}$ ROs with 5 PSN boxes each in a RO group in .v files
Instantiating c MUX stages with c PSN boxes in a MUX chain in .v files
RSP: r
RSP: r
Instantiating r RO groups in .v files
Instantiating r MUX chains in .v files
Generating the .tcl file, And the .xdc constraint file.
Generating the .tcl file, And the .xdc constraint file.

Fig. 13: This tool works on both Linux and Windows environments.

VII. Conclusion

In this paper we explore the design and implementation of a new multi-personality PUF (MP-PUF) on Xilinx Vivado. Unlike previous works where the multi-personality is achieved by altering the challenges only, the proposed design generates a unique PUF circuit for each personality. We also provided sufficient details in implementing and improving the MP-PUF on FPGA through Vivado, which can also be beneficial to other delay based PUF designs.

In addition, we have developed a PUF automation tool to generate the necessary project files in Vivado, simply based on user specified parameters and the pre-loaded templates. This tool makes MP-PUF accessible to researchers who plan to use it for their researches but lack of time or sufficient knowledge of HDL and FPGA.

References

[1] B. Gassend *et al.*, "Silicon physical random functions," *Proceedings of the Computer and Communications Security Conference*, 2002.

[2] B. Gassend, M. V. Dijk, D. Clarke, E. Torlak, S. Devadas, and P. Tuyls, "Controlled physical random functions and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[3] Intrinsic, *Butterfly PUF based secure key management for FPGA*, 2016.

[4] C. Helfmeier *et al.*, "Cloning physically unclonable functions," *Hardware-Oriented Security and Trust*, 2013.

[5] D. Nedospasov *et al.*, "Invasive puf analysis," *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2013.

[6] E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," *Proceedings of the 44th annual Design Automation Conference*, 2007.

[7] A. Maiti, I. Kim, and P. Schaumont, "A robust physical unclonable function with enhanced challenge-response set," *IEEE Transactions on Information Forensics and Security*, 2012.

[8] L. Jae W, S. Devadas *et al.*, "A technique to build a secret key in integrated circuits for identification and authentication applications," *VLSI Circuits*, 2004.

[9] D. Lim, S. Devadas *et al.*, "Extracting secret keys from integrated circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005.

[10] R. Forr, "The strict avalanche criterion: spectral properties of boolean functions and an extended definition," *Proceedings on Advances in cryptology*, 1990.

[11] M. Majzoobi, F. Koushanfar, and M. Potkonjak, "Lightweight secure pufs," *Computer-Aided Design*, 2008.

[12] A. Mahmoud *et al.*, "Combined modeling and side channel attacks on strong pufs," *IACR Cryptology ePrint Archive*, 2013.

[13] J. Delvaux and I. Verbauwhede, "Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise," *Hardware-Oriented Security and Trust*, 2013.

[14] M. Soybali, B. Ors, and G. Saldamli, "Implementation of a puf circuit on a fpga," *New Technologies, Mobility and Security (NTMS)*, 2011.

[15] S. Morozov, A. Maiti, and P. Schaumont, "An analysis of delay based puf implementations on fpga," *ARC*, 2010.

[16] S. Khoshroo, "Design and evaluation of fpga-based hybrid physically unclonable functions," 2013.

[17] Xilinx, "Vivado design suite user guide - synthesis," 2013.

[18] ——, "Vivado design suite user guide - using constraints," 2015.

[19] ——, "Vivado design suite user guide - implementation," 2013.