TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

Master's Thesis

submitted by

Benedikt Tutzer

Registration Number 01225461

A Practical Design Methodology for Hardware Security Primitives

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria, February 2022

Study code:	066 504
Field of study:	Embedded Systems
Supervisors:	Christian Krieg
	Axel Jantsch

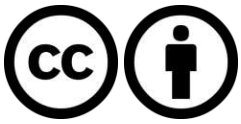
If you find this work useful, please cite it using the following BibTeX entry:

```
@Thesis{Tutzer2022,
 type        = {Master's Thesis},
 author      = {Benedikt Tutzer},
 title       = {A Practical Design Methodology for
    Hardware Security Primitives},
 school      = {Vienna University of Technology (TU Wien)},
 year        = {2022},
 address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
 month       = {3},
}
```

Contact us:

benedikt.tutzer@tuwien.ac.at

christian@drkrieg.at



Acknowledgements

First and foremost I want to thank my fiancé Yasmin. During the process of this work you heard of exciting developments as well as setbacks and drops in motivation. You managed to put a smile on my face even after the hardest of days and motivated me to go on. I am beyond grateful for your warm and loving support.

Vielen Dank auch meinen Eltern Monika und Leonhard, die mein Studium erst ermöglichten und auch meinen Geschwistern Judith, Mirjam und Philipp.

Sofia, Consti, Philipp, Moses, Maxi, Michael, and Jona. For all the support in the office. The constructive and destracting conversations. For all the shared tee, beer, and frustration. For the Büro- and Bierorunde that have become a welcome routine. For all the graduation parties of the ones that finished before me and the anticipation to the ones ahead. For every laugh, every screw, every lunch, every ball of terror, every dessert, every cat picture, and every candy. I think this work could not have been completed without your help and esteemed opinions. From the bottom of my heart, thank you.

Thank you to my friends back in South Tirol as well as Vienna or wherever you may be.

Thank you to all my fellow hockey players for all the practices and matches.

Thank you to my supervisors Christian Krieg for all his support and patience and Axel Jantsch for providing a welcoming and productive environment.

Abstract

In the past decades computer networks started playing major roles in peoples social, professional, and official lives. Massive amounts of private and confidential data is sent through networks on a daily basis. We need to protect that data from unauthorized and potentially malicious access. This requires authentication and encryption. Authentication and encryption are usually implemented in software, but hardware can be used to improve speed and security. An upcoming technology in cloud-environments are *field-programmable gate arrays (FPGAs)* and we use that architecture to showcase our work.

In the context of digital circuitry, physical unclonable functions and random number generators are a promising technology for authentication and key-generator for encryption algorithms respectively. Most implementations of physical unclonable functions and random number generators leverage timing phenomena in the circuits. The effectiveness of the designs depends on the timing of the circuits. Often, the designs use timing phenomena that provoke metastabilities or that violate setup- and hold-time requirements. We find that common design tools try hard to avoid the phenomena that we want to leverage, as they lead to non-determinable states that are usually undesired. The implementation must therefore be done manually by a skilled engineer.

Manual implementation, i.e., placement and routing, with strict timing requirements is a hard and time-consuming task. It requires trial and error and a deep understanding of the underlying architecture. We propose a novel timing constraint tailored to security primitives and a placement and routing algorithm that respects our novel timing constraint. We name the constraint *DELAY_GROUP*. It annotates nets that need to have equal delay. Our novel timing constraint can be defined directly in the design in hardware description languages or during synthesis in *tool command language (TCL)*. Our algorithm interacts with state-of-the-art synthesis tools via TCL. First, the algorithm places the design in a fashion to keep cells that are connected by nets within the same *DELAY_GROUP* at equal distance on the FPGA. Second, the algorithm guides the synthesis tool to finding routes between these cells that have equal delay. The algorithm substitutes the manual effort needed during the implementation of security primitives. We thereby reduce the amount of time a skilled engineer needs to spend on the implementation by over 98%, from two months to under six hours.

We conduct experiments on the Xilinx Virtex UltraScale+ VCU118 development board. Our algorithm guides the vendor-specific synthesis software Vivado to a suitable implementation using TCL. The development board connects to a host computer using *universal synchronous receiver/transmitter (UART)* and outputs oscillating signals used in the random number generator design to a high speed oscilloscope using a low-voltage differential signaling link. As a use-case design we choose the self timed ring based *true random number generator (TRNG)*, a high performance *random number generator (RNG)* design that meets the *AIS 20/31* criteria. We first analyze the predicted timing calculated by static timing analysis. This confirms our algorithm generates implementations that meet our timing constraint. Then, we measure

the real timing using the oscilloscope to attest the functionality of the true random number generator. Lastly, we check the final output, the generated random numbers, using stochastic test suites.

The experiments show that our algorithm matches the implementation performance of a skilled engineer in less time without manual intervention of an engineer on the example of a self timed ring true random number generator. Our algorithm manages to keep the relative delay difference within all individual delay groups under 30%. The implemented true random number generators pass the *FIPS 140-2 rngtest* and the *NIST 800-22* stochastic test suites.

We conclude that our algorithm is suitable to implement security primitives with much less effort compared to the state-of-the-art synthesis flow. This makes the implementation cheaper by cutting down on design costs, more reliable by removing the human from the design loop, and more easily portable as our algorithm reduces the effort of moving to a different hardware target. This enables low-budget projects to deploy reliable security measures within their devices. We predict our work to make basic security primitives more accessible to engineers.

Kurzfassung

Computernetzwerke spielen seit den letzten Jahrzehnten eine große Rolle im sozialen Leben, in der Arbeit sowie bei Amtswegen. Unvorstellbare Mengen an privaten und vertraulichen Inhalten werden täglich von solchen Netzwerken übertragen. Diese Inhalte müssen vor unbefugten und vermeintlich böswilligen Zugriffen geschützt werden. Dafür sind Authentifizierung und Verschlüsselung notwendig. Diese werden üblicherweise in Software implementiert, Hardware kann jedoch verwendet werden, um die Funktionen zu beschleunigen und sicherer zu gestalten. Eine aufkommende Methode um Hardware-Security in Cloud-Umgebungen zu implementieren sind *Field-Programmable-Gate-Array (FPGAs)*. Wir verwenden FPGAs zum Veranschaulichen unserer Arbeit.

Bei digitalen Schaltungen sind Physical-Unclonabe-Functions für die Authentifizierung, und Zufallsgeneratoren zum Erzeugen von Schlüsseln für Verschlüsselungsalgorithmen, beliebte Technologien für hardwaregestützte Security-Systeme. Die meisten Implementierungen von Physical-Unclonable-Functions und von Zufallszahlengeneratoren reizen Phänomene aus, die von sehr kurzen zeitlichen Effekten stammen. Diese Effekte, wie z.B. Metastabilitäten, verhindern die Bestimmbarkeit des Verhaltens der Systeme. Die meisten Entwicklungswerkzeuge verhindern das Auftreten solcher Effekte deshalb systematisch. Sie müssen durch gezieltes manuelles Platzieren und Routen der Netze provoziert werden. Die Implementierung muss also von einem erfahrenen Entwickler in mühsamer manueller Arbeit durchgeführt werden.

Diese Arbeit ist sehr zeitaufwendig und damit mit hohen Kosten verbunden. Iterativ müssen Lösungen ausprobiert und so lange verbessert werden, bis das Zeitverhalten zufriedenstellend ist. Dies verlangt viel Zeit und ein tiefes Verständnis der verwendeten Architektur. Wir präsentieren in dieser Arbeit eine Methode das Zeitverhalten auf eine Weise zu beschränken, die auf Security-Primitive zugeschnitten ist. Dazu entwickeln wir einen Algorithmus, der aktuelle Entwicklungswerkzeuge ansteuert und diesen Prozess somit automatisiert. Wir nennen die Zeitverhalten-Beschränkung *DELAY_GROUP*. Die Beschränkung notiert Verbindungsnetze welche die gleiche Zeitverzögerung beim Signaltransport aufweisen sollen. Sie kann sowohl direkt in der Hardware-Beschreibung oder während der Synthese in *tool command language (TCL)* angegeben werden. Unser Algorithmus interagiert dann mit aktuellen Synthese-Werkzeugen über TCL. In einem ersten Schritt werden Logikzellen, die mit Netzen in derselben *DELAY_GROUP* sind, äquidistant zueinander platziert. Dann werden die entsprechenden Netze so geleitet, dass sie dieselbe Zeitverzögerung aufweisen. Unser Algorithmus ersetzt den manuellen Aufwand der zum Implementieren von Security-Primitiven notwendig ist. Dadurch wird die Zeit, die ein erfahrener Entwickler in die Implementierung investieren muss um 98% reduziert, von zwei Monaten auf unter sechs Stunden.

Wir führen Experimente auf einem Xilinx Virtex UltraScale+ VCU118 Entwicklungsboard durch. Unser Algorithmus interagiert hier mit dem vom Hersteller zur Verfügung gestellten Synthesewerkzeug Vivado. Das Entwicklungsboard

ist über *universal synchronous receiver/transmitter (UART)* an einen Host-Computer und über eine differentielle Niederspannungsleitung mit einem Oszilloskop verbunden. Als Beispieldesign verwenden wir den self-timed-ring basierten Zufallszahlengenerator, ein hochperformanter Zufallszahlengenerator, der den *AIS 20/31* Richtlinien entspricht. Um zu überprüfen, dass das Zeitverhalten unserer Beschränkung entspricht, analysieren wir das vorhergesagte Zeitverhalten mithilfe von statischer Analysen. In einem zweiten Schritt messen wir das tatsächliche Zeitverhalten mit dem Oszilloskop, um die Funktionalität zu bestätigen. Zuletzt analysieren wir die produzierten Zufallszahlen mit stochastischen Testbatterien.

Wir zeigen anhand des Beispiels eines self-timed-ring Zufallszahlengenerators, dass unser Algorithmus die Implementierung eines erfahrenen Entwicklers in weniger Zeit und ohne menschliches Einschreiten erledigen kann. Unser Algorithmus hält die relative Zeitverzögerungsdifferenz zwischen allen Netzen unter einer definierten Schranke von 30%. Der so implementierte Zufallszahlengenerator produziert Zufallszahlen, die die *FIPS 140-2 rngtest* und die *NIST 800-22* stochastischen Testbatterien bestehen.

Der Algorithmus kann also verwendet werden, um Security-Primitive wie Authentifizierung und Verschlüsselung mit wenig Aufwand in Hardware zu implementieren. Das macht die Implementierung durch Verminderung der benötigten Arbeitszeit kostengünstiger, durch das Entfernen des Menschen in der Implementierungs-Kette zuverlässiger und durch die Automation leichter portierbar. Das erlaubt selbst Projekten mit kleinem Budget den Einsatz solcher Schutzfunktionen in deren Geräten. Wir erwarten, dass unser Algorithmus dazu beiträgt, die Einstiegsschwelle für Schutzfunktionen in digitalen Schaltkreisen zu verringern.

Contents

Abstract 5

Kurzfassung 7

Glossary 16

1 Introduction 17

1.1 Problem statement . . . 17
1.2 Motivating example . . . 18
1.3 Proposed design flow . . . 19
1.4 Limitations . . . 19
1.5 Research questions . . . 20

2 Preliminaries 21

2.1 Security primitives in reconfigurable hardware . . . 22
2.1.1 Random number generators . . . 23
2.1.2 Physical unclonable functions . . . 28
2.2 Timing requirements . . . 31
2.2.1 Static timing analysis . . . 32
2.2.2 Typical timing requirements in electronic design automation . . . 32
2.2.3 Timing requirements for security primitives . . . 34
2.3 Use-case example . . . 34
2.3.1 The self-timed-ring-based true random number generator . . . 35
2.3.2 Timing requirements of the self timed ring . . . 36
2.3.3 Entropy assessment . . . 38
2.3.4 Certificates . . . 39

3 Proposed algorithm 41

3.1 State of the art . . . 41
3.2 Design preparations . . . 43
3.3 Placement step . . . 45
3.4 Routing step . . . 47

4 Experiments **51**
4.1 Selection of Device under Test . . . 51
4.2 Setup . . . 52
4.3 Methodology . . . 52
4.4 Expectations . . . 55
4.5 Limitations . . . 55
4.5.1 Collection of random numbers . . . 55
4.5.2 Metering . . . 57
4.5.3 Clock interfaces and connectors . . . 57
4.6 Results . . . 57
4.6.1 Design flow automation . . . 57
4.6.2 Static timing analysis . . . 58
4.6.3 Clock behavior . . . 59
4.6.4 Stochastic tests . . . 69

5 Discussion **73**
5.1 Design flow automation . . . 73
5.2 Static timing analysis . . . 75
5.3 Clock behavior . . . 75
5.4 Stochastic tests . . . 76

6 Conclusion **79**

A Additional experimental results **81**

Bibliography **87**

List of Tables

2.1 Evaluation of random number generators on FPGAs 26
2.2 Example arbiter physical unclonable function 30
2.3 Truth table of the Mueller C-element 35

4.1 Implementation-time comparison between manual placement and routing and our proposed algorithm. . 58
4.2 Results of the stochastic tests 71
4.3 Results of the stochastic tests 72

A.1 Results of the stochastic tests 82
A.2 Results of the stochastic tests 83
A.3 Results of the stochastic tests 84

List of Figures

1.1 *True random number generator (TRNG)* cores presented in O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016 compared by power, area, and performance. 19

1.2 Comparison of the current state-of-the-art design flow for field-programmable gate arrays and our proposed flow to facilitate the placement and routing of security primitives. 20

2.1 Software encapsulation 22

2.2 Entropy extraction from clock-jitter 25

2.3 Schematic of a ring oscillator and three true random number generator circuits 27

2.4 Arbiter *physical unclonable function (PUF)* circuit 29

2.5 Arbiter physical unclonable function timing example 30

2.6 *Ring oscillator (RO)* PUF circuit 31

2.7 Example synchronous circuit and the result of a static timing analysis on said circuit. 32

2.8 Possible timing diagram of the example synchronous circuit in Figure 2.7, showing hold-time t_H and setup time t_S. 33

2.9 Self timed ring timing 36

2.10 Self timed ring circuit 37

3.1 Comparison of the current state-of-the-art design flow for field-programmable gate arrays and our proposed flow to facilitate the placement and routing of security primitives. 42

3.2 Design hierarchy of Listing 3.1 44

3.3 Circular placement of the combinatorial loop in Listing 3.1 45

3.4 Example delay values of nets A, B, C and D on a time bar 48

4.1 Laboratory setup 53

4.2 Picture of the laboratory setup 54

4.3 Execution of experiments 56

4.4 Results of the static timing analysis 59

4.5 Analysis of the implemented ring oscillator consisting of 53 inverters. 63

4.6 Analysis of a 64-stage self-timed-ring with one missing token implemented by Vivado (without constraints). 64

4.7 Analysis of a 64-stage self-timed-ring with one missing token manually implemented by a skilled engineer. 65

4.8 Analysis of a 64-stage self-timed-ring with one missing token implemented by our proposed algorithm. . 66
4.9 Analysis of a 256-stage self-timed-ring with 7 missing tokens implemented by Vivado (without constraints). 67
4.10 Analysis of a 256-stage self-timed-ring with 7 missing tokens implemented by our proposed algorithm. . 68

Glossary

BRAM Block RAM. 51, 52, 55, 57

COSO-TRNG Ring-oscillator-coherent sampling based *true random number generator (TRNG)*. 18, 25, 26

CPU Central processing unit. 21

CSP Cloud Service Provider. 18, 23, 24

ECC error correction code. 22

EDA Electronic design automation. 19, 31, 33, 34, 36

EDIF Electronic Design Interchange Format. 43

ERO-TRNG Ring-oscillator-based elementary TRNG. 18, 25–27

FIPS Federal Information Processing Standard. 39, 69

FMC *field-programmable gate array (FPGA)* Mezzanine Card. 57

FPGA Field-programmable gate array. 5, 15, 18, 19, 23–28, 31, 32, 34, 47, 52, 55, 57, 73, 74, 79, 80

GSL GNU scientific library. 39

HDL Hardware description language. 18, 19, 41, 43, 73

I/O input/output. 23, 80

IC Integrated circuit. 21, 31, 58

IP-core intellectual property core. 23, 28, 41, 43

LUT Lookup table. 26, 27

LVDS Low voltage differential signaling. 52, 57, 75

MURO-TRNG Multi-ring-oscillator-based TRNG. 18, 25–28

NIST National Institute of Standards and Technology. 39, 40

PCIe Peripheral Component Interconnect Express. 57, 77

PLL Phase-locked loop. 16, 18, 25

PLL-TRNG Coherent-sampling-based TRNG using *Phase-locked loops (PLLs)*. 18, 25, 26

PnR Placement and routing. 17, 19, 28, 31, 33, 34, 41–43, 51, 53, 55, 57, 58, 60, 73, 74, 79, 80

PRNG Pseudo random number generator. 24

PUF Physical unclonable function. 13, 20, 22, 23, 28–31, 34, 79

RNG Random Number Generator. 5, 22–24, 28, 34, 35, 38, 39, 41, 51, 52, 69

RO Ring oscillator. 13, 27, 28, 31, 35, 51–54, 57, 59, 60

RO-PUF Ring oscillator based physical unclonable function. 28, 30

SDRAM Synchronous dynamic RAM. 57

SMA SubMiniature version A. 52, 57, 75, 76, 80

STA static timing analysis. 19, 32–34, 48, 51, 53, 57, 79

STR Self timed ring. 16, 18, 22, 26, 28, 35, 36, 41, 51–53, 55, 57–63, 69, 73, 75, 76, 79, 80

STR-TRNG *Self timed ring (STR)*-based TRNG. 18, 22, 26, 28, 34–36, 41, 43, 45, 51–53, 57–60, 63, 69, 73, 76, 79, 80

TCL Tool command language. 5, 7, 43, 45, 47, 49, 53

TERO-TRNG Transition-effect-ring-oscillator-based TRNG. 18, 19, 25, 26

TRNG True random number generator. 5, 13, 15–20, 22, 25–28, 31, 39–41, 51, 53, 55, 73, 74, 79

UART Universal asynchronous receiver/transmitter. 5, 8, 18, 52, 53, 55

VHDL VHSIC hardware description language. 43

WELL Well Equidistributed Long-period Linear. 24

XDC Xilinx Design Constraints. 19

XOR exclusive-or. 27, 32, 33, 35, 36, 51, 75, 76

Chapter 1

Introduction

This chapter elaborates the need for hardware security in digital circuitry and why hardware security primitives are expensive to implement (Section 1.1). We give a scenario where hardware security is of particular importance by scheming a motivating example (Section 1.2). In Section 1.3 we propose a novel design flow for the automated implementation of hardware security primitives. The novel design flow is suitable for a wide range of hardware security primitives, but some designs still cannot be implemented automatically as explained in Section 1.4. We condense our goals into two research questions in Section 1.5.

1.1 Problem statement

In cryptography, high-quality cryptographic keys are needed to maintain a sufficient degree of confidentiality. High-quality keys mainly depend on highly unpredictable random numbers. Because it is hardly possible to derive truly unpredictable random numbers in software systems, it is necessary to exploit unpredictable user input, uncontrollable network communication or physical noise as sources of entropy. In digital systems, examples for physical random sources include jittery signals and circuit metastability. To exploit such physical random sources, a digital design must meet tight physical constraints. These usually constrain the *placement and routing (PnR)* steps in the design and verification flow to meet strict timing requirements. In state-of-the-art design flows for physical random sources, this is done manually, requires manual PnR skills, and is usually a very time-consuming process. Also, resulting implementations are architecture-dependent. The same process has to be carried out for each and every target architecture.

The state-of-the-art design process for physical random sources also has severe implications on security. Because manual PnR is necessary for implementing *true random number generators (TRNGs)*, a potential user of a cryptosystem cannot synthesize a high-level description for a given high-security application on-the-fly. This means that the user has to use pre-built TRNG implementations, and has to trust these implementations. However, to maintain a high level of security the user must have full control over the physical random source which includes being able to instantiate a TRNG core at runtime in reconfigurable environments. However, as the target architecture is not necessarily known at design time, TRNG instantiation requires to fully automate PnR under the given physical constraints that enable to exploit the physical random source. This is not feasible yet with existing design flows.

This work elaborates a method to enable a fully-automatic synthesis of high-level designs under given physical constraints. We propose a set of physical parameters and a methodology to constrain these parameters at *hardware description language (HDL)* level. We show that using our methodology, (1) we significantly reduce the design-time of TRNGs, and (2) we significantly increase the security level of a system that runs a high-security application.

1.2 Motivating example

Deploying an application to the cloud is a prominent technique to improve and reduce time to market while keeping costs low, but it also introduces a higher demand for security. This is because the deployer is not in physical control of the hardware anymore. As the *cloud service provider (CSP)* cannot be trusted, the hardware provided by the CSP cannot be trusted either. Developers are responsible for encrypting all confidential data to protect it not only from third party access, but also from access from the CSP. This is particularly hard, because the CSP has access to the hardware.

Most encryption algorithms rely on random numbers, so the need for secure data protection implies a dependency on random numbers. In a cloud setting, random numbers can be obtained from:

- A dedicated hardware TRNG
- A software TRNG that extracts entropy from environmental noise such as device drivers and network traffic

Both these sources are in control of and can be manipulated by the cloud service operator meaning they cannot be trusted.

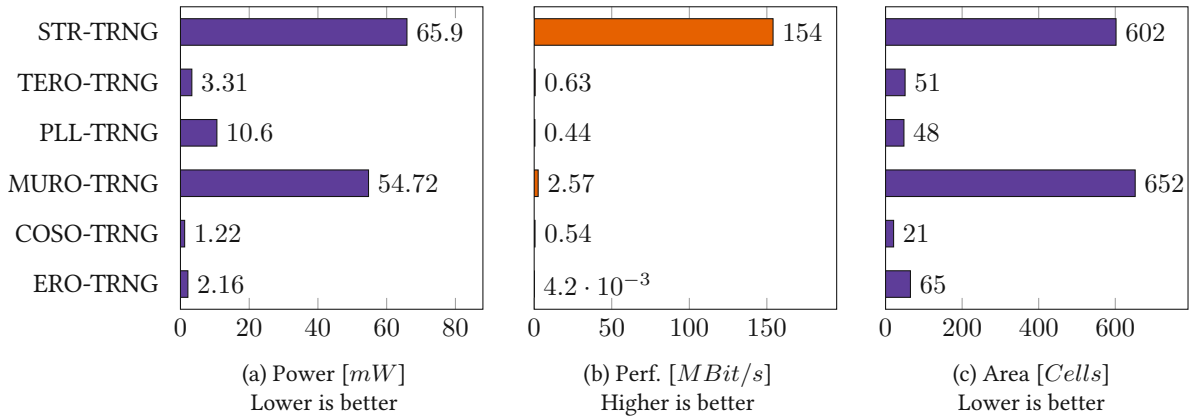
To establish a root of trust we propose using a user-controlled hardware TRNG by instantiating a hardware TRNG core in a *field-programmable gate array (FPGA)* on the remote machine. As discussed in Section 1.1, this is no trivial task and requires a lot of design time. Therefore, we propose a novel design flow for hardware TRNGs to accelerate development of the core.

We instantiate a TRNG core on a Xilinx *Virtex Ultrascale+* FPGA and connect it via a *universal synchronous receiver/-transmitter (UART)* interface to a host PC. The UART interface can be used to configure the TRNG and retrieve bytes of random data. We show the improved performance using stochastic tests on the generated random data that is transferred to the PC.

To showcase the usability of the proposed design flow we synthesize a *self timed ring (STR)-based TRNG (STR-TRNG)* core in different configurations. In a comparison of digital-only TRNGs published by Petura et al.[1] it was the best performing core but also one of the hardest to implement.

Figure 1.1 shows the cores presented by Petura et al.[2] in the Area-Power-Performance design space. The *ring-oscillator-based elementary TRNG (ERO-TRNG)*, *ring-oscillator-coherent sampling based TRNG (COSO-TRNG)*, *coherent-sampling-based TRNG using* Phase-locked loops (PLLs) *(PLL-TRNG)* and *transition-effect-ring-oscillator-based TRNG (TERO-TRNG)* are all low-power, low-area but also low-performance cores. The *multi-ring-oscillator-based TRNG (MURO-TRNG)* performs a little better at the cost of significantly higher area and power consumption. The STR-TRNG offers by far the highest performance by using less area than the MURO-TRNG and consuming only 20% more power.

[1] O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016.
[2] Ibid.



(a) Power $[mW]$ Lower is better

(b) Perf. $[MBit/s]$ Higher is better

(c) Area $[Cells]$ Lower is better

Figure 1.1: TRNG cores presented in O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016 compared by power, area, and performance. More details can be found in Table 2.1.

1.3 Proposed design flow

Figure 1.2 shows the state-of-the-art (left) and the proposed design flow (right). The cumbersome manual PnR task is replaced by a PnR algorithm that takes into account the timing constraints defined on HDL or *Xilinx Design Constraints (XDC)* level. It removes the human from the loop providing a fully autonomous synthesis flow without the need for intervention.

1.4 Limitations

The proposed flow has a few shortcomings. It can only generate cores that have no timing dependencies that cannot be checked with *static timing analysis (STA)*. Also, one still needs to select an appropriate TRNG design and, therefore, understand the TRNG designs. To improve that, we suggest a few improvements to the flow to further aid designers in integrating a TRNG in their circuits with minimal knowledge:

- **Generic TRNGs**

  Instead of constraining physical properties on HDL-level, one could implement a TRNG interface which creates a TRNG based on area / power / performance requirements. The designer would only need to specify the maximum available area and power for the core and the desired performance and, if possible, the tools would automatically select and constrain a core based on that.

- **Consider hardware feedback.**

  The proposed flow can be used to implement TRNGs designs for which the quality of the TRNG is dependent on its routing. The routing dependencies however must be known at design time and cannot change from device to device. A resulting design would then produce random numbers of similar quality on multiple devices within the same family. This is not the case for all designs. E.g., the TERO-TRNG from Petura et al.[3] needs to be placed and routed for each single device individually. This could be supported by incorporating the target device into the design-cycle. First, constraints are defined based on some heuristic. The design would need to be synthesized based on those constraints and programmed onto the target FPGA. The *electronic design automation (EDA)* software would then need to monitor the desired runtime properties and adapt the constraints accordingly. This

[3] Ibid.

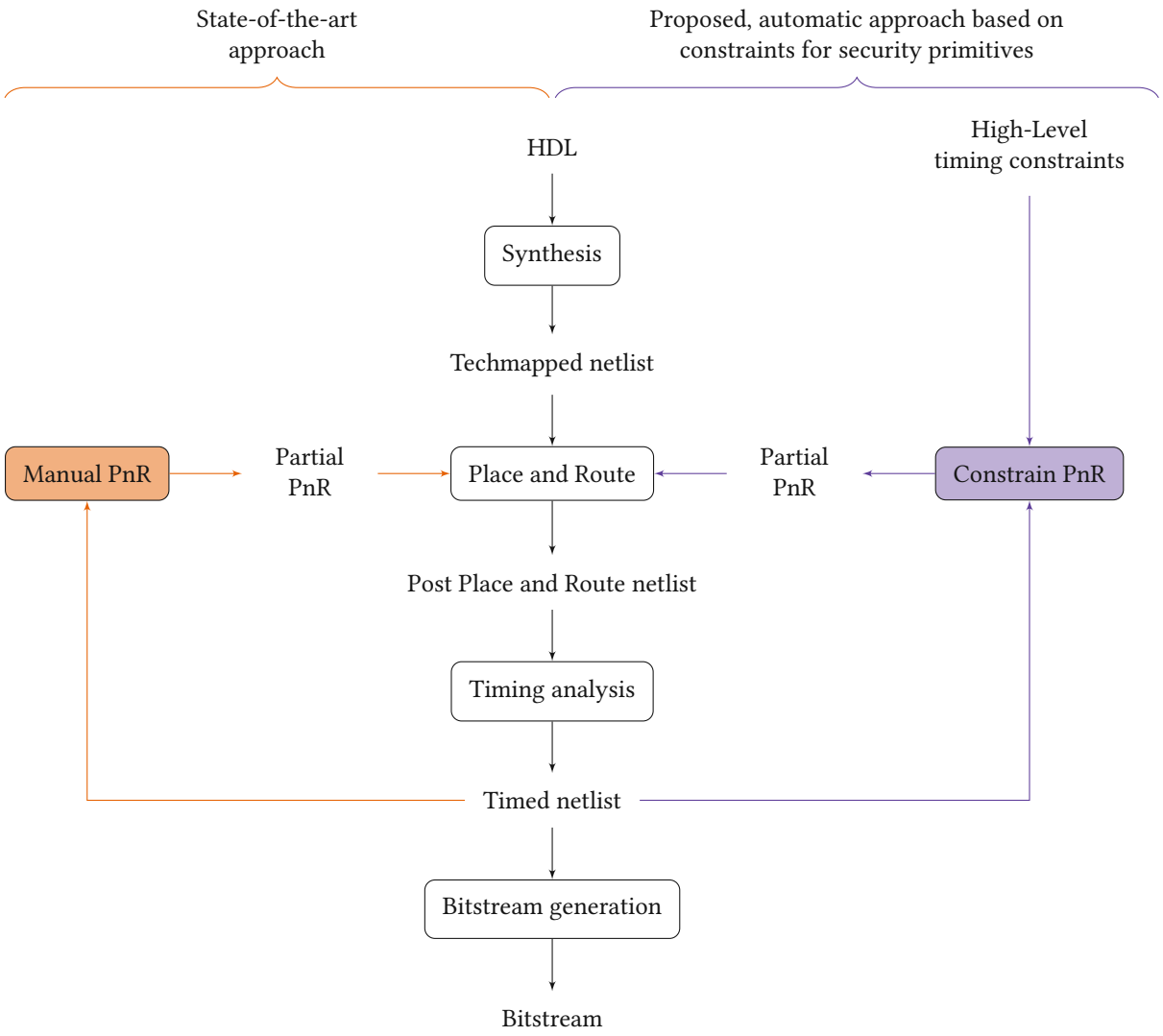


Figure 1.2: Comparison of the current state-of-the-art design flow for field-programmable gate arrays (left) and our proposed flow to facilitate the placement and routing of security primitives (right). We introduce high-level timing constraints specific to security primitives. We can then avoid the time-consuming manual placement and routing step with our proposed algorithm that respects these timing constraints.

hardware in the loop cycle could then be continued until all requirements are met or the synthesis-tool is certain that no such configuration exists.

1.5 Research questions

We aim to answer the following research questions:

1. Do common designs of the security primitives '*true random number generator (TRNG)*' and '*physical unclonable function (PUF)*' possess common general properties related to their physical implementation?

2. To what extent can we automate the implementation of security primitives in reconfigurable digital circuits and how much development time can be saved by automation?

Chapter 2

Preliminaries

Cybersecurity is an omnipresent property in today's computer-aided world. With governments starting to regulate how businesses must handle confidential data, it has evolved from best-practice to unavoidable duty.[1] All businesses, no matter how little or large, need to protect all processed customer data from unprivileged access. Moving data across public networks is risky, so all data flow needs to be secure.

Data flow is usually modeled in layers. These layers commonly include,[2] but are not limited to,[3]

1. The *application* layer,
2. The *presentation* layer,
3. The *session* layer,
4. The *transport* layer,
5. The *network* layer,
6. The *data-link* layer and
7. The *physical* layer.

This work focuses on Layer 7. It was shown that hardware security primitives are susceptible to malicious attacks[4] and to Trojan insertion in the *integrated circuit (IC)* supply chain.[5] Researchers found widely used *central processing units (CPUs)* susceptible to side-channel attacks.[6]

[1] European Commission. *Art. 32 GDPR: Security of processing*. Ed. by European Commission. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679. Apr. 27, 2016. (Visited on 05/10/2021).

[2] J. Day and H. Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (Dec. 1983).

[3] M. Yildiz et al. "A Layered Security Approach for Cloud Computing Infrastructure". In: *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. ISSN: 2375-527X. Dec. 2009; J. K. Barr et al. "Layered security in digital watermarking". en. US8190901B2. May 29, 2012.

[4] M. Rostami, F. Koushanfar, and R. Karri. "A Primer on Hardware Security: Models, Methods, and Metrics". In: *Proceedings of the IEEE* 102.8 (Aug. 2014).

[5] C. Krieg, C. Wolf, and A. Jantsch. "Malicious LUT: A stealthy FPGA Trojan injected and triggered by the design flow". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ISSN: 1558-2434. Nov. 2016; C. Krieg et al. "Toggle MUX: How X-optimism can lead to malicious hardware". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2017.

[6] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2019; M. Lipp et al. "Meltdown". In: *arXiv:1801.01207 [cs]* (Jan. 2018). arXiv: 1801.01207.

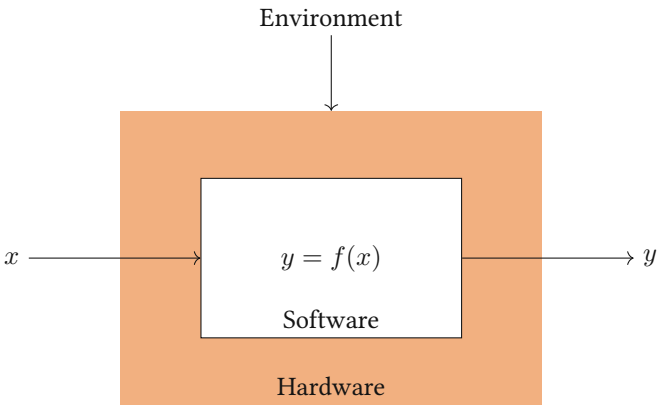


Figure 2.1: Software encapsulation. Designers usually design Hardware to absorb any environment impact within an operating range. Software is thereby protected from environmental change and can run deterministically.

This chapter explores the current state-of-the-art in hardware security primitives (Section 2.1). We focus on *true random number generators (TRNGs)* (Section 2.1.1) and *physical unclonable functions (PUFs)* (Section 2.1.2). We show that hardware security primitives are dependent on particular timing requirements that are hard to satisfy (Section 2.2). We select a security primitive from the class of TRNGs, the *self timed ring (STR)-based TRNG (STR-TRNG)*, as a use-case example (Section 2.3).

2.1 Security primitives in reconfigurable hardware

Software needs hardware to run on. Hardware encapsulates software by absorbing environmental noise, as seen in Figure 2.1. This makes software deterministic. Software will only change its behavior when either the perceivable environment or the input data change.

This is generally considered a good thing, as determinability improves software stability. However, because the input data and the software-environment it operates in can be controlled, software is encapsulated in a deterministic world without randomness.

Things are different for hardware, as the environment is much harder to control. Examples for environmental variations include temperature and power fluctuations, cosmic rays, and process variations; there are numerous factors that affect the internal of a hardware circuit. Hardware designers take great measures to prevent environmental changes to be visible to the user on it's output. This includes wide voltage[7] and timing tolerances[8] in logic circuits and *error correction code (ECC)* techniques.[9]

On the other hand, hardware also affects the environment. Electronic and magnetic fields can be used to observe internal hardware signals. This is called a side channel attack.

With these variations in mind, hardware can be designed:

1. for non-deterministic behavior, i.e., *random number generators (RNGs)*, for

[7] JEDEC. *Interface Standard for Nominal 3 V/3.3 V Supply Digital Integrated Circuits.* Tech. rep. JEDEC, June 2006; JEDEC. *2.5 V ± 0.2 V (Normal Range) and 1.8 V – 2.7 V (Wide Range) Power Supply Voltage and Interface Standard for Nonterminated Digital Integrated Circuits.* Tech. rep. JEDEC, June 2006.

[8] Xilinx. *UG903 Vivado Design Suite User Guide: Using Constraints.* Ed. by Xilinx. 2019.1. June 21, 2019.

[9] M. Y. Hsiao. "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes". In: *IBM Journal of Research and Development* 14.4 (July 1970).

2. deterministic behavior unknown and unpredictable during production, i.e., PUFs[10] or to

3. hide sensitive information from side channel attacks by producing additional electronic and magnetic fields.

Intellectual property cores (IP cores) that are designed to provide such functionality are often considered *hardware security primitives*.[11] In this work, we elaborate on Security primitive 1, RNGs, and Security primitive 2, PUFs.

RNGs and PUFs will be discussed in Section 2.1.1 and Section 2.1.2 respectively with a focus on RNGs since we build an RNG as our use-case example. Concrete design examples will be given for *field-programmable gate arrays (FPGAs)* for the same reason.

2.1.1 Random number generators

Random numbers are needed in a variety of applications. In some domains, such as video gaming, there is no strict requirement on the quality of the random numbers. In other domains, such as gambling and cryptography, predictable random numbers can cause loss of capital[12] or leakage of restricted information.[13] Due to the deterministic property of processes, unpredictable random numbers can be hard to obtain. RNGs need a source of entropy. Example sources of entropy include:

1. Network traffic,

2. Timing and values of *input/output (I/O)* devices,

3. Radioactive decay,

4. Sensor noise,

5. Quantum noise,

6. Deterministic procedures,

7. Metastability of logic blocks,

8. Clock jitter, and

9. Combinations of the above.

If users are in full control over a system, they can freely choose an adequate source of entropy from the list above. However, if they are not in control of the physical hardware (e.g., in cloud environments), most of the options listed above are not available. Entropy source 1 and Entropy source 2 can be controlled by the *cloud service provider (CSP)* and are therefore not trustworthy enough as sources of entropy for cryptographic algorithms. Sensors are needed for Entropy source 3 to Entropy source 5. These sensors are usually not available in a cloud setting and if they would be, they would be in control of the CSP and therefore also unsuitable for security applications. Random numbers obtained from deterministic procedures (Entropy source 6) are not truly random. As the word suggests, they are calculated

[10] T. Rahman. "Hardware-based security primitives and their applications to supply chain integrity". PhD Dissertation. University of Florida, 2017.
[11] Y. Bi et al. "Emerging Technology-Based Design of Primitives for Hardware Security". In: *J. Emerg. Technol. Comput. Syst.* 13.1 (Apr. 2016).
[12] R. Forgrave. *The Man Who Cracked the Lottery*. en. https://www.nytimes.com/interactive/2018/05/03/magazine/money-issue-iowa-lottery-fraud-mystery.html. May 2018. (Visited on 06/24/2021).
[13] M. Green. *The many flaws of Dual_EC_DRBG*. https://blog.cryptographyengineering.com/2013/09/18/the-many-flaws-of-dualecdrbg/. 2013. (Visited on 02/19/2021).

deterministically. Anybody with knowledge of the used procedure can potentially predict the generated numbers. They fall into the class of *Pseudo random number generators (PRNGs)*. State-of-the-art approaches[14] include:

- The Mersenne Twister,
- The *Well Equidistributed Long-period Linear (WELL)* family,
- The XORShift, and
- The Tausworthe random number generator.

Hardware implementations for reconfigurable hardware exist as well.[15]

Some CSPs offer FPGAs as a shared resource, allowing users to implement custom logic connected to their cloud instances. In that case, Entropy source 7 and Entropy source 8 can be used as entropy sources in such cloud environments.

Petura et al. present a variety of commonly used RNG implementations[16] for FPGAs. They select six designs based on the following criteria:

1. Digital-only design
2. AIS-20/31 compliance
   (a) Simple and comprehensible design
   (b) Clearly defined source of entropy
   (c) Stationary random process with feasible stochastic model
   (d) Accessible raw binary signal for testing
3. FPGA vendor- and family agnostic

AIS-20/31 compliance allows the generated random numbers to be used in cryptographic applications. Criterium 1 and Criterium 3 make them easily portable to many hardware targets.

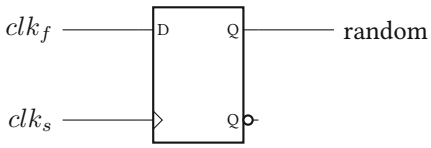
All designs in Petura et al.'s work share the basic working principle: a slow, jittery clock clk_s is used to sample a much faster clock clk_f in order to extract entropy from the jitter of clk_s. An example circuit demonstrating the sampling is shown in Figure 2.2a.

The timing diagram of the sampling process can be seen in Figure 2.2b. The time-function of the sampling clock, clk_s, has windows of uncertainty around its edges. These windows stem from the clock's imprecise timing and are called *jitter windows*. *Jitter* is an inherent property of all clock generators. It is defined as the deviation from true periodicity and measured either as an absolute value in seconds or relative to the clock period in percent.

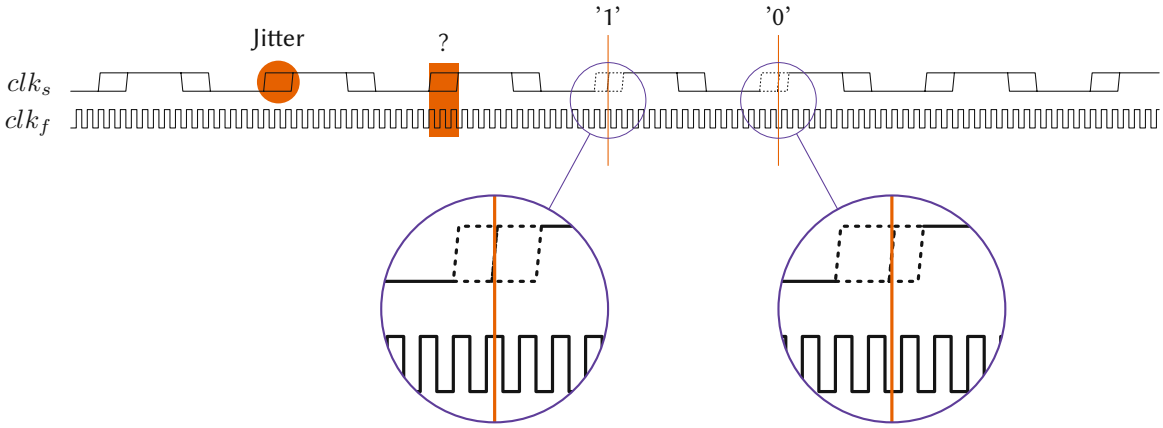
[14] K. Bhattacharjee, K. Maity, and S. Das. "A Search for Good Pseudo-random Number Generators : Survey and Empirical Studies". In: *arXiv:1811.04035 [cs]* (Nov. 2018). arXiv: 1811.04035.

[15] M. Bakiri et al. "Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses". en. In: *Computer Science Review* 27 (Feb. 2018).

[16] O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016.



(a) Clock-sampling random number generator circuit. The fast, precise clock clk_f is sampled by the slow, imprecise clock clk_s.



(b) Entropy extraction from clock-jitter. The jitter in the sampling clock makes the sampling result in a random bit.

Figure 2.2: Schematic (Subfigure a) and timing diagram (Subfigure b) of the clock-sampling random number generator.

In Figure 2.2b, jitter windows are illustrated as overlapping double-edges (one jitter window is highlighted by an orange circle). The actual edge of the signal may occur at any time during such a window. The exact moment is determined at random. This makes it a derogatory property of clocks, but opens the possibility to exploit that uncertainty to extract randomness.

If such a jittery clock is used to sample a second clock, it is indeterminable at what exact time the sampling takes place. If that second clock is faster than the jittery clock, multiple clock-periods of the fast clock will fit into the windows of uncertainty of the jittery clock. This is shown in the orange rectangle in Figure 2.2b. It implies that the sampling may result in a logic 1 or 0, and the outcome is dependent on the exact timing of the sampling edge. This is illustrated by the magnified regions in the purple circles, with the orange line showing the exact occurrence of the edge. In the left circle, the rising edge of clk_s occurs while clk_f is high, so the sampling results in a 1. In the right circle, the jitter causes the rising edge of clk_s to occur at a slightly different point during the period. In this case, the sampling results in a 0 because clk_f is low when the rising edge of clk_s occurs.

Petura et al. implemented and tested six such designs on three FPGA families from different vendors. The evaluated designs are:

1. The *ring-oscillator-based elementary TRNG (ERO-TRNG)*,
2. The *ring-oscillator-coherent sampling based TRNG (COSO-TRNG)*,
3. The *multi-ring-oscillator-based TRNG (MURO-TRNG)*,
4. The *coherent-sampling-based TRNG using* Phase-locked loops (PLLs) *(PLL-TRNG)*,
5. The *transition-effect-ring-oscillator-based TRNG (TERO-TRNG)* and

TRNG type	FPGA device	Area (LUT/Reg) Metric 1	Power $[mW]$ Metric 2	Bit rate $[Mbits/s]$ Metric 3	Efficiency $[bits/\mu Ws]$ Metric 4	Entropy $1/bit$ Metric 5	Entropy * Bit rate Metric 6	Feasib. & Repeat. Metric 7
ERO-TRNG	Spartan 6	46/19	2.16	0.0042	1.94	0.999	0.004	5
	Cyclone V	34/20	3.24	0.0027	0.83	0.990	0.003	
	SmartFusion 2	45/19	4	0.014	3.5	0.980	0.013	
COSO-TRNG	Spartan 6	**18/3**	**1.22**	0.54	442.6	0.999	0.539	1
	Cyclone V	**13/3**	**0.9**	1.44	**1600**	0.999	1.438	
	SmartFusion 2	**23/3**	**1.94**	0.328	169	0.999	0.327	
MURO-TRNG	Spartan 6	521/131	54.72	2.57	46.9	0.999	2.567	4
	Cyclone V	525/130	34.93	2.2	62.9	0.999	2.197	
	SmartFusion 2	545/130	66.41	3.62	54.5	0.999	3.616	
PLL-TRNG	Spartan 6	34/14	10.6	0.44	41.5	0.981	0.431	3
	Cyclone V	24/14	23	0.6	43.4	0.986	0.592	
	SmartFusion 2	30/15	19.7	0.37	18.7	0.921	0.340	
TERO-TRNG	Spartan 6	39/12	3.312	0.625	188.7	0.999	0.624	1
	Cyclone V	46/12	9.36	1	106.8	0.987	0.985	
	SmartFusion 2	46/12	1.23	1	813	0.999	0.999	
STR-TRNG	Spartan 6	346/256	65.9	**154**	**2343.2**	0.998	**154.121**	2
	Cyclone V	352/256	49.4	**245**	**4959.1**	0.999	**244.755**	
	SmartFusion 2	350/256	82.52	**188**	**2286.7**	0.999	**188.522**	

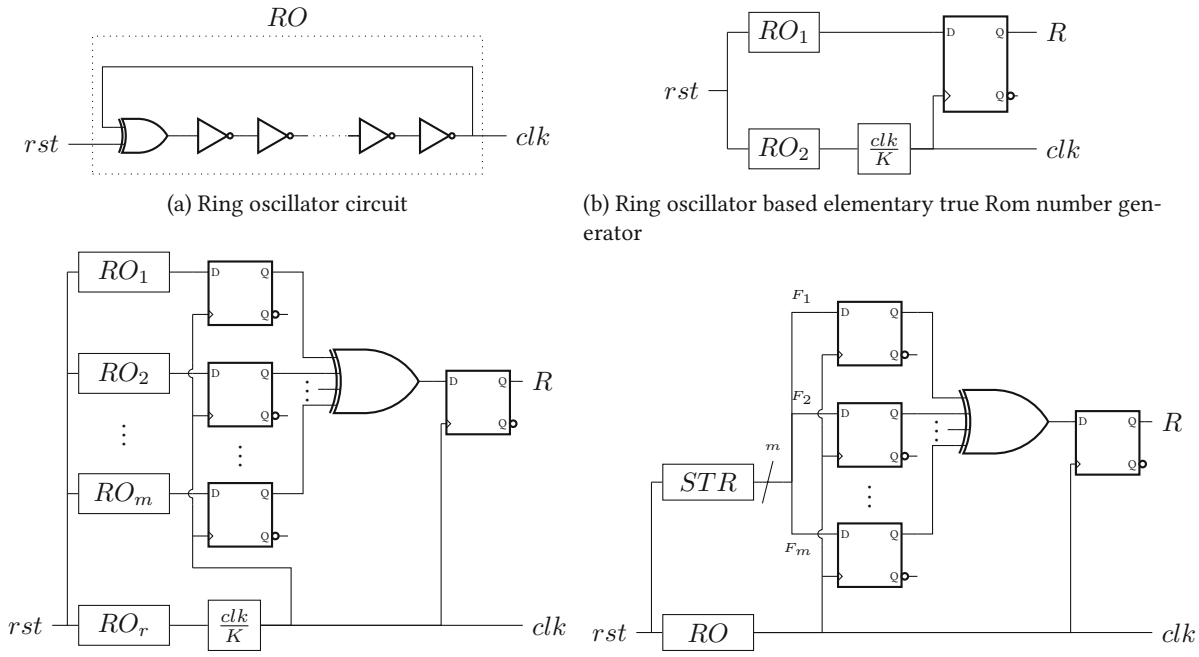
Table 2.1: Evaluation of random number generators on FPGAs by O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016

6. The STR-*based TRNG (STR-TRNG)*

They compared them on the basis of seven key parameters (Metrics):

1. *Area* in *lookup table (LUT)* and register counts,
2. *Power consumption* in mW,
3. *Bit rate* in $Mbits/s$,
4. *Efficiency* in $\frac{bits}{\mu Ws}$,
5. *Entropy* per bit in $\frac{1}{bit}$,
6. *Entropy* $\times$ *Bit rate* in $\frac{1}{s}$, and
7. *Feasibility and repeatability* on a scale from 1 to 5. Easily portable designs that produce good results across all tested devices score highest (up to 5). Designs that need manual intervention for each device score lowest (down to 1).

In the following we discuss their findings, which are summarized in Table 2.1: All designs show an entropy per bit of 0.92 or more making them suitable for cryptographic use. Three of the designs stand out in particular: The ERO-TRNG, the MURO-TRNG and the STR-TRNG are all based on the entropy extraction from jittery clocks discussed in Section 2.1.1



(a) Ring oscillator circuit

(b) Ring oscillator based elementary true Rom number generator

(c) Multi-ring oscillator based true random number generator

(d) Self-timed-ring based true Rom number generators

Figure 2.3: Schematic of a ring oscillator (Subfigure a) and three true random number generator circuits (Subfigures b to d) as evaluated in O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016.

and illustrated in Figure 2.2b. They all use a *ring oscillator (RO)* for jitter extraction (schematic in Figure 2.3a). They have a comparable entropy per bit of at least 0.98, but they differ strongly in Metric 1 to Metric 4 and Metric 7.

We discuss the results for these three designs in more detail (performance numbers for comparison are given for the *Spartan 6* implementation. Performance numbers for the *Cyclone V* and *SmartFusion 2* implementations can be found in Table 2.1):

1. The ***ring-oscillator-based elementary TRNG (ERO-TRNG)***[17] is based on two ROs of the same size. One of the ROs feeds a clock divider which in turn samples the second RO (Figure 2.3b). It can be implemented on any FPGA without manual intervention. It produces good results while being relatively small (46 LUTs, 19 registers), but it is very slow (Output bit rate: $0.0042 Mbits/s$).

2. The ***multi-ring-oscillator-based TRNG (MURO-TRNG)***[18] aims at mitigating the shortcomings of the ERO-TRNG by adding more ROs. m ROs are implemented in parallel and fed through an *exclusive-or (XOR)* gate (Figure 2.3c). The result is sampled by one more RO that again has its output frequency divided. The resulting bit rate is much higher ($2.57 Mbits/s$), but comes at the cost of size (521 LUTs, 131 registers) and power ($54.72 mW$). Due to the possibility of some of the many ROs locking onto each other, it scores lower on the feasibility and repeatability scale compared to the ERO-TRNG.

[17] M. Baudet et al. "On the Security of Oscillator-Based Random Number Generators". In: *Journal of Cryptology* 24.2 (Apr. 1, 2011).

[18] B. Sunar, W. J. Martin, and D. R. Stinson. "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks". In: *IEEE Transactions on Computers* 56.1 (Jan. 2007).

3. The ***self timed ring (STR)-based TRNG (STR-TRNG)***[19] is the best performing RNG in Petura et al.'s comparison (Output bit rate: $154Mbits/s$, Efficiency: $2343.2bits/\mu Ws$). It consists of an RO sampling an STR (thoroughly discussed in Section 2.3). The STR-TRNG is constructed similarly to the MURO-TRNG, but uses clocks produced by individual stages of an m-stage STR instead of the m parallel ROs (Figure 2.3d, details on the STR in Section 2.3). It spots the highest bit rate and efficiency at the cost of high area usage and power consumption. The STR needs to be manually placed and routed, resulting in a low score on the feasibility and repeatability scale (2 out of 5).

The numbers in Table 2.1 give the STR-TRNG the edge over other designs in terms of performance. The bad feasibility and repeatability score make the implementation time-consuming and inaccessible to users with no experience in manual *placement and routing (PnR)*. We aim to mitigate this shortcoming with our proposed PnR algorithm (Chapter 3) designed specifically for hardware security primitives. It makes the implementation of a STR-TRNG as easy as the implementation of any other IP-core.

2.1.2 Physical unclonable functions

A second class of non-deterministic circuits are PUFs. PUFs are functions that provide a digital *fingerprint* as a response to a *challenge* input. Hardware implementations capitalize on the presence of process variation to determine a unique fingerprint on each device. This implies that the behavior changes from device to device but is repeatable. The PUF will yield the same results on the same device on multiple runs, but different results on different devices. Typically, PUFs are defined as

$$Y = P(C). \tag{2.1}$$

Users can provide a PUF a challenge C, getting back a response Y. Due to the PUF's properties, response Y depends on a chip's physical properties and as the physical properties impose unique variations on each chip, PUFs can therefore be used to identify this chip. PUFs can be used in security applications to authenticate chips. They are therefore often considered a chip's fingerprint.

Popular PUF implementations for FPGAs include the *ring oscillator based physical unclonable function (RO-PUF)* and the arbiter PUF, summarized and evaluated by Suh and Devadas,[20] both discussed below. To use PUFs for authentication they must always produce the same result on the same chip but different results on different chips. The result of the PUF can then be used as an identifier which is unique for a device. This observation leads to the quality parameters on PUFs defined by Suh and Devadas:[21]

1. *Inter-chip variation* v_{inter}: The probability of the result being different on different chips (the higher the better) and

2. *Intra-chip variation* v_{intra}: The probability of the result being different for subsequent experiments on the same chip (the lower the better)

[19] A. Cherkaoui et al. "Comparison of Self-Timed Ring and Inverter Ring Oscillators as entropy sources in FPGAs". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2012; A. Cherkaoui et al. "A Self-Timed Ring Based True Random Number Generator". In: *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*. 2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems. May 2013.

[20] G. E. Suh and S. Devadas. "Physical Unclonable Functions for Device Authentication and Secret Key Generation". In: *2007 44th ACM/IEEE Design Automation Conference*. ISSN: 0738-100X. June 2007.

[21] Ibid.

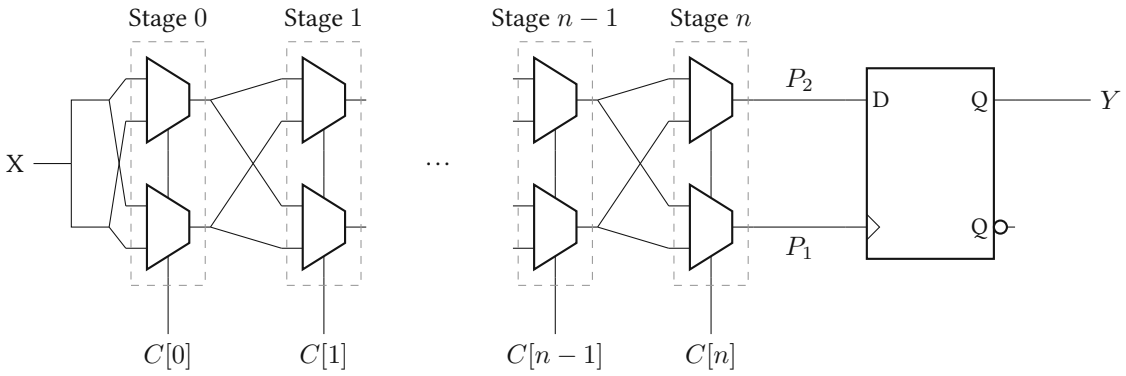


Figure 2.4: Arbiter PUF circuit as presented in G. E. Suh and S. Devadas. "Physical Unclonable Functions for Device Authentication and Secret Key Generation". In: *2007 44th ACM/IEEE Design Automation Conference*. ISSN: 0738-100X. June 2007. All paths from input X to P_1 and P_2 are of equal length. Ideally, a rising edge on input X would therefore reach P_1 and P_2 simultaneously. Due to process variation, this is not the case. There is a race condition from the input signal X to the two inputs P_1 and P_2 of the D-flip-flop near the end. The multiplexers steer the input signal X along different paths according to their select signals. All those paths should have the same delay by design, but this is skewed by process variation. Therefore, the selection of the paths influences the outcome of the race condition. The output is then dependent on whether X reaches P_1 or P_2 first which is dictated by the process variation.

To identify a chip using a PUF, its inter-chip variation must be much higher than the intra-chip variation.[22]

$$v_{inter} \gg v_{intra} \tag{2.2}$$

Only then, the PUF gives consistent responses on the same chip while being different on individual chips, making it suitable as an identifier.

Arbiter PUF

The arbiter PUF, originally introduced by Lee et al.[23] and shown in Figure 2.4, consists of n pairs of multiplexers with two inputs and a single-bit control signal. Each pair of multiplexers is considered a stage of the arbiter PUF. Stages are marked with dashed lines in Figure 2.4. The inputs for the first stage (stage $C[0]$), are all connected to the same signal X. The two multiplexers in each subsequent stage are connected to the same inputs and share the same control signal. The output pair of one stage connects to the input pairs of the next stage. The outputs of the last stage, stage n, are connected to the inputs of a D-flipflop, with one of the outputs (P_1) sampling the other (P_2). The sampling results in a race condition between the clock input signal and the data input signal of the D-flipflop.

The control signals $C[i]$ of the n stages form a challenge vector C and determine two paths from input X to the two inputs of the D-flipflop.

A critical requirement for the arbiter PUF is that all wires connecting the multiplexers and the multiplexers themselves are designed to have equal delay. This makes the delay between an edge occurring at the input X and its arrival at the inputs of the D-flipflop independent from the paths selected by the challenge vector C. Due to process variations this assumption is very unlikely to hold. Some paths will be faster than others and most importantly: the same path will have different delays on different devices.

[22] Ibid.

[23] J. W. Lee et al. "A technique to build a secret key in integrated circuits for identification and authentication applications". In: *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*. June 2004.

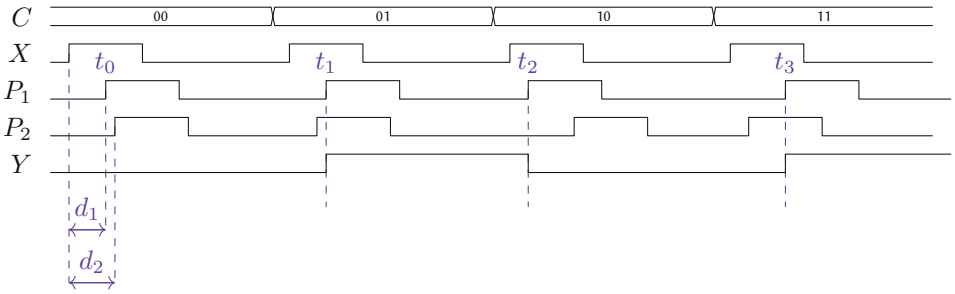


Figure 2.5: Arbiter physical unclonable function timing example. The difference in delays d_1 and d_2 alone dictates the result of the function. Is C chosen so that X reaches P_1 before P_2 ($d_1 < d_2$), the result is 0, otherwise it is 1.

The D-flipflop near the output is used to determine which of the two paths (P_1 or P_2) selected by C has the longer delay. If the path to the clk input (P_1) is faster than the path to the D input (P_2), the sampling will result in a logic 0, as D is still 0 at the time the rising edge arrives at clk. If the path to the clk input is slower than the path to the D input, the sampling will result in a logic 1, as D is already 1 at the time the rising edge arrives at clk.

Figure 2.5 shows an example behavior of an arbiter PUF with two stages (the challenge signal C is therefore a two-bit signal). In this example, the user iterates over all possible challenges, allowing time in between for the arbiter PUF to respond. Shortly after the challenge signal is set, X is set to high for a short period of time. Due to the delay within the arbiter PUF, the rising edge of the X signal is seen at the inputs P_1 and P_2 after a delay (d_1 and d_2 respectively. As discussed before, these delays depend on the device on which the PUF is implemented). Because the paths from X to P_1 and P_2 are different, the delay is different as well (d_1 and d_2 respectively). For the challenge $C = 00$, the first edge is seen at P_1 after t_0; therefore, the D-flipflop samples P_2. This sampling results in a 0 visible at Y because P_2 is still low at t_0 (the edge reaches P_1 earlier than it reaches P_2). The second time the edge reaches P_1 is at t_1 for challenge $C = 01$. The delay to P_2 was smaller; therefore P_2 is already high when the sampling happens and a 1 is seen at the output. The rest of the challenges compute similarly, resulting in the complete correspondence between challenge and response as seen in Table 2.2.

Challenge	Response
00	0
01	1
10	0
11	1

Table 2.2: Example arbiter physical unclonable function

The output Y for all possible challenge vectors $C \in \mathbb{Z} < 2^n$ is therefore depending on the device. Suh and Devadas determine the inter-chip variation of the arbiter PUF to 23% and the intra-chip variation to 0.7%. The property in Equation (2.2) is satisfied, making the PUF usable as a chip identifier.

Ring oscillator PUF

The RO-PUF, introduced by Suh and Devadas and depicted in Figure 2.6, also utilizes the delay-variance between devices to create a PUF. n identical ring oscillators are connected to two $n \times 1$ multiplexers with multi-bit control signals C_A and C_B. The outputs of both multiplexers are connected to one counter per multiplexer. At the end, the outputs of the two counters are compared.

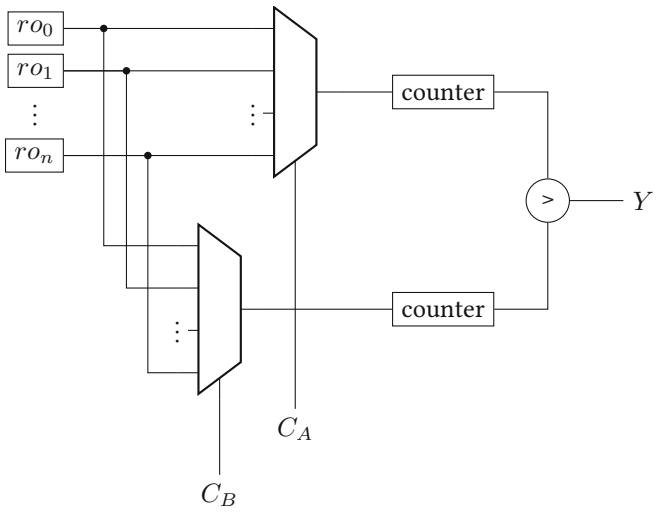


Figure 2.6: RO PUF circuit as introduced in G. E. Suh and S. Devadas. "Physical Unclonable Functions for Device Authentication and Secret Key Generation". In: *2007 44th ACM/IEEE Design Automation Conference*. ISSN: 0738-100X. June 2007. Multiple identical ring oscillators are implemented in parallel. Their frequency will differ slightly due to process variations. Challenge signals C_A and C_B select two of them. The two selected clocks drive a counter each. The comparison near the output indicates which clock is faster and is, therefore, dependent on the process variations.

C_A and C_B are used to select two of the ROs. Due to manufacturing imperfections, the ROs will oscillate with slightly different frequencies. The difference in frequencies results in one of the counters counting faster. The comparator towards the output then decides which RO has the higher frequency.

The multi-bit multiplexer control signals C_A and C_B can again be considered challenges with Y as the challenge response.

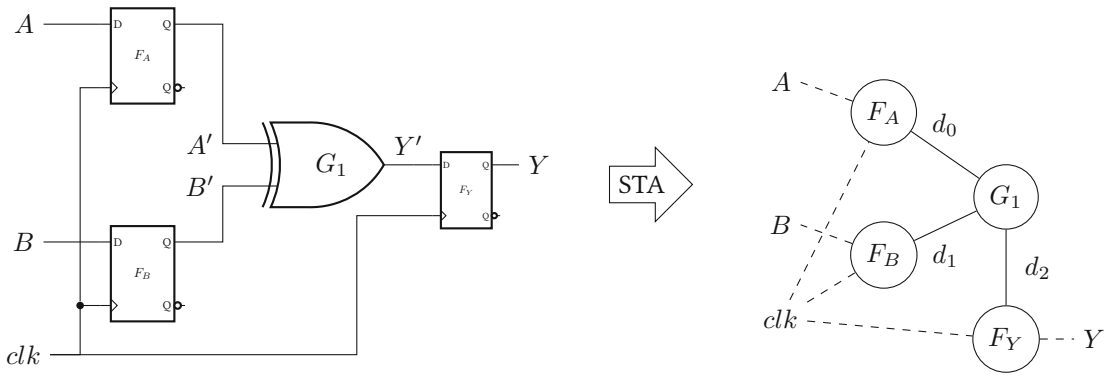
In the experiments conducted by Suh and Devadas, the inter-chip variation per bit is 46.14%, very close to the ideal of 50%. The intra-chip variation is 0.48%.[24] Since the inter-chip variation is much larger, Equation (2.2) is satisfied, the RO PUF identifies a chip and can therefore be used for authentication.

2.2 Timing requirements

The TRNG and PUF designs presented in Section 2.1 have one particular property in common: the performance of the designs depends on their routing. In particular, routes must satisfy delay requirements. Satisfying timing constraints is a common problem in digital IC development. FPGA vendors therefore include the option to constrain paths in their synthesis and PnR tools.[25]

Before discussing timing constraints, we investigate how timing is assessed for FPGAs in Section 2.2.1. Second, we show typical constraints for FPGA designs in Section 2.2.2. These are the constraints that most FPGA vendors implement. Section 2.2.3 elaborates what additional requirements are necessary for security primitives and justifies why those are not typically implemented in *electronic design automation (EDA)* tools.

[24] Suh and Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation".
[25] Xilinx. *UG945 Vivado Design Suite Tutorial: Using Constraints*. Ed. by Xilinx. 2019.1. June 24, 2019; Xilinx, *UG903 Vivado Design Suite User Guide: Using Constraints*.



(a) Example circuit

(b) Corresponding timing graph

Figure 2.7: Example synchronous circuit (Subfigure a) and the result of a static timing analysis on said circuit. (Subfigure b)

2.2.1 Static timing analysis

There are multiple ways to determine the timing of a digital circuit. Transistor-level simulation can yield highly accurate results, but is unfeasible for large designs due to design complexity. When designing for FPGAs one does not usually have knowledge of the exact gate-level design of the FPGA, completely disqualifying gate-level simulation for this use-case. The second option is in-circuit testing, which is also very precise. In-circuit testing is also not an option on FPGAs since designers do not have timing-accurate access to internal signals. The last option is *static timing analysis (STA)*. It uses an abstract model of the hardware in various environmental conditions to give delay boundaries for the design. Depending on the model it can be very accurate while scaling well for large designs.

STA checks every path in a design against the timing model of the device. This process yields the delay a signal traversing that path will experience. That delay may originate from the length of a wire or from the switching speed of a logical element. The result of the STA is a timing graph. Nodes in the original design are nodes of the timing graph, including their switching delay. Interconnecting wires are the graph's edges. The delay of a wire is the weight of the edge. This can be seen on the example of Figure 2.7. The simple circuit in Figure 2.7a shows a synchronous XOR operation on two signals A and B. The corresponding timing graph can be seen in Figure 2.7b. The weight of the edges between the nodes is the delay between the corresponding logic gates d_i.

2.2.2 Typical timing requirements in electronic design automation

Timing requirements are usually defined for synchronous circuits and not for asynchronous circuits. In synchronous circuits, a clock signal dictates the pace of operation by synchronizing the data transfer in D-flipflops. Figure 2.7 shows an example implementation of a synchronous two-input XOR operation with all D-flipflops driven by the clk clock signal. Figure 2.8 shows the timing diagram for an arbitrary sequence of inputs, with A', B' and Y' being the two input signals after synchronization and the output signal before synchronization respectively.

A circuit must meet the following requirements to perform as expected:

1. Results must be steady during a window of time around the sampling positive (or negative) edges of the clock signal. All input signals must be stable t_S seconds before the sampling edge (*setup time*) and all output signals must be held stable for at least t_H seconds after the sampling edge (*hold time*). This is due to metastability which develops in synchronization elements when the input changes very close to sampling edges.

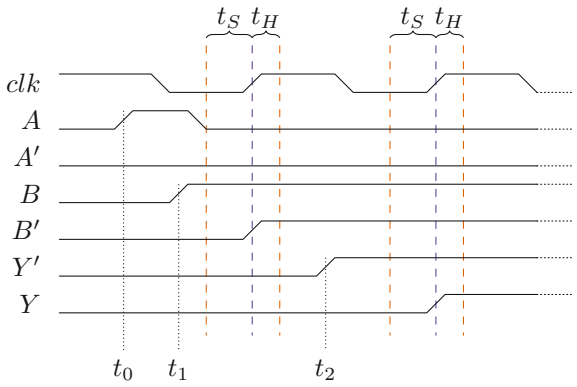


Figure 2.8: Possible timing diagram of the example synchronous circuit in Figure 2.7, showing hold-time t_H and setup time t_S.

This requirement is met for the circuit given in Figure 2.7 with the timing in Figure 2.8. The times when sampling edges occur are marked with a dashed purple line and the window in which the signals are not allowed to change is bounded by two dashed orange lines.

2. All operations between two synchronization elements must complete within one clock cycle.

   This requirement is also met in Figure 2.8. The result Y' of the XOR gate G_1 is computed at t_2, well before the next sampling clock edge.

3. Input changes are only registered if they are visible for at least one sampling clock edge.

   This requirement is not met in Figure 2.8. The time the input A is high (starting at t_0) is not long enough to be sampled by F_A. The change is therefore never visible at A' and has no impact on the result. Input B on the other hand stays high after t_1 and is therefore sampled properly by F_B.

If one of these requirements is not met, placement and routing of the circuit must be improved or the clock frequency has to be changed.

STA gives designers insights about the timing of their implemented circuits and allows them to verify whether their expectations are met. EDA tools use STA to check the timing during synthesis to optimize PnR according to the requirements listed above. To make that possible, designers have to declare their needs to the EDA tool via *timing constraints*. Such timing constraints typically include:[26]

1. **Clock constraints:** If all properties of a clock signal are known, the STA model is sufficient to assess the timing in terms of setup and hold times for all synchronized signals. The designer therefore has to provide the following properties of all clocks in a design:

   (a) Period

   (b) Duty-cycle

   (c) Latency

[26] Xilinx, *UG945 Vivado Design Suite Tutorial: Using Constraints*; Xilinx, *UG903 Vivado Design Suite User Guide: Using Constraints*; R. Cofer and B. F. Harding. *Chapter 9 - Design Constraints and Optimization.* Ed. by R. Cofer and B. F. Harding. Embedded Technology. Burlington: Newnes, 2006. Chap. 9.

(d) Jitter

2. **Input constraints**: STA can only determine the timing within the boundaries of a device. To check the design against setup and hold time violations, EDA tools need to be made aware of delays existing outside the device. This is done by specifying minimum and maximum delays from the originator (e.g., another chip) to the input pin on the device relative to the clock signal(s) that synchronizes the signal. They indirectly specify during which timeframes (relative to the clock) the device can expect the input to be valid. The PnR algorithm has to make sure the device samples the input only during those timeframes.

3. **Output constraints**: The routes to an output interface must be checked as well, to make sure the output produced by the device is within the specs of the device receiving it. This is also done by specifying minimum and maximum delays from the output pin of the FPGA to the input interface it feeds relative to the synchronizing clock. The minimum and maximum delays specify the timeframes (relative to the clock) the output signal needs to be valid for the receiving chip to safely register it. The PnR algorithm must make sure the device only produces outputs that match this specification.

2.2.3 Timing requirements for security primitives

The constraints presented in Section 2.2.2 all follow the same goal: To produce a design that behaves the same on all devices all the time in a wide range of environments. This is not desired when designing any of the security primitives presented in Section 2.1: RNGs use environmental noise to extract entropy while PUFs use device characteristics to compute a device-specific fingerprint. Nevertheless, five out of six designs presented by Petura et al.[27] and both designs presented by Suh and Devadas[28] need careful placement and routing to produce satisfactory results. They follow the exact opposite goal, to make designs highly conditioned by environmental noise (RNG) or by device characteristics (PUF).

A concept often used both in RNGs and PUFs are *race conditions*. These occur when two or more signals are precisely timed to change at the same time. Depending on which of the input signals arrives first, the output signal changes. PUFs use race conditions to measure the differences in delay on different devices due to manufacturing imperfections while RNGs use them to purposefully provoke metastability.

The most common timing constraint for race conditions is *equal delay along multiple routes*. A short check against the constraints presented in Section 2.2.2 reveals that in state-of-the-art EDA tools it is not possible to specify such a timing constraint. Designers have to make sure on their own that the design behaves appropriately by iterative manual placement and routing with continuously checking its timing properties.

2.3 Use-case example

In this work, we introduce timing constraints for security primitives. We test them on a sample design, an STR-TRNG. The core itself is presented in Section 2.3.1. The timing requirements on the STR-TRNG that cannot be specified using traditional timing constraints are discussed in Section 2.3.2. The main quality parameter we maximize is the entropy

[27] Petura et al., "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices".
[28] Suh and Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation".

achieved by the RNG. In Section 2.3.3 we showcase methods that can be used to assess the performance of RNGs purely by the random numbers it produces. State-of-the-art certification entities are listed in Section 2.3.4.

2.3.1 The self-timed-ring-based true random number generator

The STR-TRNG, first introduced by Cherkaoui et al.,[29] consists of two oscillating circuits. It is based on the concept of jitter extraction by clock sampling as discussed in Section 2.1.1 and as shown in Figure 2.2a. The sampling clock is derived from a common RO with n delay elements. The sampled clock is an STR. STRs, first introduced by Elissati et al.,[30] are oscillation circuits that can produce very high-speed, high-resolution, low phase-noise clocks. Cherkaoui et al. constructed a stochastic model to calculate a lower bound for the entropy from frequency and jitter measurements.[31] They prove that with adequate routing the STR-TRNG produces enough entropy to be used in cryptographic applications.

The schematic of an m-stage STR is given in Figure 2.10. The STR has two inputs, an active-high *reset* signal and an *initialization vector* (omitted in Figure 2.10 for better readability). At the outputs F_i and R_i it produces an oscillation.

An m-stage STR consists of m *Mueller C-gates* C_i. We showcase one of the Mueller C-gates magnified in Figure 2.10. A C-element hold its value y at the output if both of its two inputs X_1, X_2 are different from each other, and adopt the input value if they are equal (Table 2.3). They are arranged in a ring. One input of a C-element is the output of the

$X_{1,n}$	$X_{2,n}$	Y_n
0	0	0
0	1	y_{n-1}
1	0	y_{n-1}
1	1	1

Table 2.3: Truth table of the Mueller C-element

previous C-element, the other input is the output of the next element. The connection from the output of stage C_i to the next stage C_{i+1} is called *forward connection* F_i. The connection from the output of stage C_i to the previous stage C_{i-1} is called *reverse connection* R_{i-1}. The ring can be reset asynchronously by raising the $Reset$ signal. All stages output their *initialization value* $Init_i$ while the $Reset$ signal is high.

The initialization vector $Init$ is made up of N_T evenly distributed logic ones. The ones are called *tokens* while the zeroes are called *Holes* of the STR. If the number of tokens N_T is carefully chosen to be coprime to the size m of the STR, each stage will produce an output clock F_i with constant mean phase difference. The higher the number of *tokens*, the higher the output frequency.

All outputs F_i are then connected to an XOR gate producing the final clock output (in Figure 2.3d). The XOR gate calculates the parity of its inputs. That means that the output toggles each time a single input signal toggles. Since the

[29] Cherkaoui et al., "Comparison of Self-Timed Ring and Inverter Ring Oscillators as entropy sources in FPGAs"; Cherkaoui et al., "A Self-Timed Ring Based True Random Number Generator".

[30] O. Elissati et al. "Self-Timed Rings: A Promising Solution for Generating High-Speed High-Resolution Low-Phase Noise Clocks". In: *VLSI-SoC: Forward-Looking Trends in IC and Systems Design*. Ed. by J. L. Ayala, D. Atienza Alonso, and R. Reis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[31] A. Cherkaoui et al. "A Very High Speed True Random Number Generator with Entropy Assessment". In: *Cryptographic Hardware and Embedded Systems - CHES 2013*. Ed. by G. Bertoni and J.-S. Coron. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

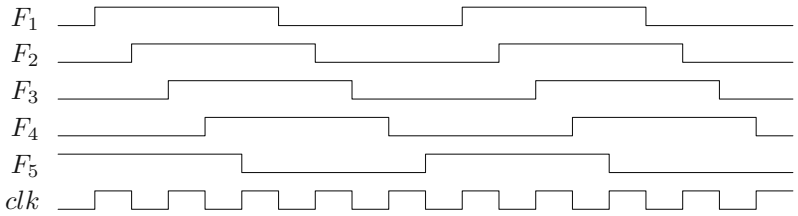


Figure 2.9: Self timed ring timing. Each stage outputs a clock signal. The output of the individual stages is shifted by T/STR_SIZE. The final clock is obtained by an or-operation, resulting in a frequency that is *STR_SIZE* times faster than the clocks of the individual stages.

internal clocks F_i toggle one after the other, the XOR parity computation results in a clock signal as well. The final clock clk is m times faster compared to the clocks produced by the individual stages F_i since it toggles for every F_i. This can be seen in Figure 2.9 on the example of an STR with $m = 5$ stages.

2.3.2 Timing requirements of the self timed ring

Petura et al.[32] give the STR-TRNG a 2 out of 5 (with 5 being very feasible and easily repeatable) on their feasibility and repeatability scale. This is due to the fact that the STR oscillator needs careful placement and routing in order to obtain stable and well-timed oscillation.[33] When implementing the STR-TRNG, designers need to make sure that two sets of signals have similar delays. The two sets of signals are:

1. Signals internal to the Mueller C-gates ($I_{i,1-6}$, purple in Figure 2.10) and

2. Signals interconnecting the Mueller C-gates (F_i and R_i, orange in Figure 2.10).

The smaller the difference in delay of all signals in those two sets, the more precise and fast the oscillation of the STR.

This is due to the *Charlie effect* and the *Drafting effect* in the Mueller C-gates.[34] The requirement in Equation (2.3) ensures the oscillation to be constant at 50% duty cycle.

$$\frac{N_T}{N_B} \approx \frac{D_f}{D_r} \tag{2.3}$$

Unfortunately, it is not possible to specify such a constraint in state-of-the-art EDA tools.[35] Constraints can only be defined relative to a clock signal. There is no way of constraining a group of signals to have similar delays.

[32] Petura et al., "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices".
[33] Cherkaoui et al., "A Very High Speed True Random Number Generator with Entropy Assessment".
[34] J. Hamon et al. "High-Level Time-Accurate Model for the Design of Self-Timed Ring Oscillators". In: *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*. ISSN: 1522-8681. Apr. 2008.
[35] Altera. *Quartus II TimeQuest Timing Analyzer, Quartus II 9.0 Handbook, Volume 3*. Ed. by Altera. Mar. 2019; Lattice. *Timing Closure*. Ed. by Lattice. Oct. 2013; Xilinx, *UG945 Vivado Design Suite Tutorial: Using Constraints*.

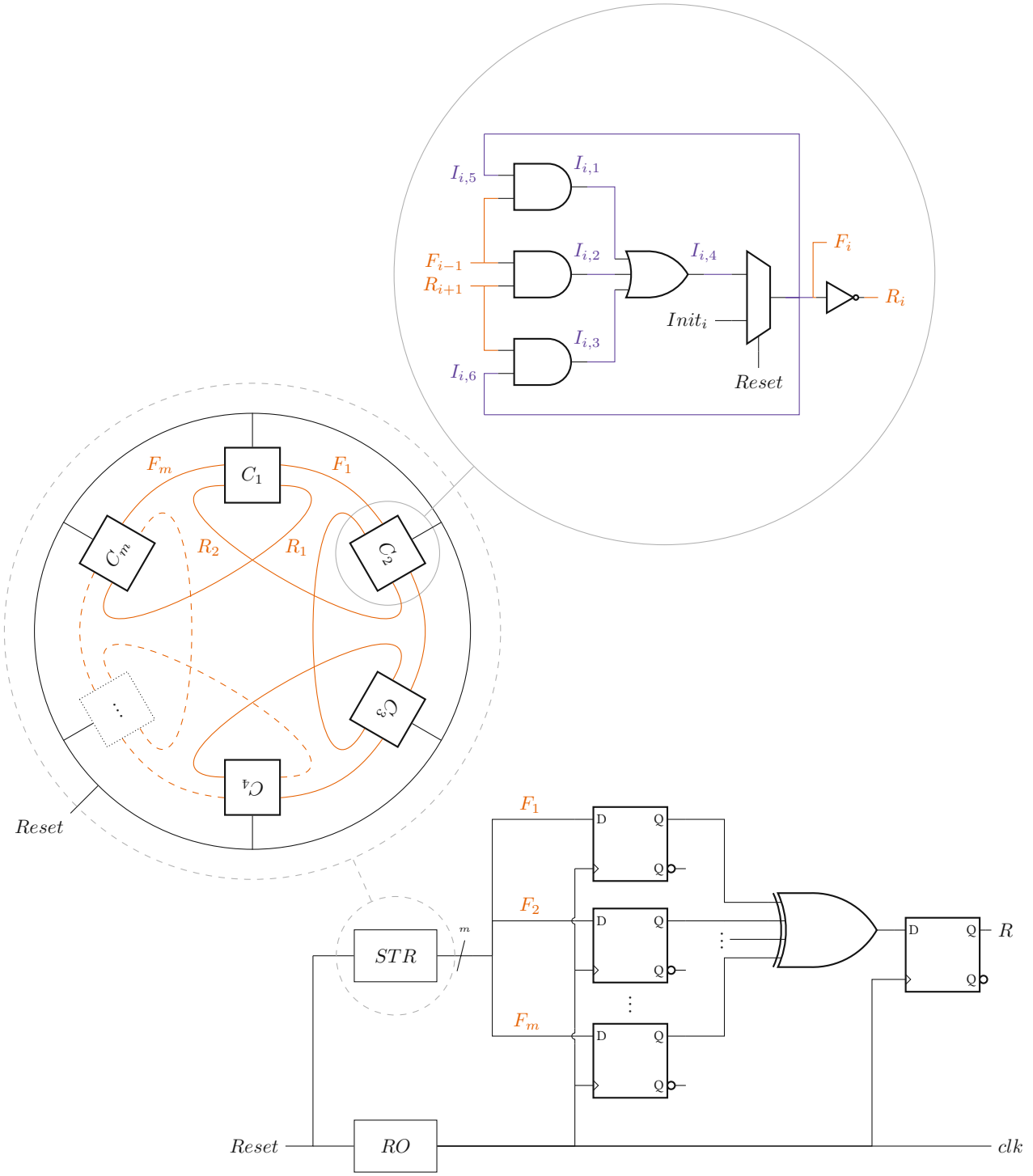


Figure 2.10: Self timed ring circuit (adapted from O. Elissati et al. "Self-Timed Rings: A Promising Solution for Generating High-Speed High-Resolution Low-Phase Noise Clocks". In: *VLSI-SoC: Forward-Looking Trends in IC and Systems Design*. Ed. by J. L. Ayala, D. Atienza Alonso, and R. Reis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012). m Mueller C gates (detailed sub-circuit on the right) are connected in a circular pattern. All of them oscillate at the same frequency but are offset to each other. D-flipflops sample them and an xor-gate combines the sampled m F_i outputs of Mueller C gates. A D-flipflops samples the output, resulting in a random signal R at the output.

2.3.3 Entropy assessment

Since the use-case we select in this work is an RNG, we need methods to judge its generated random numbers. This can be done by calculating the entropy of the RNG. Entropy was defined by Shannon[36] as

$$H(X) = -\sum_{i-1}^{n} P(x_i) log P(x_i). \tag{2.4}$$

Equation (2.4) describes a metric for the entropy of a random variable X with possible outcomes $x_1, \ldots, x_n$ which occur with probability $P(x_1), \ldots, P(x_n)$.

This formal equation requires knowledge about the probability of outcomes $P(x_i)$. These in turn cannot be calculated without a precise physical model of the underlying random process, which can be very hard to achieve.

To simplify the quantification of the entropy, we use stochastic tests. Stochastic tests are a tool from probability theory and quantify the probability of a hypothesis being valid or not. The hypothesis in this case is that the numbers generated by an RNG are random. The criteria the random numbers have to adhere to are defined as:[37]

1. All possible outcomes should have the same chance of occurrence,
2. Prediction of outputs should be impossible,
3. Numbers must be generated faster than they are consumed and
4. No predictable patterns should occur.

If the probability of meeting the hypothesis is high enough, stochastic tests can be used as an approximation for the entropy. This is common practice for RNGs used for security purposes.

In the following, we present four commonly used methods for stochastic testing of RNGs.

Bias

The easiest property to check is the *bias* of the RNG (Item Criterion 1). For binary random numbers, this means that the number of ones and the number of zeroes in the sample output is equal (probability of occurrence is 0.5 both for ones and for zeroes). It can be checked by counting the ones and zeroes in a binary stream and dividing by the length of the stream (Equation (2.5)).

$$p_{bias} = \frac{N_{ones}}{N} \tag{2.5}$$

However, it is very simple to satisfy this requirement. Imagine a toggle circuit changing on every clock cycle. Its output is unbiased but certainly not random.

Therefore, this property is merely necessary, but not sufficient to denominate the underlying process random.

[36] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948).
[37] L. Afflerbach. "Criteria for the assessment of random number generators". en. In: *Journal of Computational and Applied Mathematics* 31.1 (July 1990).

NIST test suite

The NIST test suite is a battery of stochastic tests on RNGs[38] published by the *National Institute of Standards and Technology (NIST)*. It consists of the mathematical description of 15 stochastic tests. Multiple software implementations of the NIST test suite exist that apply these tests with a wide range of input parameters to evaluate one or more streams of random numbers. The default implementation requires at least 100-10^6 bits of input data, depending on the test. All tests can be run on the same input data, this implies that one million bits of input data suffice for the NIST test suite to run all tests.

Dieharder test suite

A second set of stochastic tests is the *dieharder* test suite.[39] It is based on the *GNU scientific library (GSL)*. Dieharder does not only consist of tests but also bundles a set of random number generators and is primarily used to test software RNGs using the GSL RNG interface. This is due to the fact that it can consume over 10^{12} bits of data to perform all tests once.

However, it also supports testing on binary input files, which makes them suitable to test RNG implemented in hardware without providing a GSL interface.

Federal Information Processing Standard (FIPS) 140-2 tests

The FIPS 140-2 standard specifies requirements for cryptographic modules including both hardware and software components for certification by the NIST.[40] They are defined over the model of the RNG. However, stochastic tests were developed that aim to test for FIPS 140-2 compliance. An RNG passing the FIPS 140-2 stochastic tests is not guaranteed certification by NIST. In comparison to the dieharder suite and the NIST suite, the stochastic test suites devised for FIPS 140-2 are easier to pass, a fact that is criticized by researchers.[41] It was shown that the FIPS tests are unable to identify adversarial biases on simple biased-by-design test RNGs. They are deprecated by official standards, but nonetheless still widely used by hardware designers, e.g. in hardware-level self-test schemes.[42]

2.3.4 Certificates

Designers of TRNGs may request a third-party entity to certify their design. This assures their customers that the design is reliable. The main certificates are:

1. The ***Federal Information Processing Standard (FIPS)*** certificate[43] is split into four security levels. The lowest level, level 1, targets personal computers. The highest level, level 4, defines requirements for physically unprotected but security critical devices.
   NIST provides stochastic tests as a necessary but not sufficient requirement for the FIPS certificate. The tests are discussed in Section 2.3.3.

[38] A. Rukhin et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications.* 2002.
[39] R. G. Brown. *Dieharder, A Random Number Test Suite. version Version 3.31. 1, Duke University Physics Department.* https://webhome.phy.duke.edu/~rgb/General/dieharder.php. 2004. (Visited on 02/02/2022).
[40] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules.* en. Tech. rep. Dec. 2002.
[41] D. Hurley-Smith, C. Patsakis, and J. Hernandez-Castro. "On the unbearable lightness of FIPS 140-2 randomness tests". In: *IEEE Transactions on Information Forensics and Security* (2020).
[42] Ibid.
[43] National Institute of Standards and Technology, *Security Requirements for Cryptographic Modules.*

2. The **AIS-20/31** certificate defines requirements on the stochastic model of the TRNG.
   It is standardized by NIST. Unfortunately, NIST provides no stochastic tests for *AIS-20/31*. However, the stochastic models for all TRNGs presented in this work adhere to the requirements of *AIS-20/31*.[44]

[44] Petura et al., "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices".

Chapter 3

Proposed algorithm

Assume we want to place and route a *random number generator (RNG)* according to the timing requirements discussed in Section 2.2.3. This requires manual effort, as detailed in Section 3.1. Manual *placement and routing (PnR)* requires skilled labor and is time-consuming. After years of experience with the *Vivado Design Suite* for *Xilinx* it still takes the author two months to place and route a 64-stage *self timed ring (STR)-based* true random number generator (TRNG) *(STR-TRNG)*. To speed up this process, we propose an algorithm that performs the PnR automatically.

The algorithm acts on timing constraints we defined with security primitives in mind. Designers can set them directly in *hardware description language (HDL)* code, as described in Section 3.2. We detail the proposed placement algorithm in Section 3.3 and the proposed routing algorithm in Section 3.4.

In a first step, the functional constraints are translated to physical constraints on the routes of a design (Section 3.2). Then, the algorithm places the design so that all constrained cells are placed in a manner to facilitate the routing step (Section 3.3). This includes placing cells that are connected to each other in proximity and placing cells that are part of a combinatorial loop in a circle. After the placement step, the algorithm routes the constrained nets accordingly. (Section 3.4).

3.1 State of the art

Timing requirements are one driving factor of PnR algorithms. Unfortunately, as discussed in Section 2.2.3, the timing constraints that are widely used do not support PnR of the security primitives discussed in Section 2.1. Users implementing security primitive *intellectual property cores (IP cores)* need to do the routing manually.[1] We illustrate the state-of-the-art design flow in Figure 3.1 (left).

Once the functional design is complete, the hardware implementation begins. We manually route a STR-TRNG (as described in Section 2.3) design as a baseline to compare our algorithm to. The target architecture is a *Xilinx Virtex UltraScale+ VCU118* development board.

[1] O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016.

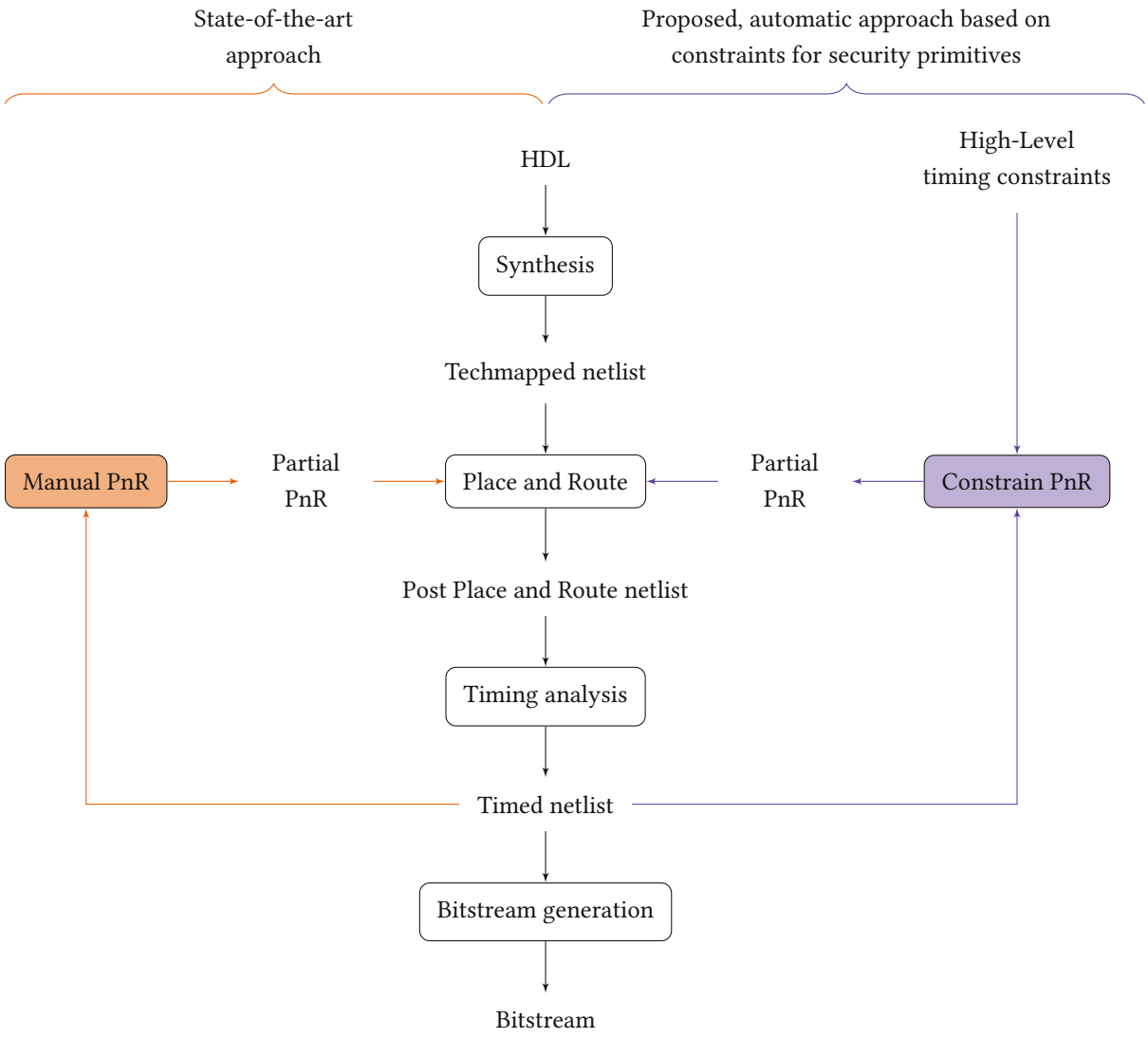


Figure 3.1: Comparison of the current state-of-the-art design flow for field-programmable gate arrays (left) and our proposed flow to facilitate the placement and routing of security primitives (right). We introduce high-level timing constraints specific to security primitives. We can then avoid the time-consuming manual placement and routing step with our proposed algorithm that respects these timing constraints.

As inexperienced users, our approach is mostly based on guesswork, trial, and error. This manual PnR process can be outlined by the following steps:

1. Visually arrange the cells in a circle.
2. Select a group of nets that need to have equal delay.
3. Route the group.
4. Obtain and evaluate static timing.
5. If all nets have similar delay, move on to Step 8.
6. Select one net that does not satisfy the constraint of it's group and reroute it.
7. Go to Step 4.

```
1  module A (
2          input wire [15:0] ai,
3          output wire [7:0] ao
4  );
5          B b1 (ai[15:8], ao[7:4]);
6          B b2 (ai[ 7:0], ao[3:0]);
7  endmodule
8
9  module B (
10         input wire [7:0] bi,
11         output wire [3:0] bo
12 );
13         C c1 (bi[3], bi[2], bo[3], bo[0]);
14         C c2 (bi[1], bi[0], bo[0], bo[1]);
15         C c3 (bi[1], bi[0], bo[1], bo[2]);
16         C c4 (bi[1], bi[0], bo[2], bo[3]);
17 endmodule
18
19 module C (
20         (* DELAY_GROUP = "X" *)   input wire cx1, // local scope
21         (* DELAY_GROUP = "X" *)   input wire cx2, // local scope
22         (* DELAY_GROUP = "*+Y" *) input wire cy,  // global scope
23         (* DELAY_GROUP = "1+Z" *) output wire cz  // relative scope
24 );
25         assign cz = cx1 & cx2 & cy;
26 endmodule
```

Listing 3.1: Example design using the *DELAY_GROUP* attribute

8. If there are more groups to route, select the next delay group in Step 2.

This is a tiring process, and it has to be repeated for each target architecture. The implementation of a 64-stage STR-TRNG takes a skilled developer over two months of full time work. This manual effort has a huge impact on design cost.

We propose an automated design flow. We extend state-of-the-art PnR tools with a timing constraint designed with security primitives in mind. The tiresome, manual flow in the left part of Figure 3.1 transforms into the automated flow without the need of user-interaction as shown on the right side of Figure 3.1.

3.2 Design preparations

The first step in automating the manual process is to formulate the requirement of equal delay, discussed in Section 2.2.3, in the STR-TRNG IP-core.

We propose an HDL attribute called *DELAY_GROUP*. It allows the assignment of nets to hierarchically named groups. The algorithms discussed in Sections 3.3 and 3.4 are designed to implement the design so that nets in the same group have equal (or similar-enough) delay. We give example annotations for the *Verilog* HDL, but the attribute can be set in other languages as well (e.g., VHDL attributes, *Electronic Design Interchange Format (EDIF)* properties, *tool command language (TCL)* properties).

Delay group names are scoped to a hierarchy level. Upon defining a *DELAY_GROUP* attribute, we can specify where the group should be valid. Once the synthesis tool reads the design, our algorithm expands all names to absolute names. These are valid globally for the design. We give an example design in Listing 3.1 and illustrate its hierarchy in Figure 3.2 to aid the explanation of the possible scopes.

Delay groups can be defined:

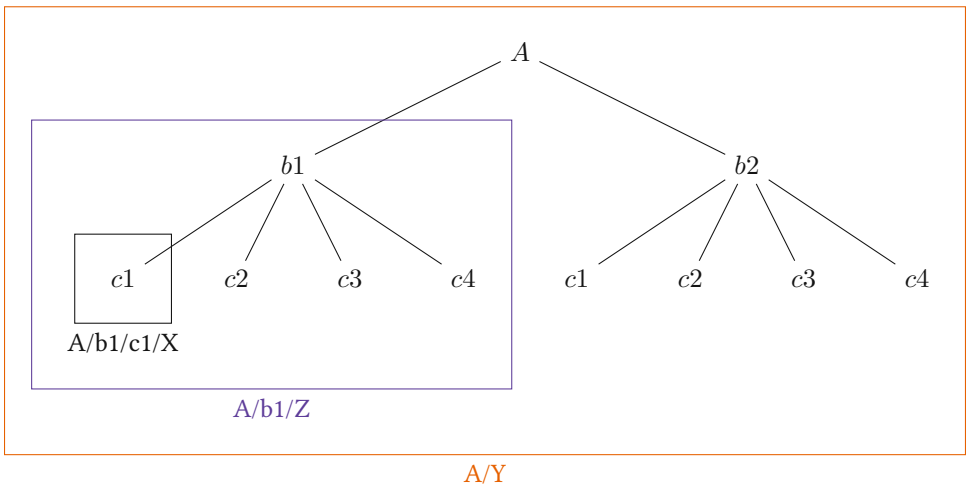


Figure 3.2: Design hierarchy of Listing 3.1. We show the top module A instantiating two modules of type B ($A/b1$ and $A/b2$) that each instantiate four modules of type C ($A/b1/c1$, $A/b1/c2$, $A/b1/c3$ and $A/b1/c4$ for $A/b1$ and $A/b2/c1$, $A/b2/c2$, $A/b2/c3$ and $A/b2/c4$ for $A/b2$). The wires in the modules of type C are annotated with *DELAY_GROUP* attributes. We use three distinct scoping types for delay groups and highlight one of each using boxes. Global, shown in the orange box, hierarchical, shown in purple and local, shown in black.

1. **Local to the module they are defined in.**

   If simply a name is given, the delay group is local to the module the net is defined in.

   This is the case for group X in module C of Listing 3.1. Wires $cx1$ (Line 20) and $cx2$ (Line 21) will be routed to have the same delay within the same instance of the module. Each instance will have its own delay group. Wires $A/b1/c1/cx1$ and $A/b1/c2/cx1$ will not be routed to have the same delay, as they reside in different instances of C.

   The absolute delay group names deducted by the synthesis tool are *A/b1/c1/X* for *A/b1/c1* (highlighted in a black box in Figure 3.2), *A/b1/c2/X* for *A/b1/c2*, *A/b1/c3/X* for *A/b1/c3*, …

2. **Within *n* steps of hierarchy above the instantiation of the module.**

   If nets that need to be assigned to the same delay group reside in different modules, we need a way to match them together. This can be done by specifying in how many levels of hierarchy above the specification of the net the group needs to be valid.

   In Listing 3.1 this is the case for group Z (Line 23). We defined it to be valid one level of hierarchy above the instantiation of the module ($1 + Z$). We can check the hierarchy given in Figure 3.2 to understand that Group Z is valid for instances of *B*. $A/b1/c1/cz$ and $A/b1/c2/cz$ will be routed to have the same delay. $A/b1/c1/cz$ and $A/b2/c1/cz$ will not be routed to have the same delay, as they reside in different instances of B.

   The absolute delay group names deducted by the synthesis tool are *A/b1/Z* (highlighted by the purple box in Figure 3.2) and *A/b2/Z*.

3. **Globally.**

   Users can also specify global delay groups. This is done in the same way as Item 2, but an asterisk is used instead of specifying levels of hierarchy.

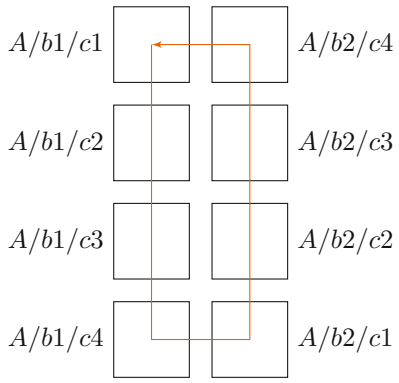


Figure 3.3: Circular placement of the combinatorial loop in Listing 3.1, consisting of 8 cells

We did this for delay group Y (Line 22). $A/b1/c1/cy$, $A/b1/c2/cy$, $A/b1/c3/cy$, $A/b1/c4/cy$, $A/b2/c1/cy$, $A/b1/c2/cy$, $A/b1/c3/cy$, and $A/b2/c4/cy$ will all be routed to have the same delay.

The absolute delay group name deducted by the synthesis tool is *A/Y*, highlighted by the orange box in Figure 3.2. Nets attributed with this delay group, no matter where in the design, are assigned to the same group.

As we can see, the scoping of delay groups widens the application domain significantly. In our use-case example, the STR-TRNG in Section 2.3, we make use of this feature to create one delay group per Mueller C-gate.

3.3 Placement step

Before connecting the individual cells with nets, we have to place them. The placement step is not imperative to meet the timing requirements, but adequate placing can aid the router to find sufficient routes faster while requiring less resources. The main goals of the placer are:

1. Identify cells that are part of combinatorial loops consisting of nets that were assigned the same delay group and place them in a circle, and

2. Place cells that are connected by nets within the same delay group close to each other.

Goal 1 assures that cells within a combinatorial loop are as equidistant as possible. This aids the router to find routes with equal delay.

We use an adapted version of Tarjan's algorithm[2] implemented in TCL to find combinatorial loops in the design. We achieve circular routing by counting the cells involved in a combinatorial loop. Half of them are routed, one after the other, in a column going downwards. We route the other half in an adjacent column going upwards. This assures all cells have equal distance to their neighbors in the loop. The equidistant placement of the cells makes it easier to find routes between them that have equal delay.

We illustrate the circular placement for the example design in Listing 3.1 in Figure 3.3. The 8 instances of module C are connected in a combinatorial loop. The TCL implementation of the placement step is given in Listing 3.2.

[2] R. Tarjan. "Depth-first search and linear graph algorithms". English. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE Computer Society, Oct. 1971.

```
proc placeCells {cells x starty dir} {
      upvar $starty y
      if {[llength $cells] == 0} {
            return 0
      }
      if {[llength $cells] == 1} {
            set cell [get_cells [lindex $cells 0]]
            if {[get_property IS_PRIMITIVE $cell]} {
                  if {[get_property GROUPING $cell]} {
                        puts "place_cell $cell SLICE_X${x}Y${y}"
                        if [catch "place_cell $cell SLICE_X${x}Y${y}"] {
                              if {$dir} {
                                    incr y
                              } else {
                                    set y [expr {$y-1}]
                              }
                              placeCells $cell $x y $dir
                        } else {
                              if {$dir} {
                                    incr y
                              } else {
                                    set y [expr {$y-1}]
                              }
                              set_property IS_BEL_FIXED 1 $cell
                              set_property IS_LOC_FIXED 1 $cell
                        }
                  }
            } else {
                  set children [get_cells -hierarchical -filter "PARENT == $cell"]
                  placeCells $children $x y $dir
            }
            return 1
      }
      set loopCells [findLongestLoop [get_cells $cells]]
      set maxXLeft 1
      for {set i 0} {$i<[expr ([llength $loopCells]-1)/2]} {incr i} {
            set width [placeCells [lindex $loopCells $i] $x y $dir]
            if {$width > $maxXLeft} {
                  set maxXLeft $width
            }
            set cells [lsearch -inline -all -not -exact $cells [lindex $loopCells $i]]
      }
      set backupX $x
      set newX [expr {$x + $maxXLeft}]
      set maxXRigth 1
      for {set i [expr ([llength $loopCells]-1)/2]} {$i<[expr [llength $loopCells] -1]} {incr i} {
            set width [placeCells [lindex $loopCells $i] $newX y [expr {!$dir}]]
            if {$width > $maxXRigth} {
                  set maxXRigth $width
            }
            set cells [lsearch -inline -all -not -exact $cells [lindex $loopCells $i]]
      }
      set x $backupX
      for {set i 0} {$i<[llength $cells]} {incr i} {
            placeCells [lindex $cells $i] $x y $dir
      }
      return [expr {$x + $maxXLeft + $maxXRigth}]
}
```

Listing 3.2: Placer implementation

3.4 Routing step

The router tries to connect the pins of already placed cells. For each delay group it tries to find routes so the difference in delay of all nets in the delay group is within a pre-defined relative threshold. This is achieved by iterating through all delay groups and routing one after the other.

The routing could be implemented in two distinct ways:

1. Extract the complete topology, including timing details, for a *field-programmable gate array (FPGA)* and implement a router or

2. Use an existing router and existing timing models to find routes that suit our requirements.

Item 1 requires knowledge of the topology of the target hardware. Vendors don't always provide that. We choose the higher level approach, as described by Item 2. It integrates well with the state-of-the-art design approach (Figure 3.1). We implement it to work with *Xilinx Vivado* but it is portable to other synthesis tools that support TCL. We target the *Xilinx Ultrascale+* architecture, but the design principle can be transferred to other FPGA architectures. The only requirement is that the corresponding toolchain supports TCL, which is satisfied for almost all state-of-the-art design tools.

We outline the main strategy employed by the router in Algorithm 1. The router's goal is to satisfy all timing requirements set by the delay-group attribute. It does so by looking for delay-groups that break the timing requirement. The router re-routes the fastest (lowest delay) nets in the group until the routing satisfies the timing requirement. The strategy of manual routing (described in Section 3.1) inspires this approach. We give details on the steps of particular interest below.

Data: Placed design containing delay groups, relative delay difference threshold
Result: Placed and routed design where the delay difference of nets in the same group is within a pre-defined relative threshold

```
1  Create map groups = (delayGroupName) → {nets};
2  while groups ≠ ∅ do
3      group = groups[0];
4      if satisfied(group) then
5          remove group from groups;
6          continue;
7      end
8      determine slowestNet;
9      fastNets = {net|delay(slowestNet) * threshold > delay(net)};
10     while fastNets ≠ ∅ do
11         unroute fastNets;
12         determine minDelay and maxDelay for fastNet[0] to satisfy criterion;
13         route fastNet[0] to have delay within [minDelay, maxDelay];
14         if delay(fastNet[0]) > delay(slowestNet) then
15             slowestNet = fastNet[0];
16         end
17         fastNets = {net|delay(slowestNet) * threshold > delay(net)};
18     end
19 end
```

Algorithm 1: Routing algorithm

Mapping of group names to nets (Line 1)

First, as discussed in Section 3.2, we derive absolute group names for all nets with the *DELAY_GROUP* attribute. All nets with the same absolute delay group name are placed in a list. A map, resolving absolute group names to lists of nets, is created. Initially, all groups are assumed not to satisfy the timing criterion (delay difference within a pre-defined relative threshold).

Checking if nets meet criterion (Line 4)

We perform *static timing analysis (STA)* and find the fastest (smallest delay) net N_f and the slowest (highest delay) net N_s. Only if the delay difference is within the relative threshold the criterion is met. The criterion can therefore be expressed as:

$$delay(N_s) * threshold < delay(N_f) \tag{3.1}$$

Determine which nets to re-route (Line 9)

If we find a group that violates the criterion in Equation (3.1), we need to change the routing. The easiest way to do so is to make the fastest nets take a detour to increase their delay.

Determine acceptable delay boundaries for net (Line 12)

To determine the delay-boundaries for the net we selected, we need to check which possible values would result in our timing criterion in Equation (3.1) being satisfied.

We illustrate this problem in Figure 3.4. It shows a timeline and delay values for four nets A, B, C and D. Net A (purple) is very fast and is therefore selected to be rerouted. Nets B, C and D (orange) satisfy the timing criterion. We want to find the boundary delay values for net A to also satisfy the criterion. As our criterion checks the delay of the fastest against the delay of the slowest net, there are three possible ways for A to satisfy the criterion:

- $delay(B) < delay(A) < delay(D)$
- $delay(A) > delay(D) * threshold$
- $delay(B) > delay(A) * threshold$

This results in the delay interval $[delay(C) * threshold, delay(B)/threshold]$, illustrated by a thick line in Figure 3.4.

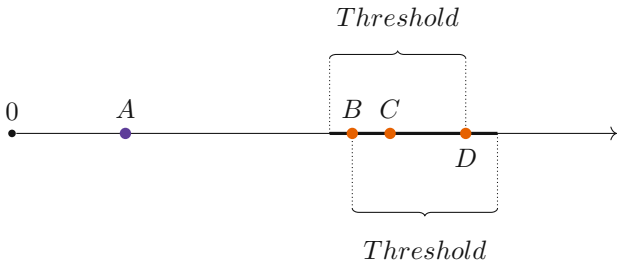


Figure 3.4: Example delay values of nets A, B, C and D on a time bar

Route net within allowed delay boundaries (Line 13)

The nets selected in Line 9 must now be routed within the delay boundaries determined in Line 12. We use *Vivado's* built-in *route_design* TCL command to route the nets accordingly.

Chapter 4

Experiments

This chapter elaborates the experiments to evaluate the performance of the *placement and routing (PnR)* algorithm discussed in Chapter 3. Section 4.1 motivates the selection of the cores we use for the experiments. The experimental setup is described in Section 4.2. We perform *static timing analysis (STA)*, oscilloscope measurements and capture generated random numbers, as detailed in Section 4.3. We expect our proposed algorithm to outperform the core placed and routed by Vivado and match the performance of the manually placed and routed core, as discussed in Section 4.4. In Section 4.5, we evaluate limitations of the experimental setup that may affect the measured data. Finally, in Section 4.6, we give the results of the experiments.

4.1 Selection of Device under Test

Out of the *random number generator (RNG)* cores compared by Petura et al.[1] we choose the *self timed ring (STR)-based* true random number generator (TRNG) *(STR-TRNG)* to evaluate our PnR algorithm, because out of all the compared cores it is the fastest and most efficient while relying on precise PnR, making it the ideal example for our experiments (Section 2.3).

For our purposes we implemented a generic STR-TRNG design using five parameters:

1. *STR_Size* is the number of Mueller-gates in the STR.

2. *RO_Size* is the number of delay-elements in the *ring oscillator (RO)* that samples the STR.

3. For large STRs, the different delays of different paths within the *exclusive-or (XOR)* gate near the output may have an impact on performance. Therefore, the output of individual stages is sampled first, then passed to a tree of synchronous XOR gates.[2] The maximum number of XOR-gates within a branch of that tree is defined by *CONCURRENT_XOR*.

4. *BRAM_WIDTH*, the width of the *Block RAM (BRAM)* used to buffer the random data.

[1] O. Petura et al. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016.

[2] A. Cherkaoui et al. "A Self-Timed Ring Based True Random Number Generator". In: *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*. 2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems. May 2013.

5. *BRAM_DEPTH*, the depth of the BRAM used to buffer the random data.

These parameters directly affect the digital circuitry of the design. We must therefore generate an individual bitstream for each combination of parameter values we want to test in our experiments. We select $BRAM\_WIDTH = 32$ and $BRAM\_DEPTH = 1048576$ for all experiments. This exhausts the BRAM on our development board and gives us a buffer of $4,194,304$ bytes (around $4MB$). As the functionality of the RO we use in the STR-TRNG design is not dependent on its routing, we select an RO of 53 inverters for all experiments to keep the number of necessary bitstreams low.

The number of tokens in the ring is configurable at runtime. In contrast to the parameters listed above, the number of tokens does not affect the circuitry but merely the reset value of each STR stage and can therefore be changed during reset without flashing a different bitstream. We allow this operation through a *universal synchronous receiver/transmitter (UART)* interface. The UART interface interprets each received byte as an unsigned integer N_{mt}. The maximum number of tokens is $STR\_Size/2$. N_{mt} represents the number of tokens missing to full occupancy. The number of tokens N_t circulating in the ring is therefore given as in Equation (4.1).

$$N_t = \frac{STR\_SIZE}{2} - N_{mt} \tag{4.1}$$

Note that N_t and *STR_SIZE* must be coprime for the STR to work as a precise clock.[3]

After the number of tokens is changed, the STR is reset by the UART in order to distribute the correct amount of tokens in the ring.

4.2 Setup

To conduct the measurements, we flash the STR-TRNG onto a Xilinx Virtex UltraSCALE+ *field-programmable gate array (FPGA)* on a VCU118 Rev2.0 development board.[4] We connect the board to a host computer. We use a JTAG connection for programming over JTAG and a UART connection for data extraction and configuration. To evaluate the quality of the two oscillators contained in the RNG we connect a high-speed oscilloscope (Agilent Infiniium DSO90804A, 8GHz, 40GSa/s) to the *Low voltage differential signaling (LVDS)* output *USER_SMA_CLK*.

We choose the VCU118 development board for it's *SubMiniature version A (SMA)* connector capable of high frequency transmission. We want to measure two output clocks, the output of the RO and of the STR. Unfortunately, only one such output is available. To mitigate this, we use a switch so select which clock is visible at the output.

The UART interface is not fast enough to transfer random numbers in real time. We use a buffer to store a sequence of random numbers that is transmitted once the buffer is full.

The complete setup is illustrated in Figure 4.1. We show pictures of hardware we use in the setup in Figure 4.2.

4.3 Methodology

During the measurements we acquire data from four sources:

[3] A. Cherkaoui et al. "A Very High Speed True Random Number Generator with Entropy Assessment". In: *Cryptographic Hardware and Embedded Systems - CHES 2013*. Ed. by G. Bertoni and J.-S. Coron. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[4] Xilinx. *UG1224 VCU118 Evaluation Board User Guide*. Ed. by Xilinx. 1.4. Oct. 17, 2018.

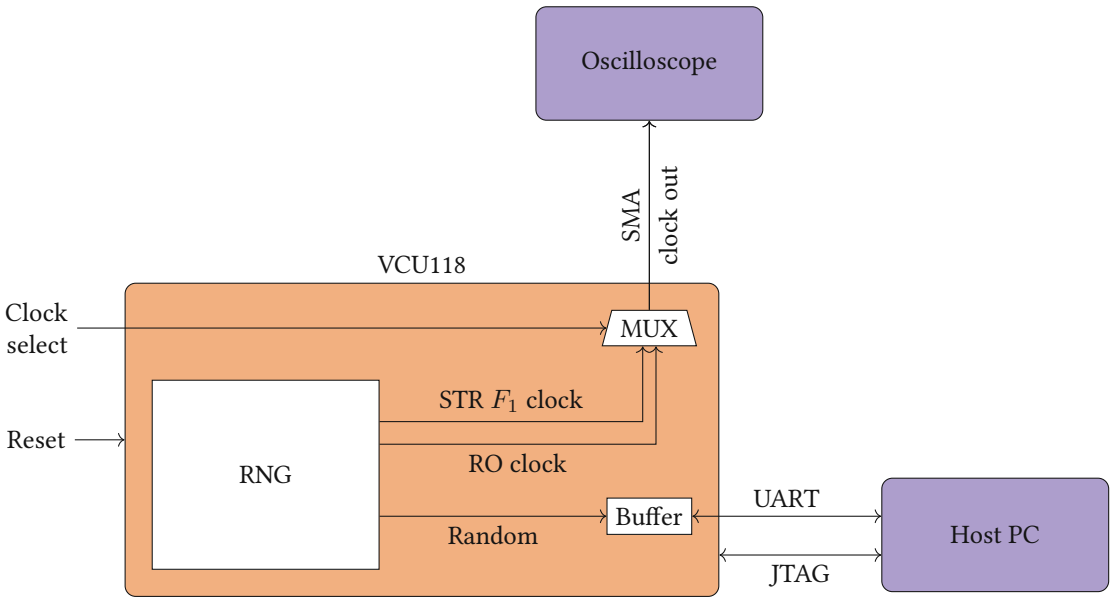


Figure 4.1: Laboratory setup. The system under observation, a Xilinx VCU118 development board, is connected to an oscilloscope via SMA and to a Host PC via UART and JTAG. We use the oscilloscope to measure the produced clock signals. We use the Host PC for configuration and to store the generated random numbers.

- Implementation time,
- STA results to confirm the timing requirements are met,
- Oscilloscope measurements to evaluate the oscillation of the RO and STR, and
- Generated random numbers to assert the quality of the TRNG.

Data acquisition using the oscilloscope needs manual intervention to select the clock that is visible to the oscilloscope, and therefore has to be carried out manually. Gathering random data over UART is done by a script and performed autonomously for a high number of implementations and configurations.

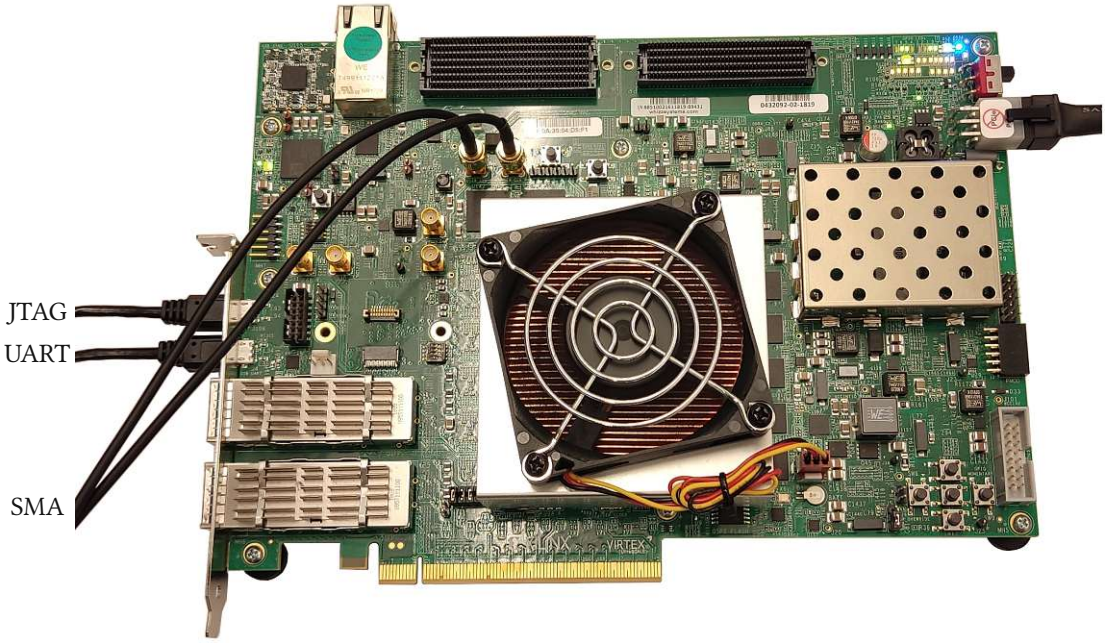
We carry out the experiments with three kinds of PnR in order to answer our research questions:

1. Default Vivado PnR to prove that the STR-TRNG does not perform adequately if not routed adequately (default Vivado configuration without knowledge of our novel timing constraint),
2. A manually placed and routed design to show the level of performance an experienced designer can reach, and
3. An implementation generated by the proposed PnR algorithm to make sure we meet the performance of the manually placed and routed design without the effort of manual placement and routing.

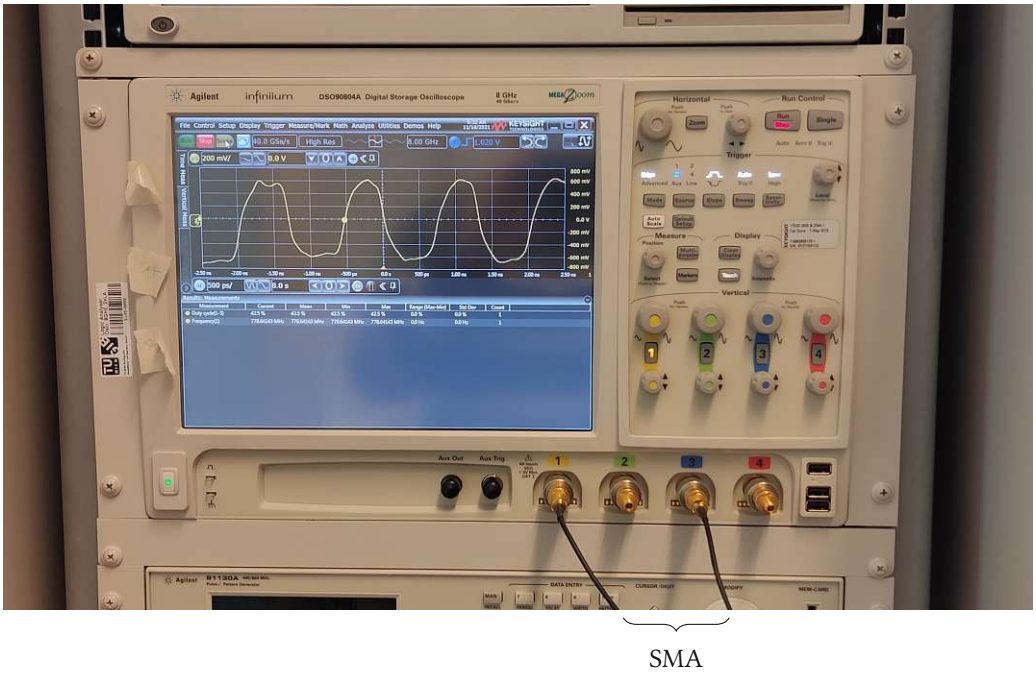
We gather data for multiple configurations, varying the size of the STR (*STR_SIZE*) and the number of tokens in the ring N_t. For the manually placed and routed implementation we only use one size ($STR\_SIZE = 64$). Due to the long time required for manual PnR it is prohibitively expensive to implement more than one size.

We list the steps carried out in the laboratory below:

1. Perform STA in *tool command language (TCL)*
2. Generate bitstreams



(a) Xilinx Virtex UltraSCALE VCU118 Development Board



(b) Agilent Infiniium DDSO90804A Digital Oscilloscope

Figure 4.2: Picture of the laboratory setup. The development board is shown in Subfigure a. It is connected to the oscilloscope (Subfigure a) using low voltage differential signaling.

3. Test bitstreams individually (performed manually)

   (a) Flash bitstream onto device

   (b) Select RO clock

   (c) Save waveform dump and screen of the oscilloscope

   (d) Note clock frequency and duty cycle of the RO

(e) Select STR clock

(f) Save waveform dump and screen of the oscilloscope

(g) Note clock frequency and duty cycle of the STR

(h) Reconfigure with different number of tokens (via UART)

4. Batch-test bitstreams (performed by a script)

(a) Flash bitstream onto device

(b) Iterate over number of tokens (via UART)

(c) Store generated random numbers

A high-level overview is given in Figure 4.3.

4.4 Expectations

Both for the manually placed and routed implementation and for the implementation generated by the proposed algorithm we expect to see the following improvements over the default Vivado PnR:

- Steeper edges in the clock and constant duty cycle of around 50% due to the timing-dependencies of the STR,
- Frequency should be a little lower compared to the default Vivado design due to the routes being longer in general, and
- More stochastic tests pass due to the improved quality of the clock.

4.5 Limitations

The precision of the measured data is limited by the experimental setup. In the following, we highlight and discuss limitations and how the experimental setup could be expanded to mitigate them.

4.5.1 Collection of random numbers

We provide a UART interface to configure the design and retrieve random numbers. The TRNG is faster than the maximum bit rate of the UART interface. This impedes us to retrieve the random numbers in real time. We therefore buffer them on BRAM in the FPGA. The TRNG fills up the buffer. Once the buffer is exhausted, all collected random numbers are transferred to the host PC over UART. This implies that the maximum number of consecutively generated random numbers is limited by the BRAM capacity, which in turn is limited by the FPGA in use.

The capacity of the BRAM on the *VCU118* development board is sufficient to perform meaningful *NIST 800-22* tests[5] and for *rngtest* (Section 2.3.3). The *dieharder test* suite requires more data than the BRAM can store (as discussed in Section 2.3.3) and is therefore not used.

[5] A. Rukhin et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications.* 2002.

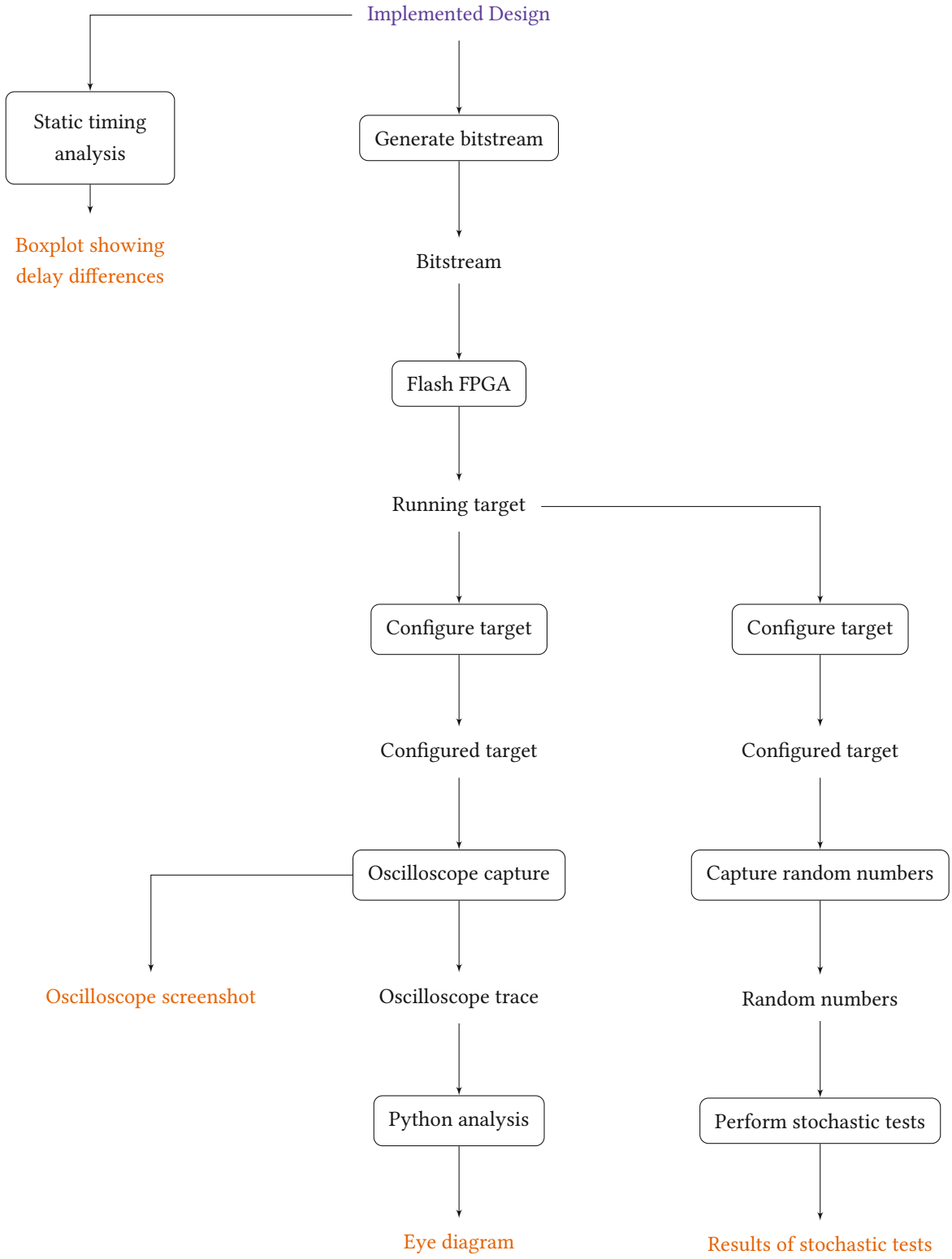


Figure 4.3: Execution of experiments. We use static timing analysis, an oscilloscope, python scripts, and stochastic test suites to gather the data we show in Section 4.6 (orange leafs).

This issue could be mitigated by:

1. Using an FPGA with more BRAM capacity,
2. Using fast external storage (e.g., *Synchronous dynamic RAM (SDRAM)*), or
3. Implementing an interface capable of transmitting the random numbers in real time (e.g., *Peripheral Component Interconnect Express (PCIe)*).

4.5.2 Metering

We expect the frequency of the STR clock we use in our use-case design to be above $1GHz$. The exact frequency depends on the routing and is not predictable. We need appropriate high-frequency oscilloscopes to capture meaningful data. The 8GHz, 40GSa/s Agilent Infiniium DSO90804A we use is at its limits. We cannot capture the output of the STR as a whole, but can only capture the output of a single stage (E.g. F_1 instead of clk in Figure 2.9). This means the measured clock is *STR_SIZE* slower than the clock that is sampled by the STR-TRNG. This could be improved by using a faster oscilloscope. For the larger designs, even the fastest oscilloscopes available ($110GHz$[6]) would not suffice to capture the final clock output.

4.5.3 Clock interfaces and connectors

A second limitation regarding the measurement of the generated clock, closely related to the limitation discussed in Section 4.5.2, lies within the interfaces and connectors used to route the clock signal to the measurement device. To avoid quality impairments in this route, we use SMA connectors and cables rated for $12.4GHz$ in a LVDS setup. As there is only one pair of SMA output connectors available (a single signal output requires two connectors due to the differential signaling), we can only measure one clock at a time. We have to use a multiplexer to select if the RO or STR single-stage clock is visible at the output.

We could measure both of them at the same time (or even the RO clock and multiple STR single-stage clocks) given a developement board with more SMA outputs, a custom board, or an *FPGA Mezzanine Card (FMC)* breakout board.

4.6 Results

This section evaluates the results of the experiments. We first compare our proposed design flow to the state-of-the-art design flow in Section 4.6.1. We then divide the results into STA (Section 4.6.2), clock behavior (Section 4.6.3) and stochastic tests (Section 4.6.4).

4.6.1 Design flow automation

The main metric we can use to compare our proposed, automated, design flow to the state-of-the-art manual approach is time spent for implementation. However, this metric has two flaws:

1. The time spent during manual PnR is subjective to the designer that performs it. It is out of the scope of this project to perform a user-study with a sufficient number of users to draw a significant conclusion.

[6] Keysight. *Infiniium UXR-Series Oscilloscopes*. Ed. by Keysight. May 11, 2021.

2. The time spent during manual PnR is human lifetime of an expert *integrated circuit (IC)* designer. The time spent by the algorithm is machine time. We consider the former much more expensive in terms of cost.

Due to a lack of alternatives we use implementation-time as a metric anyhow. We still consider the results representative due to the large gap between the measured times. We give empirical numbers for PnR of a single 64-stage STR in Table 4.1. The time for manual PnR is the time spent by the author of this work (highlighted in orange in Table 4.1). The time for the proposed algorithm is the time spent by an Intel Xeon Platinum 8160 2,1GHz 24C/48T CPU. We do not need to prepare the hardware design for manual PnR, as manual PnR does not rely on constraints defined in the design itself.

Step	Manual PnR	Proposed algorithm
Preparation	Not needed	1h
Placement	8 hours	9 minutes
Routing	320 hours	4 hours, 11 minutes
Sum	328 hours	5 hours, 20 minutes

Table 4.1: Implementation-time comparison between manual placement and routing (by a skilled engineer) and our proposed algorithm (by a server). We highlight the time a skilled engineer needs to spend during the implementation in orange.

We conclude that automatic routing is over 60 times faster compared to manual routing for our use-case example.

4.6.2 Static timing analysis

The main goal of our algorithm is to meet our novel timing requirement for security primitives (Section 2.2.3). The most direct metric we use to quantify our algorithm's performance is the evaluation of the static timing of the resulting implementation. The interesting traits are the differences in the propagation delays *within* the stages of the STR and *between* the stages of the STR (see Section 2.3). The delay differences of all cirtical paths in the STR-TRNG design are plotted in a boxplot in Figure 4.4. The STR size is given on the x-Axis, with the relative delay differences on the y-Axis. They are grouped by STR size. The colors differentiate the PnR method. Our goal is to minimize the relative delay differences between the different nets, given on the y-Axis. From the box plot it is easy to read statistical properties as minimum, median, maximum, and 25%- and 75% quantiles.

We give results for the default Xilinx Vivado PnR (without knowledge of timing constraints — orange), for manual PnR (by a skilled engineer — light gray) and by our proposed algorithm (fully automated — purple). Results for manual PnR are only available for an STR consisting of 64 stages, we consider the high effort of manually routing all the configurations out of scope for this work.

We can tell immediately that *Xilinx Vivado's* default implementation results in the highest relative delay differences (*Median* ≈ 0.5). This means half the nets have at least 1.5 times more delay than other nets in their delay group. This is expected, as Vivado has no knowledge about our novel timing requirement. Instead, it is driven mostly by physical resource optimization. The box plot tells us that, in all configurations of the STR, Vivado manages to route some nets with equal delay, while some nets are up to 1.9 times longer than others.

The problems of the default Vivado implementation are solved by the implementation performed by a skilled engineer. *All* relative delay differences lie well below the default implementation's median. The highest relative delay differences

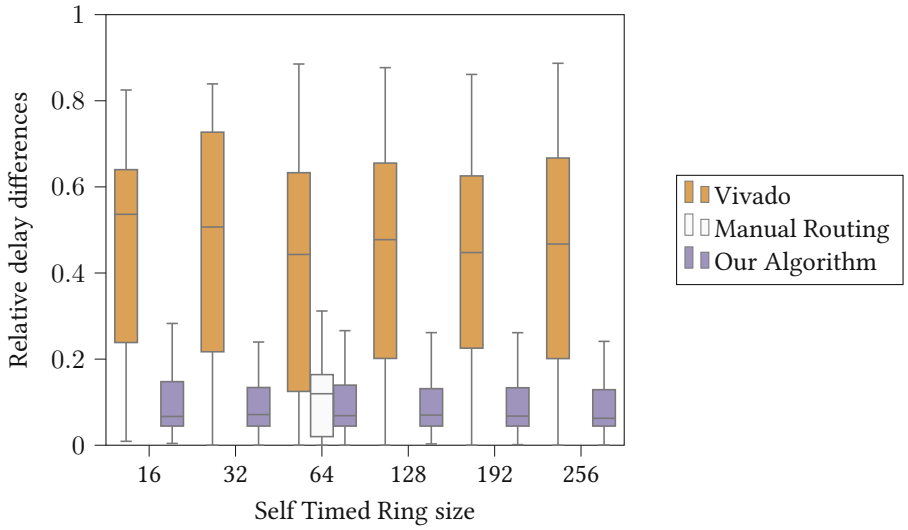


Figure 4.4: Results of the static timing analysis, comparing default Xilinx Vivado placement and routing, manual placement and routing and placement and routing of our proposed algorithm. The y-Axis gives the relative delay difference between the critical nets. The x-Axis gives the size of the self timed ring that was placed and routed.

are within 0.3 while the median is at 0.119. This implementation meets our novel timing requirement, equal delay within a set of nets, designed for security primitives. We expect the implemented STR to oscillate with a duty cycle of 50% as required by the STR-TRNG design. Additionally, we expect the resulting STR-TRNG to deliver true random numbers.

The implementation achieved by our proposed algorithm outperforms even the manual routing effort. The average median is 0.066, only a little over half the median of the manual implementation. The gaps between the 25% and 75% quantiles is also more compact compared to the manual implementation.

The timing characteristics of the implementation achieved using our algorithm prove that our algorithm provides the desired results. It meets the timing requirement we define in Section 2.2.3. We still need to prove our requirement is appropriate. In section Section 4.6.3 we show that the requirement has the desired effect on the STR.

4.6.3 Clock behavior

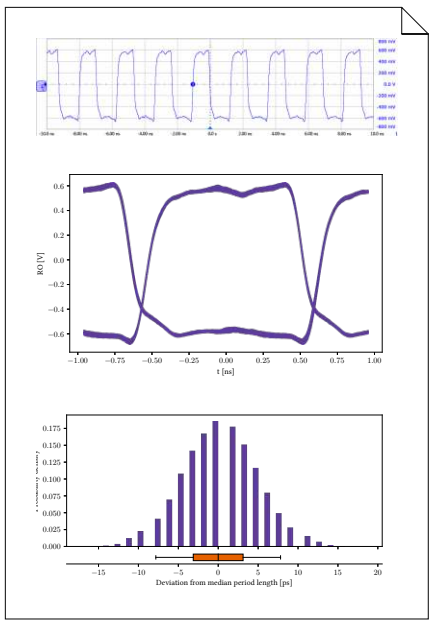
The quality of the random numbers produced by an STR-TRNG depends mostly on the quality of the two oscillating circuits in the design. We aim to confirm the findings of Cherkaoui et al. that the performance of the STR depends on its timing.[7] We want to use the entropy of the jitter of the RO, so the RO needs to have significant jitter. The RO then samples the STR (Figure 2.2a), so the STR must be much faster than the RO and very precise (Figure 2.2b). We therefore investigate the quality of the implemented clocks. We do so by capturing raw data using an oscilloscope. For each of the implemented designs, we:

1. Capture a screenshot of the oscilloscope,
2. List the measured frequency and period length,
3. Give an eye diagram (Infobox 4.2), and

[7] A. Cherkaoui et al. "Comparison of Self-Timed Ring and Inverter Ring Oscillators as entropy sources in FPGAs". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2012.

Infobox 4.1: Results layout

We perform experiments on different configurations of the STR-TRNG. We dedicate one page to each of the clocks we measured (Pages 63 to 68). The layout of these pages is explained here.



First we give a screenshot of the oscilloscope. It gives a first insight if the measured signal is oscillating or not. Highly irregular clocks can be identified by the naked eye.

Next, we give an eye diagram (Infobox 4.2) of 10000 clock cycles. Two lines that stylize the shape of an eye indicate regular oscillation. The thickness of the lines at the zero crossings shows the maximal jitter observed during the measurement.

Lastly, we show a histogram and bar plot of 100000 period lengths observed over time. Here we can see the statistical distribution of the jitter of the observed signal. A regular clock is expected to have a peak at the median period length that decays evenly to both sides.

4. Plot the jitter as a histogram and boxplot.

We present the results on one page per measurement (Pages 63 to 68) as explained in Infobox 4.1.

First, we look at the results of the RO in Figure 4.5. The RO does not need any special attention when it comes to PnR. Due to the nature of the circuit, the timings of the RO nets merely affects its frequency, not its duty cycle or its jitter. Therefore, it is feasible to use the same implementation of RO throughout all experiments to make sure the RO PnR does not skew our results.

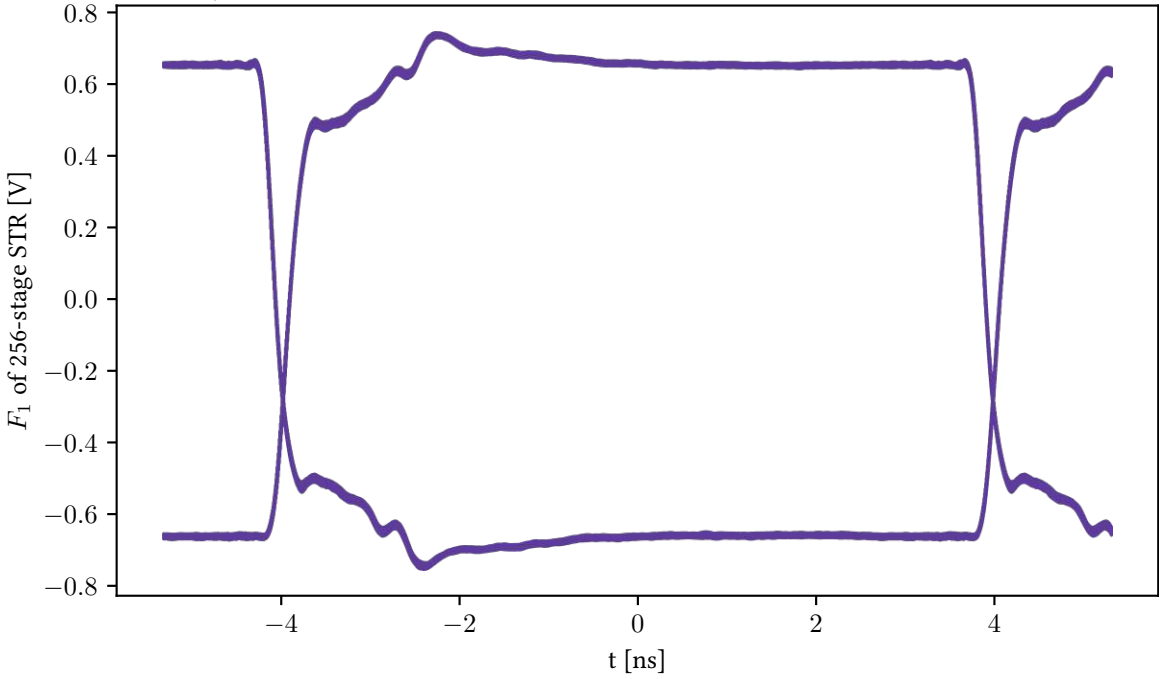
The oscilloscope trace in Figure 4.5a indicates that the RO produces a regular oscillation at $433MHz$ with an amplitude of $\approx 1.2V$. The eye-diagram in Figure 4.5b proves this observation. The lines are thin, meaning there is little deviation between periods. The oscillation is regular enough to produce a clean graph, but there is a noticeable amount of jitter present. We only care about the timing, so the differences in minimum and maximum voltage are acceptable for this purpose.

The histogram in Figure 4.5c shows an axially symmetric, bell-curve shaped probability density curve. It is not continuous on the x-Axis because of the sampling interval of the oscilloscope. The shape indicates the RO has a Gaussian jitter distribution. 90% of the time, the clock edge occurs within a $16ps$ time window. This is ideal for our purposes.

For the STR we use two different sizes: 64 and 256. For the 64-stage STR we can give results for a Vivado implementation (Figure 4.6), for a manual implementation (Figure 4.7), and for the implementation by our proposed algorithm (Figure 4.8). For the 256-stage STR we give results for a Vivado implementation (Figure 4.9) and for the implementation by our proposed algorithm (Figure 4.10). We do not implement the 256-stage STR manually due to its large size.

Infobox 4.2: Eye diagrams

We use eye diagrams to visualize qualitative properties of the clock signals produced by our oscillating circuits. We capture 10000 clock cycles and split the cycles at the zero-crossings into their positive and negative half-cycles. Then we determine the middle $t_{m,i}$ between the zero-crossings for all collected half-cycles and shift the half-cycles so that $t_{m,i} = 0$. Lastly we continue the half-cycles in both directions of the x-Axis to fill the plot.



The plots give information about the regularity of the oscillation. The overlay of the cycles gives a thin line if all the cycles are similar. If there are differences between the cycles, the line grows thicker. This is especially interesting at the zero-crossings. Thin lines at the zero-crossings indicate that all clock cycles have the same length. The thicker the lines at the zero-crossings, the more jitter is present in the clock signal.

First, we have a look at the Vivado implementation of the 64-stage STR in Figure 4.6. A quick look at the oscilloscope (Figure 4.6a) already shows that the STR does not deliver a regular clock signal. The individual cycles are of different duration and the duty cycle is also not constant. The oscilloscope screenshot alone is enough to disqualify the produced signal as a qualitative clock signal by any means of the definition.

This is also reflected in the eye diagram in Figure 4.6b. The clock signal is not regular enough to produce a decent eye. The jitter-edges are almost as long as the peak times.

In Figure 4.6c we can see that there is one dominant period length, but are plenty of periods that are shorter than that. 90% of the periods lie within a $1.6ns$ jitter window.

Next, we see how the behavior changes when implementing the design manually (Figure 4.7). The oscilloscope screenshot in Figure 4.7a shows the quality of the oscillation improves dramatically. We measure the frequency of a single stage of the STR at $63.5MHz$. We need to multiply the frequency by the size of the STR to determine the frequency of the STR as a whole (Section 4.5.2). This results in a frequency of $4.064GHz$.

The oscillation is good enough that imperfections are not perceivable by the naked eye anymore. In this case, we actually need the eye-diagram and the jitter-analysis to investigate further.

The eye-diagram in Figure 4.7b shows steep, thin edges. This indicates there is little jitter, which is confirmed by the jitter analysis in Figure 4.7c.

The last implementation of the STR of size 64 is the one implemented by our proposed algorithm (Figure 4.8). We can see that the oscillation (Figure 4.8a) is very similar to the one of the manual implementation. The measured single-stage frequency of the STR is $63.5MHz$, again leading to a frequency of $4.064GHz$ for the STR as a whole.

The eye-diagram in Figure 4.8b and the jitter-analysis in Figure 4.8c also mirror the results of the manual implementation.

The oscillation of the manual implementation in contrast to the Vivado implementation proves that our proposed constraint has the desired effect on the STR. Since the implementation by our proposed algorithm produces an oscillation that is just as good as the manual implementation we conclude that our algorithm is able to respect our proposed constraint.

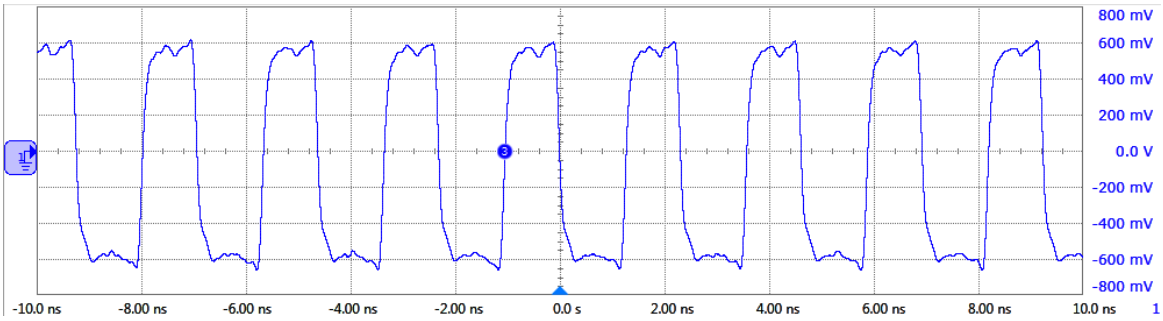
Finally, we want to discuss the performance of the 256-stage STR. We choose this design in hindsight to the results of the stochastic tests in Table 4.3, that seem to indicate that the Vivado implementation of the 256-stage design produces better results compared to the other sizes.

For this large design we do not have a manual implementation available. We consider the implementation of such a large design to be out-of-scope for this work. This means we can only compare the Vivado implementation (Figure 4.9) to the implementation by our proposed algorithm (Figure 4.10).

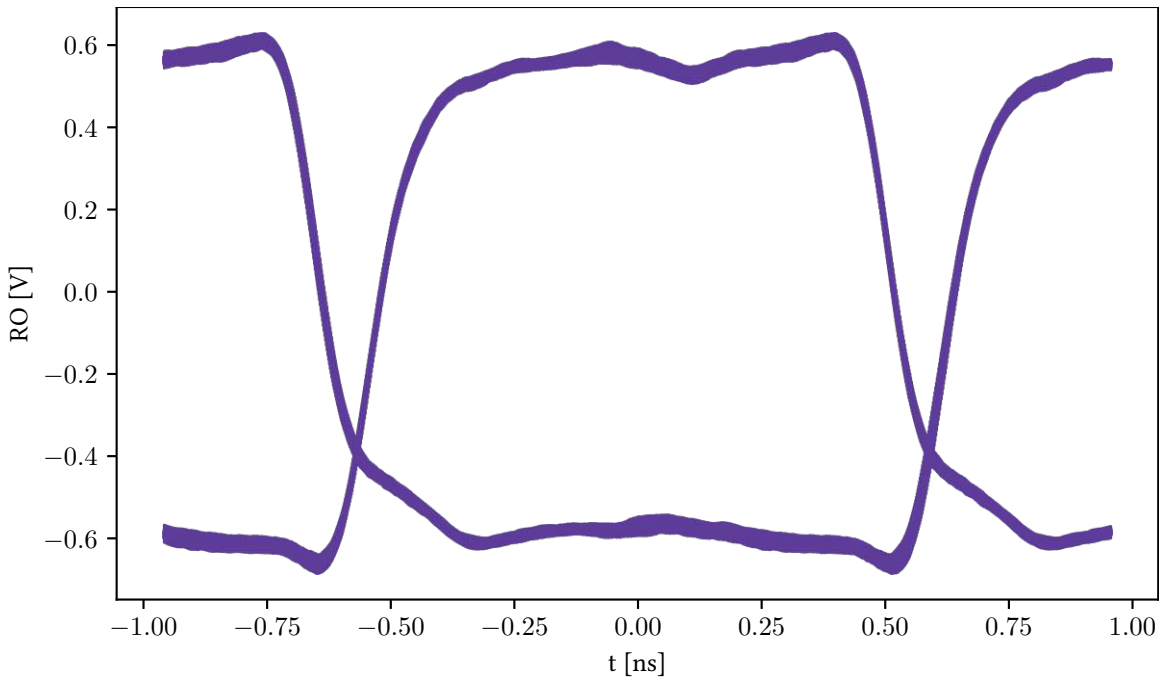
First we have a look at the oscilloscope screenshot for the Vivado implementation in Figure 4.9a. As was the case for the smaller design in Figure 4.6 we can immediately tell that the oscillation did not improve. It looks like every fifth period, either the high semi-cycle or the low semi-cycles are missing. This behavior of the STR is known as burst-mode as described by Cherkaoui et al..[8] It is a common symptom of badly implemented STRs. It manifests in two distinct regions in which period-lengths come to lie in Figure 4.9c. While over 75 (upper quartile) of the periods have similar length, the rest of the periods are much longer than that.

The implementation done by our proposed algorithm on the other hand is very similar to the one of the smaller STR design. We can again observe a very good oscillation in Figure 4.10a. We measure the frequency at a little less compared to the smaller design, at $62.8MHz$. Once again we have to multiply by the STR size, resulting in the frequency of the whole STR of $16.08GHz$. The eye-diagram (Figure 4.10b) and the jitter-analysis (Figure 4.10c) are almost identical to the respective figures (Figures 4.10b and 4.10c) of the 64-stage STR.

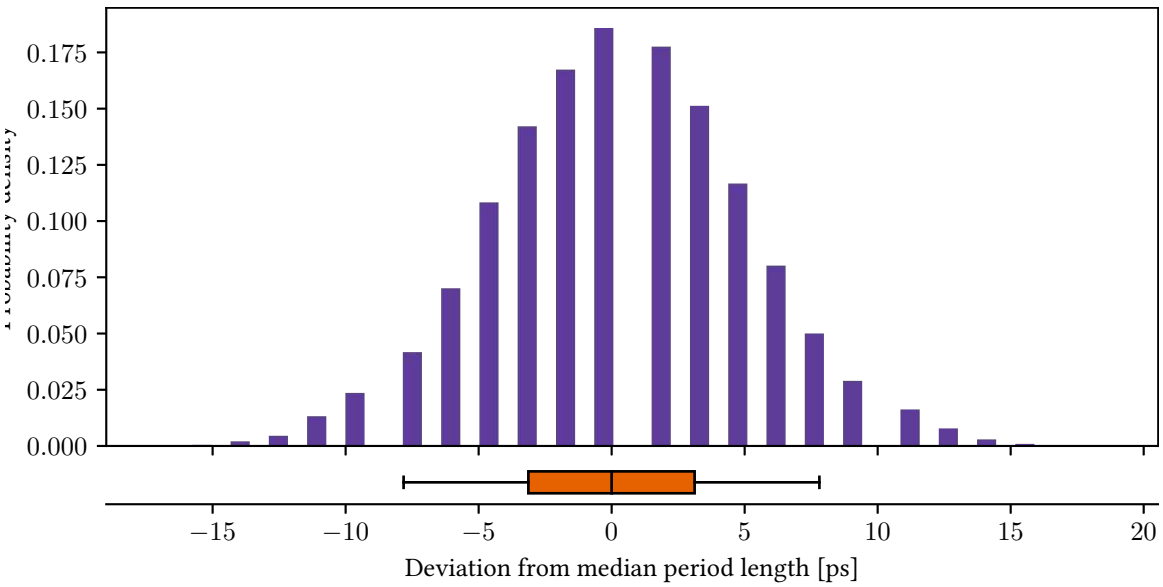
[8] Cherkaoui et al., "Comparison of Self-Timed Ring and Inverter Ring Oscillators as entropy sources in FPGAs".



(a) Oscillation captured by the oscilloscope.

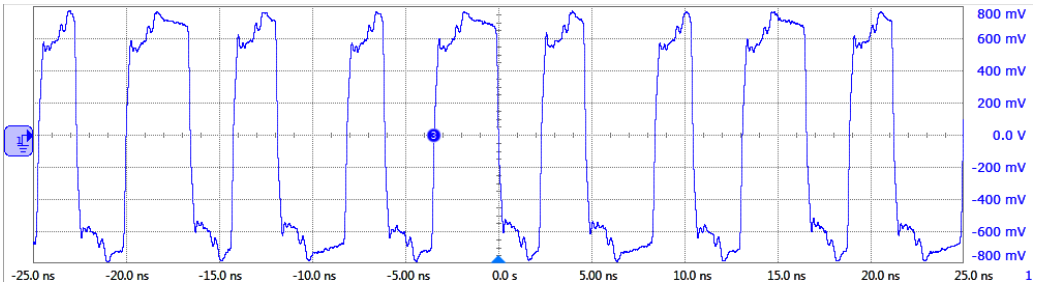


(b) Eye-diagram over 10000 clock cycles.

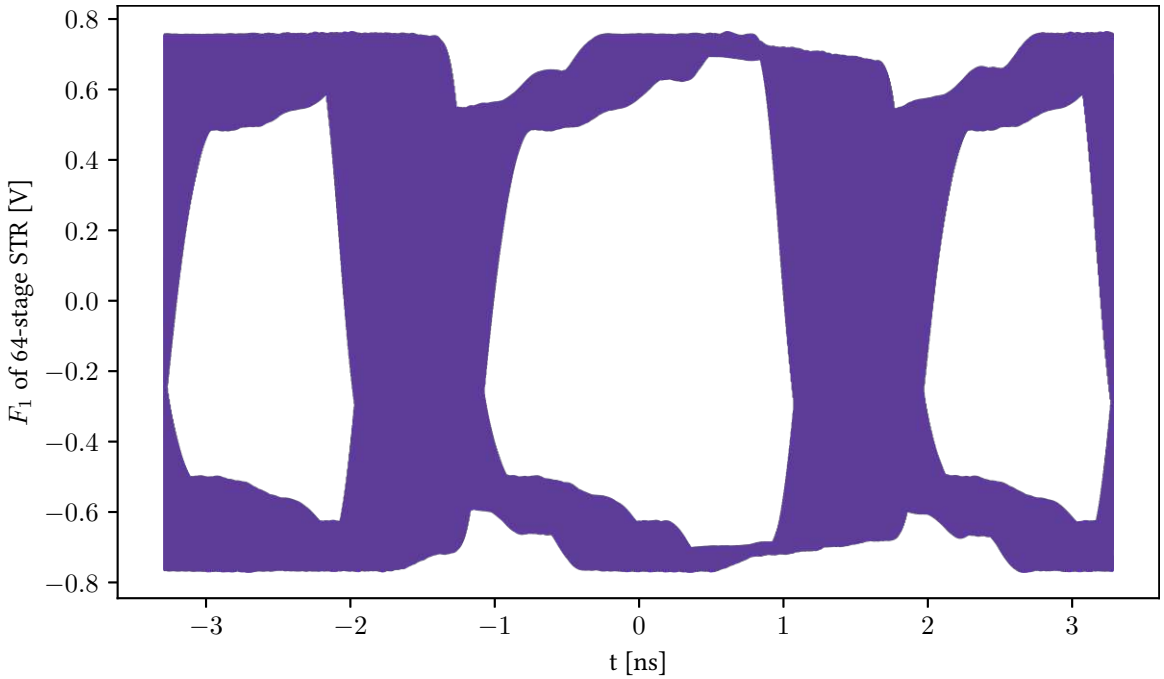


(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

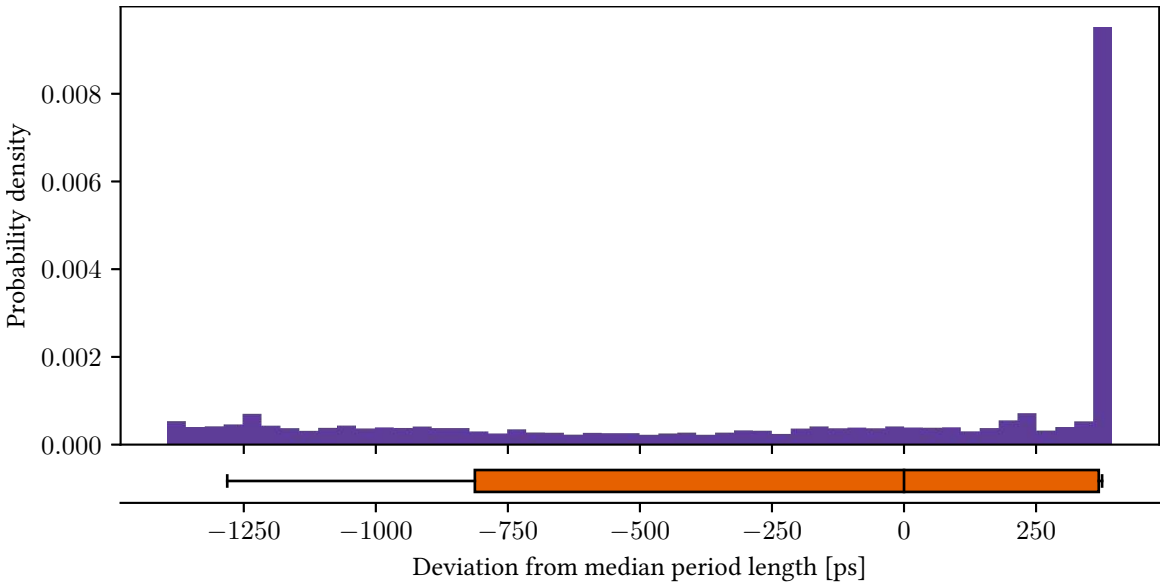
Figure 4.5: Analysis of the implemented ring oscillator consisting of 53 inverters used to sample the STR in our STR-TRNG use-case design.



(a) Oscillation captured by the oscilloscope.

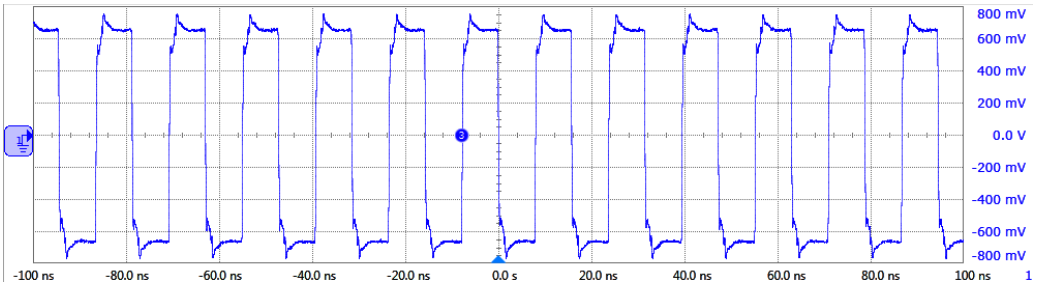


(b) Eye-diagram over 10000 clock cycles.

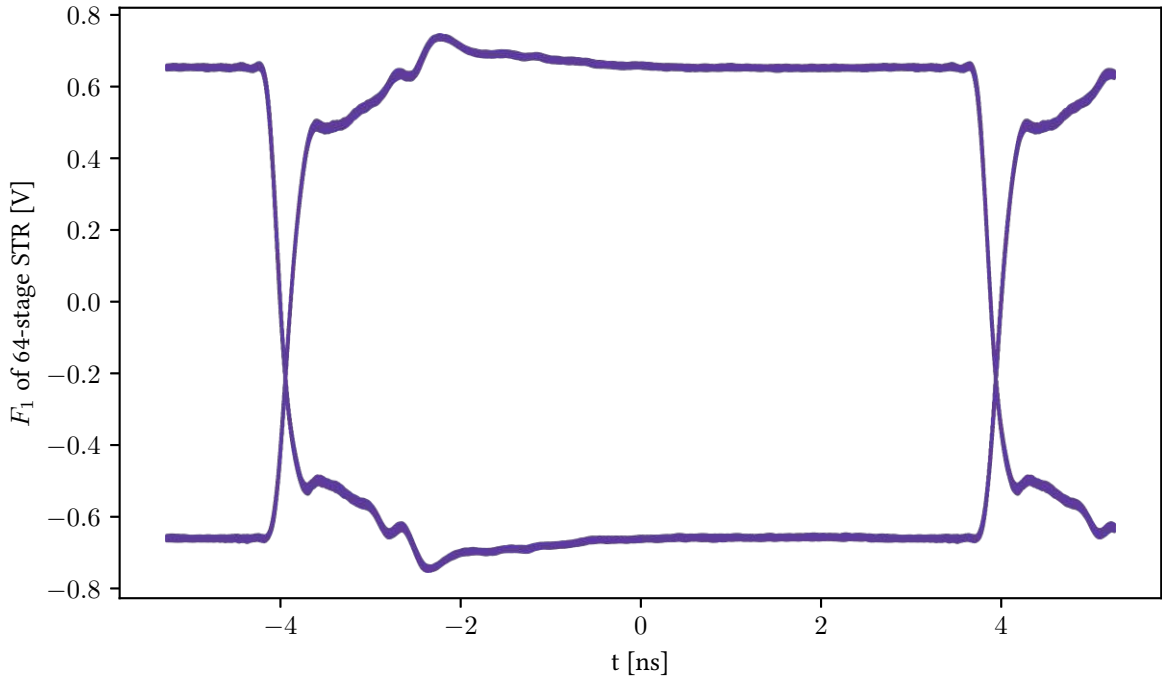


(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

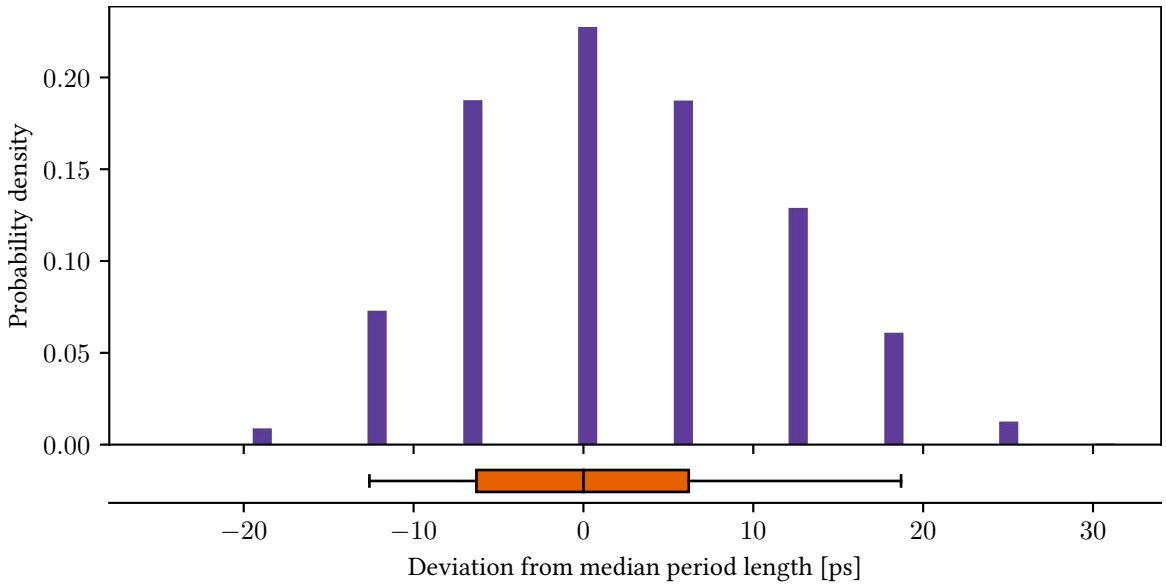
Figure 4.6: Analysis of a 64-stage self-timed-ring with one missing token implemented by Vivado (without constraints).



(a) Oscillation captured by the oscilloscope.

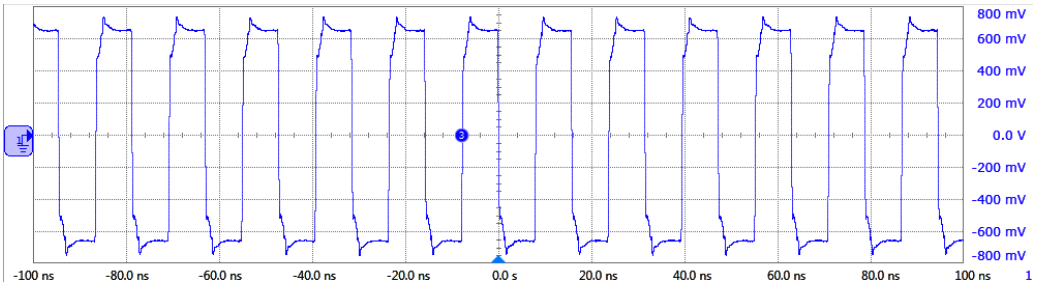


(b) Eye-diagram over 10000 clock cycles.

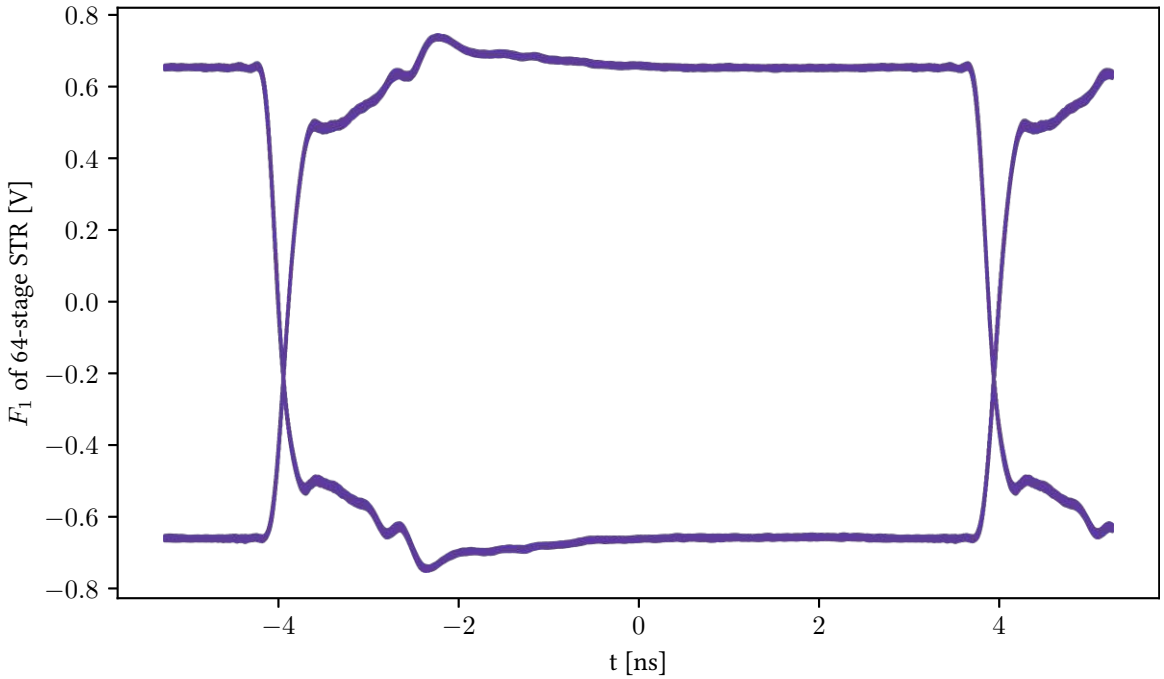


(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

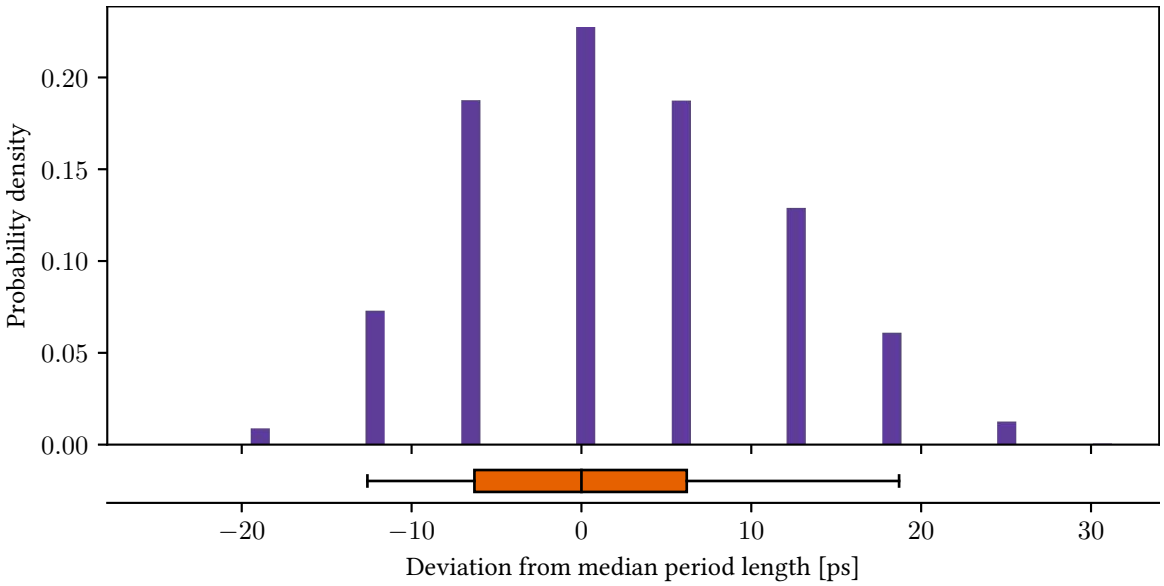
Figure 4.7: Analysis of a 64-stage self-timed-ring with one missing token manually implemented by a skilled engineer.



(a) Oscillation captured by the oscilloscope.

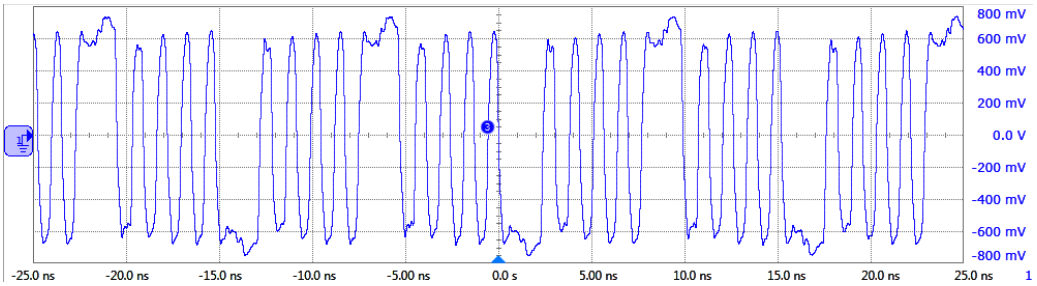


(b) Eye-diagram over 10000 clock cycles.

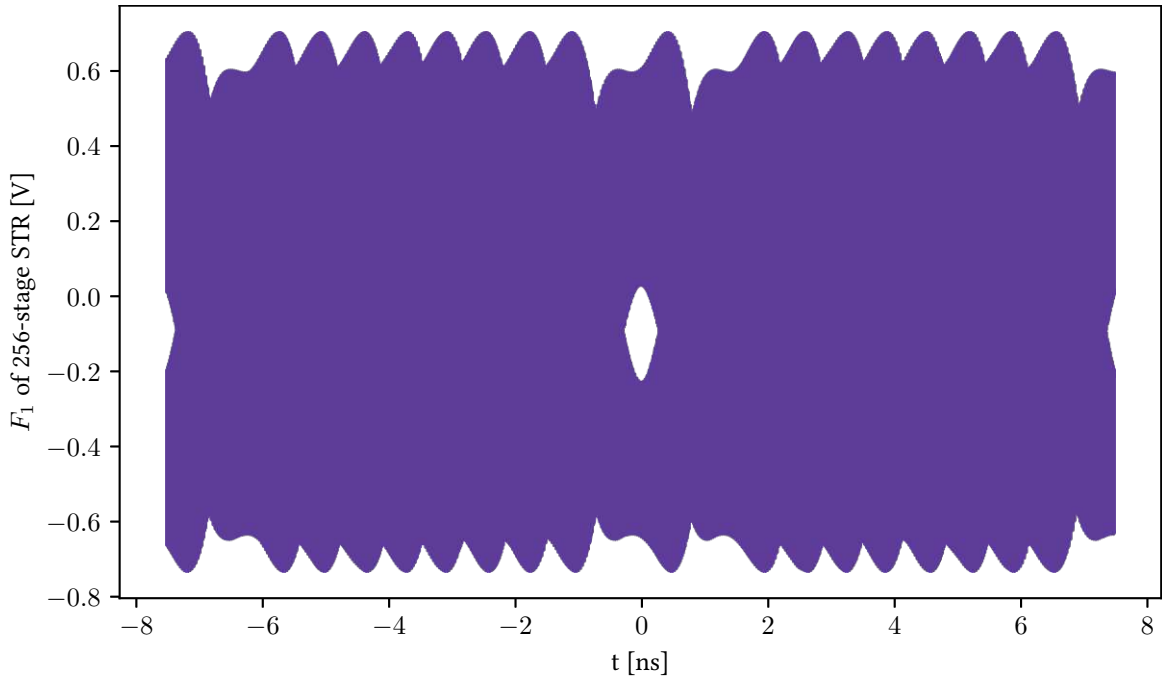


(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

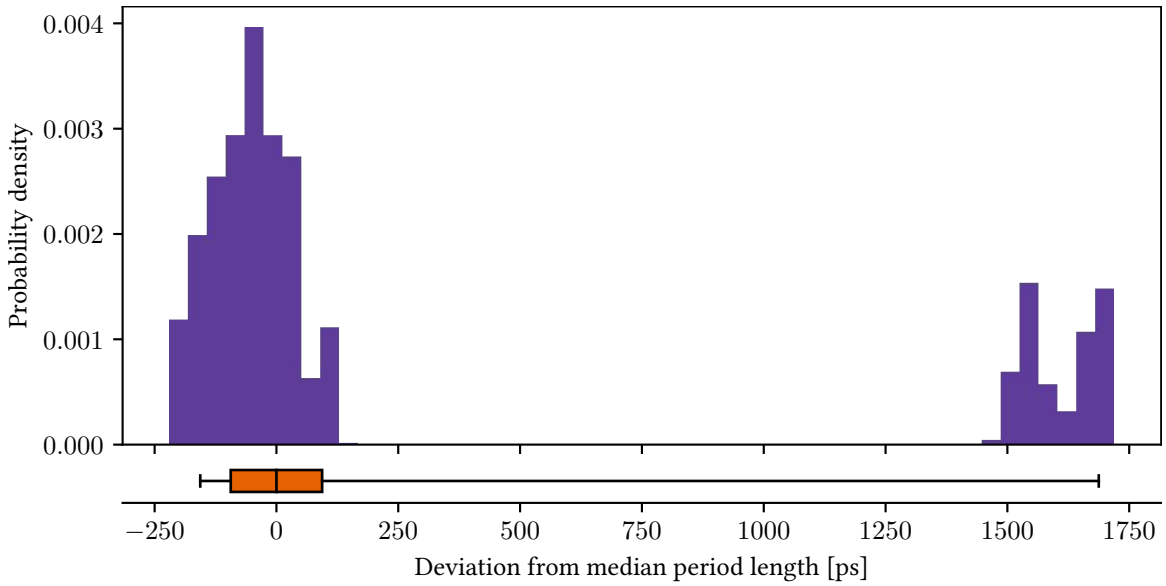
Figure 4.8: Analysis of a 64-stage self-timed-ring with one missing token implemented by our proposed algorithm.



(a) Oscillation captured by the oscilloscope.

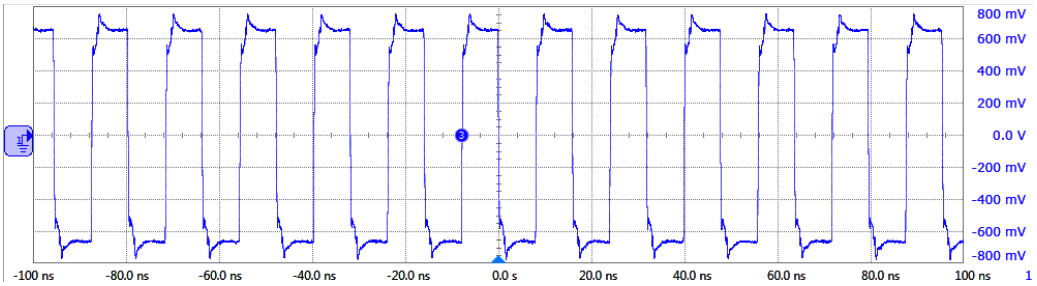


(b) Eye-diagram over 10000 clock cycles.

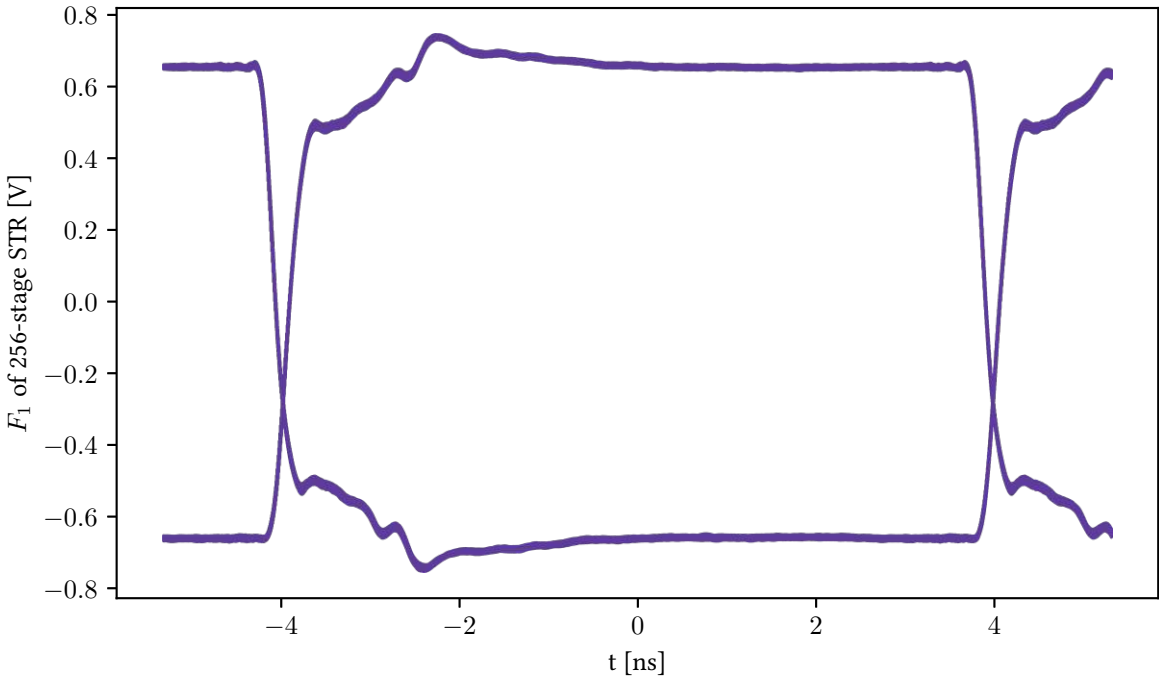


(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

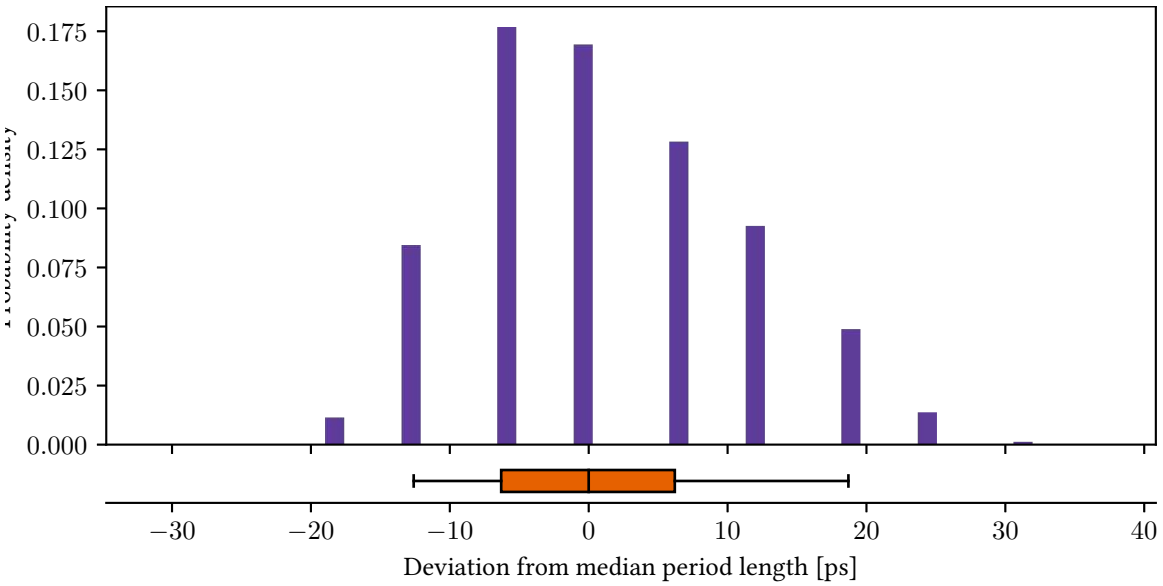
Figure 4.9: Analysis of a 256-stage self-timed-ring with 7 missing tokens implemented by Vivado (without constraints).



(a) Oscillation captured by the oscilloscope.



(b) Eye-diagram over 10000 clock cycles.



(c) Visualization of the jitter. We capture 100000 clock cycles and plot the deviation from the median period length as a histogram and a bar plot.

Figure 4.10: Analysis of a 256-stage self-timed-ring with 7 missing tokens implemented by our proposed algorithm.

4.6.4 Stochastic tests

Section 4.6.2 proves our algorithm meets our requirement. Section 4.6.3 shows our requirement has the desired effect on the behavior of the STR. Lastly, in this section, we show that our use-case example, the STR-TRNG (Section 2.3) performs as expected when implemented using our proposed algorithm. We do that by evaluating the final output of the design, the generated numbers, using statistical analysis (Section 2.3.3).

We give the results for the two configurations used in Section 4.6.3, using an STR with 64 (Table 4.2) and 256 stages (Table 4.3) here. More results can be found in the Appendix (Tables A.1 to A.3).

We perform the following tests on the generated binary files of random data:

- Check for Bias in Python (probability of any bit being '1'),
- NIST 800-22 tests,[9] and
- *Rngtest* to check compliance with *Federal Information Processing Standard (FIPS)* 140-2

We present these tests in detail in Section 2.3.3. Unfortunately, the *dieharder* tests cannot be performed due to the small amount of random data available offline (Section 4.5.1). The dieharder tests require between $10MB$ and $100MB$ of data, but we can only provide $4MB$.

A bias of 50% is a necessary, but not sufficient requirement on RNGs. We can see in the results that the other stochastic tests will not pass if the RNG is biased ($bias \neq 50\%$). Furthermore, we found the *NIST 800-22* test suite to be much more restrictive compared to *rngtest*. In some experiments we see many cases that pass *rngtest*, but not *NIST 800-22*. We do not see any cases where this was the other way around. These findings are in line with the findings of Hurley-Smith, Patsakis, and Hernandez-Castro.[10] These observations allow us to derive a qualitative hierarchy of stochastic tests, with *Bias* being the easiest test to pass and *NIST 800-22* the hardest.

Looking at Table 4.2 we can tell that the default Vivado implementation does not pass any of the tests. The bias lies below 40%. The strongly biased numbers do not pass any of the *rngtest* runs nor the *NIST 800-22* test suite. We notice from the other configurations that the bias improves with STR-size, but the other tests fail for all configurations but one. For the largest configuration (STR-size $= 256$ with 7 missing tokens, results can be seen in Table 4.3) we find that the bias is good enough to pass 99.9% of the *rngtest* runs and 65% of the *NIST 800-22* tests. This is close to acceptable performance, but looking at the neighboring columns in Table 4.3 indicates that this was a lucky coincidence.

The results for the manual implementation on the other hand look more promising. The bias is constant at 50.27%. The implementation passes over 99.8% of the *rngtest* runs. Unfortunately, the bias is not good enough to pass the *NIST 800-22 Frequency test*. The *Frequency test* is an extended bias test. Instead of investigating the probability of a logic '1' across the whole datastream, the datastream is split into chunks and the bias of the individual chunks is measured. If the *Frequency test* fails, all other tests are skipped. This is the case for the manual implementation.

Lastly we look at the results achieved by the implementation with our proposed algorithm. The bias is very close to 50% ($\pm 0.01\%$ across all implementations). The individual configurations pass between 99.89% and 99.96% of *rngtest*

[9] Rukhin et al., *A statistical test suite for random and pseudorandom number generators for cryptographic applications*.

[10] D. Hurley-Smith, C. Patsakis, and J. Hernandez-Castro. "On the unbearable lightness of FIPS 140-2 randomness tests". In: *IEEE Transactions on Information Forensics and Security* (2020).

runs. Most of the *NIST 800-22* tests pass with flying colors (10/10 runs pass), except for a few cases (still at least 7/10 runs pass).

STR Size	64											
Strategy	Vivado				Manual				Algorithm			
Missing Tokens	1	3	5	7	1	3	5	7	1	3	5	7
Bias												
Probability of logic '1'	39.50%	32.51%	36.73%	36.43%	50.27%	50.27%	50.27%	50.27%	50.00%	50.00%	50.00%	50.00%
FIPS 140-2												
rngtest	0.00%	0.00%	0.00%	0.00%	99.80%	99.90%	99.83%	99.80%	99.92%	99.93%	99.90%	99.92%
NIST 800-22												
Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Block Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	9/10	10/10	10/10
Runs test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Longest Run Of Ones test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Matrix Rank test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Discrete Fourier Transform test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Non-Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	9/10	10/10	10/10
Universal test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	9/10	10/10	10/10	10/10
Linear Complexity test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Serial test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Approximate Entropy test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Cumulative Sums test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Random Excursions test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	8/10	8/10	9/10	10/10
randomExcursionsVariant test	0/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10	8/10	8/10	9/10	10/10

Table 4.2: Results of the stochastic tests on a Self-Timed-Ring True-Random-Number-Generator of Sizes 64. We show results for the default vivado placement and routing, manual placement and routing, and our automated placement and routing. All configurations are tested ten times. Bias is the probability of each bit being '1'. The *rngtest* produces 1677 pass or fails for each run. The remaining test cases produce a single pass or fail for each run. The number given in the table is the average probability of the test passing. A value of 100% means the configuration passed the test in all ten iterations.

STR Size	256											
Strategy	Vivado						Algorithm					
Missing Tokens	1	3	5	7	9	11	1	3	5	7	9	11
Bias												
Probability of logic '1'	47.33%	50.49%	48.71%	50.00%	49.79%	49.32%	50.00%	50.00%	50.01%	50.00%	49.99%	50.00%
FIPS 140-2												
rngtest	0.00%	9.73%	56.08%	99.90%	99.66%	75.54%	99.92%	99.92%	99.92%	99.94%	99.92%	99.90%
NIST 800-22												
Frequency test	0/10	0/10	0/10	10/10	0/10	1/10	10/10	10/10	9/10	10/10	9/10	10/10
Block Frequency test	0/10	0/10	0/10	6/10	0/10	1/10	10/10	10/10	9/10	10/10	9/10	10/10
Runs test	0/10	0/10	0/10	1/10	0/10	0/10	9/10	10/10	9/10	10/10	9/10	10/10
Longest Run Of Ones test	0/10	0/10	0/10	10/10	0/10	0/10	10/10	10/10	9/10	10/10	9/10	9/10
Matrix Rank test	0/10	0/10	0/10	10/10	0/10	1/10	10/10	10/10	9/10	10/10	9/10	10/10
Discrete Fourier Transform test	0/10	0/10	0/10	7/10	0/10	0/10	10/10	10/10	8/10	10/10	9/10	10/10
Non-Overlapping Template Matchings test	0/10	0/10	0/10	10/10	0/10	1/10	10/10	10/10	9/10	10/10	9/10	10/10
Overlapping Template Matchings test	0/10	0/10	0/10	10/10	0/10	0/10	10/10	9/10	9/10	10/10	8/10	10/10
Universal test	0/10	0/10	0/10	8/10	0/10	1/10	9/10	10/10	9/10	10/10	9/10	9/10
Linear Complexity test	0/10	0/10	0/10	10/10	0/10	1/10	10/10	10/10	9/10	10/10	7/10	10/10
Serial test	0/10	0/10	0/10	1/10	0/10	0/10	10/10	10/10	8/10	9/10	9/10	8/10
Approximate Entropy test	0/10	0/10	0/10	1/10	0/10	0/10	10/10	10/10	9/10	10/10	8/10	10/10
Cumulative Sums test	0/10	0/10	0/10	10/10	0/10	1/10	10/10	10/10	9/10	10/10	9/10	10/10
Random Excursions test	0/10	0/10	0/10	6/10	0/10	1/10	9/10	8/10	9/10	8/10	8/10	7/10
randomExcursionsVariant test	0/10	0/10	0/10	8/10	0/10	1/10	9/10	9/10	7/10	9/10	8/10	7/10

Table 4.3: Results of the stochastic tests on a Self-Timed-Ring True-Random-Number-Generator of Sizes 256. We show results for the default vivado placement and routing, manual placement and routing, and our automated placement and routing. All configurations are tested ten times. Bias is the probability of each bit being '1'. The *rngtest* produces 1677 pass or fails for each run. The remaining test cases produce a single pass or fail for each run. The number given in the table is the average probability of the test passing. A value of 100% means the configuration passed the test in all ten iterations.

Chapter 5

Discussion

In this chapter we discuss our main findings. We show that a large selection of security primitives shares a common property in Chapter 2. We find that they mostly rely on race conditions and therefore their performance depends on their implementation, more precisely on the timing properties of certain nets. Critical nets must be routed to have equal delay to provoke a race condition. We define a constraint to annotate critical nets and an algorithm that automates the implementation process according to the found requirements in Chapter 3. In Chapter 4 we devise and execute experiments to test our proposed algorithm in conjunction with our proposed constraint. The conducted experiments show that our algorithm finds an implementation that satisfies our constraint down to an error margin of 30%. It does so while reducing the time a developer has to spend on the implementation from 328 hours to about 70 minutes.

To demonstrate the value of our work, we show that our automated flow is much faster than the state-of-the-art flow in Section 5.1 and that it satisfies our novel timing constraint in Section 5.2. In Section 5.3 we find that the timing requirements have the expected effect on the clocks used in our use-case design which in turn leads to high-quality random numbers as discussed in Section 5.4. This subdivision reflects the structure of Section 4.6.

5.1 Design flow automation

The main contribution of this work is the automation of the design flow for security primitives on *field-programmable gate arrays (FPGAs).* We compare implementation times for a *self timed ring (STR)-based* true random number generator (TRNG) *(STR-TRNG)* by a skilled engineer and our proposed algorithm in Table 4.1.

The state-of-the-art design flow requires an expert to perform the *placement and routing (PnR)* step. It takes the writer about two months (328 hours) to place and route the STR-TRNG appropriately. Unfortunately, we cannot measure the time that is needed to acquire the necessary skills to perform the PnR.

We get around the need for manual PnR by defining constraints in *hardware description language (HDL)* and devising an algorithm to respect these constraints. Our proposed algorithm manages to bring the implementation time down to 5 hours and 20 minutes. This time includes one hour of studying the requirements and defining them in HDL. The remaining 4 hours and 20 minutes are spent by the algorithm and not the designer. Assuming human time is more valuable and expensive than computing time, this is a decrease in time of factor 328 (Comparison of human time spent

during the state-of-the-art design flow versus during our proposed design flow, i.e. the orange cells in Table 4.1). Even if computing time is valued as highly as human time, we still note a *decrease in time of factor* 61 (Comparison of human *and* machine time spent during the state-of-the-art design flow versus during our proposed design flow, i.e. all cells in Table 4.1). In contrast to the manual implementation, our automated approach does not require any PnR skills. It is therefore usable by a wider range of designers and eases the access to security primitives for digital hardware designs.

To further improve the runtime of our algorithm, we propose two improvements:

1. *Find patterns in design and copy routes*

   This improvement requires two prerequisites:

   (a) The design needs to be repetitive (e.g., multiple instances of the same module) and

   (b) the FPGA needs to be homogenous.

   If both prerequisites are met, we can look for patterns in the design (e.g., for multiple instances of a module, leveraging subcircuit matchers (present in most synthesis tools for technology-mapping, e.g., the Ullmann Subgraph Isomorphism Algorithm[1] in Yosys[2]), or by employing a pattern-based search algorithm[3]). If matches are found, we can route one of them and try to apply the found routes to the identical matches. Assuming the FPGA is indeed homogenous, the routes may be applicable to most instances of the found pattern or module.

2. *Determine relative threshold from qualitative requirements on the random numbers*

   In the current implementation, the user defines a relative threshold that defines how much the delays of nets within a delay group are allowed to deviate from each other (Section 3.2). Setting the relative threshold too tight ($threshold \approx 1$) results in the algorithm taking a long time to find suitable routes or not finding suitable routes at all. Setting the relative threshold too loose ($0 \leq threshold \ll 1$) results in lower entropy, less speed or the TRNG nor operating at all.

   Currently, it is up to the user to find an appropriate balance. Usability could be improved by devising formulas to deduce the threshold from qualitative requirements (speed and entropy) on the random numbers.

In terms of implementation quality our algorithm is limited by the available hardware resources. For the experiments we configured the algorithm to accept differences in delays of up to 30%. We found that the algorithm is not able to achieve lower error margins with the available routing resources. We hit the same lower bound during the manual implementation, confirming this limit originates from a lack of resources rather than from a design flaw in our algorithm.

In addition to the improvement in implementation time, automation brings two more benefits: the implementation is repeatable and less error-prone.

[1] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: (Jan. 1, 1976).
[2] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/. (Visited on 02/02/2022).
[3] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. PhD thesis. Gusshausstrasse 27–29/384, 1040 Wien: Vienna University of ..., 2019.

5.2 Static timing analysis

In Figure 4.4 we compare the static timing properties of the designs implemented for the experiments. The relative delay differences for the Vivado implementation are very high, while the delay differences of the manual implementation and the implementation done by our proposed algorithm are below the 30% threshold we defined. This means that both the engineer that performed the manual implementation and *our algorithm are capable of respecting our novel timing constraint.*

A surprising fact is that the performance of the algorithm does not deteriorate with the size of the STR. Even though the larger STR means there is more competition for resources between the individual stages, the algorithm still finds an implementation that is similar in timing to the smaller STR implementations.

As discussed in Section 5.1, the 30% error margin is the best we can do with the resources on the given target platform. We have to measure the produced clocks in hardware to determine whether that error margin is low enough for the STR to oscillate fast and precisely. We do so in Section 5.3.

5.3 Clock behavior

The oscilloscope measurements in Section 4.6.3 clearly show that the manual implementation and the implementations generated by our algorithm produce regular clocks while the Vivado implementation does not. This implies that the performance of the STR depends on its implementation and *our proposed constraint is suitable to find performant implementations.*

Unfortunately, we can only measure the clock output of single STR stages F_i. This is due to two limitations in our test setup:

1. The accumulated final STR clock clk is $clk = F_i * STR\_SIZE$. Even for smaller STRs ($STR\_SIZE < 32$), that is too fast to be reliably measured by the measuring tools available to us.

2. We only have a single pair of *SubMiniature version A (SMA)* clock outputs available on our target hardware. The two outputs are designed to be used in a *Low voltage differential signaling (LVDS)* setup, and can therefore only transmit one clock at a time.

We illustrate how the individual clocks F_i are accumulated into the final clock clk in Figure 2.9. Measuring only a single F_i at a time means we can only assess the quality of a single stage. While this gives a good indication of how the STR behaves, it hides the fact that the accumulation of individual clocks is also very reliant on good placement and routing. E.g., if the net that connects F_1 to the accumulating *exclusive-or (XOR)* has higher delay than the net that connects F_2 to the XOR, F_1 would shift to the right in Figure 2.9, deforming clk. As we have no faster measurement equipment and no development boards with more SMA output connectors available we must rely on the stochastic tests to assess the performance of the STR.

It would be interesting to repeat the experiments on a development board with more SMA output connectors and an oscilloscope with a higher sampling rate and many SMA inputs. An oscilloscope with a higher sampling rate would allow us to observe the accumulated final clock clk directly. Given more SMA outputs on the board and inputs on the oscilloscope, we could observe the output of multiple stages F_i at the same time and even measure the phase shift

between them. For a reliable measurement of the phase shift, the delay of the nets connecting the output of the stages F_i to their corresponding SMA output would need to be equal. This would be another use-case for our proposed constraint and algorithm.

5.4 Stochastic tests

Lastly, we discuss the high-level output of our use-case design: the random numbers produced by the STR-TRNG. We calculate the bias, and perform *rngtest* and *NIST 800-22* tests. We give results in Tables 4.2, 4.3, A.1 and A.3.

We can immediately tell that the Vivado implementation does not produce acceptable random numbers. The numbers are highly biased and most of the stochastic tests fail. There is one configuration that seems to do well enough, the STR-TRNG using a 256-stage STR configured with 7 tokens missing, but we show in Figure 4.9a that the STR was running in burst mode for that case. The burst mode is described in the original publication of the STR.[4] It occurs when the routing is not satisfactory. When looking at a very short capture, the burst mode may look like a regular clock, but when looking at a longer capture, one can see that it constantly switches between an oscillation and a constant output. This is clearly visible in Figure 4.9a and can further be analyzed in Figures 4.9b and 4.9c. This means the result is very likely not repeatable.

The manual implementation on the other hand looks promising. As discussed in Table 4.1, implementing such a design by hand is very time-consuming. We therefore only implemented one configuration by hand (STR of size 64). Implementing all configurations manually would improve the significance of our results, but delay this work by over half a year (consider that the larger designs also take longer to implement).

The manual implementation is less biased and mostly passes the *rngtest*. However, the small bias that remains is large enough to cause the *NIST 800-22 Frequency test* to fail. As that test is a precondition to all other tests, the manual implementation fails all *NIST 800-22* tests. The observation that a design may pass several *rngtests* but fail *NIST 800-22* tests is in line with the observations of Hurley-Smith, Patsakis, and Hernandez-Castro.[5] They state that the *rngtests* are not hard enough and cannot detect an adversarial bias in a bitstream. As we identified significant bias in our results for the manual implementation, we suspect that to be the reason that the *rngtests* pass but the *NIST 800-22* tests fail.

The automated implementation is even less biased. The worst bias we could measure was $50 \pm 0.01\%$. That is enough to pass the *NIST 800-22 Frequency test* and all other *NIST 800-22* tests most of the time. The results of the stochastic tests for the automated implementation proves that the *STR-TRNG works as advertised.*

Comparing the results of the automated implementation and the manual implementation, we see that they have almost identical performance in terms of clock behavior, but differ in terms of the stochastic tests. We assume this is due to the flaw of the measurement setup discussed in Section 5.3. The oscilloscope-measurements cannot reflect the timing properties of the XOR tree at the output of the STR, as we have to measure the clock before the XOR tree.

All tests were performed with a sample size of about $4MB$ due to the limited buffering capabilities of our target hardware. This could be improved in two ways:

[4] O. Elissati et al. "Self-Timed Rings: A Promising Solution for Generating High-Speed High-Resolution Low-Phase Noise Clocks". In: *VLSI-SoC: Forward-Looking Trends in IC and Systems Design.* Ed. by J. L. Ayala, D. Atienza Alonso, and R. Reis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[5] D. Hurley-Smith, C. Patsakis, and J. Hernandez-Castro. "On the unbearable lightness of FIPS 140-2 randomness tests". In: *IEEE Transactions on Information Forensics and Security* (2020).

- Add external memory as a buffer or
- Implement an interface that is fast enough to transmit the random numbers in real time (e.g., *Peripheral Component Interconnect Express (PCIe)*).

With a sample size of $10 - 100MB$ we could add the *dieharder* test suite to our selection of stochastic tests to further support our claims.

Chapter 6

Conclusion

This research identifies common general properties of *physical unclonable functions (PUFs)* and *true random number generators (TRNGs)* that make them hard to implement in hardware. We find that PUFs and TRNGs often depend on race conditions on certain nets. For the race conditions to occur, these nets need to have equal delay. A skilled engineer needs to place and route the design accordingly. We conclude that using the state-of-the-art design flow, to implement security primitives in digital hardware one needs a skilled engineer and a lot of labor.

In this work, we present an alternative flow that allows a person that completely lacks the skills required for manual *placement and routing (PnR)* to implement security primitives in digital hardware. We define a constraint *DELAY_GROUP* that allows to group nets into groups. We propose an algorithm that then places and routes a digital design on a *field-programmable gate array (FPGA)* so that nets that are in the same *DELAY_GROUP* have the same delay within an error margin of 30%. This results in a novel design flow for hardware security primitives. The manual effort of PnR is deprecated. We reduce the human labor required for the implementation by factor 328.

To demonstrate the usefulness of our proposed constraint and algorithm, we follow a chain of implications on a *self timed ring (STR)-based TRNG (STR-TRNG)* as a use-case example. We place and route the STR-TRNG by hand first and then repeat PnR using our proposed constraint and algorithm. We record the difference in the amount of labor that is required. The manual implementation takes about a month of skilled labor while the automated approach takes mere hours. In the future, this time could further be improved by leveraging the fact that oscillator circuits are often repetitive. This means there should be patterns in the design so that routing one occurrence and applying the same route to all other occurrences could work and reduce the implementation time even more.

Analyzing the physical implementation using *static timing analysis (STA)* we see both the manual implementation and the automated implementation meet our novel timing constraint of equal delay up to an error margin of 30%. In both approaches we see an error margin of 30% is necessary, there are simply not enough routing resources available on our target hardware to route the design more precisely.

We see the error margin is low enough when analyzing the behavior of the clocks the STR-TRNG produces using an oscilloscope. We see that both the manual implementation and the automated implementation oscillate appropriately,

while an implementation that is completely agnostic of any timing requirements fails to produce a usable clock. This proves the design requires special attention during PnR in the first place.

Lastly we analyze the produced random numbers using stochastic testing. We see that the performance of our selected use-case example, the STR-TRNG, depends on its internal timing properties. The manual implementation and our automated implementation, being aware of our novel timing requirement, perform well while the plain implementation without awareness of the special timing requirements fails even the most basic random tests.

In this work we were limited by the routing and buffering resources of our target hardware, the *input/output (I/O)* capabilities of our target hardware, and by our experimental setup. The target hardware did not allow us to keep the error margin below 30% as discussed above. Also, its single differential *SubMiniature version A (SMA)* output only allowed us to observe one of the internal clocks at a time. Due to the sample-rate of our oscilloscope, this clock can not be the final output of the STR, as that is too fast. Additionally, the target hardware limits the amount of random numbers we can buffer to $4MB$, which is not enough for one widely used test suite, namely *dieharder*. To further argue the usability of our work, one would need:

- An FPGA development board with more differential SMA outputs as well as a faster oscilloscope with enough differential SMA inputs to sample all the FPGA SMA outputs simultaneously, and
- A development board with either more memory to increase the buffer size or a faster communication interface to transfer the generated random numbers in real time, avoiding buffering at all.

This work molds the required skill to manually place and route hardware security primitives into an algorithm. It thereby reduces the manual effort and skill required to implement hardware security primitives making them more accessible to hardware designers.

Appendix A

Additional experimental results

Here we present experimental results for configurations not discussed in Section 4.6 and Chapter 5. They were omitted in Section 4.6 and Chapter 5 as they do not add additional information that affects the conclusion in any way. Nevertheless, they add value as they show that the results shown in Section 4.6 are not outliers.

STR Size	16				32							
Strategy	Vivado		Algorithm		Vivado				Algorithm			
Missing Tokens	1	3	1	3	1	3	5	7	1	3	5	7
Bias												
Probability of logic '1'	11.58%	22.56%	50.00%	50.00%	11.65%	12.56%	13.10%	15.15%	50.00%	50.00%	50.01%	50.00%
FIPS 140-2												
rngtest	0.00%	0.00%	99.89%	99.91%	0.00%	0.00%	0.00%	0.00%	99.89%	99.95%	99.96%	99.90%
NIST 800-22												
Frequency test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Block Frequency test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Runs test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Longest Run Of Ones test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	9/10	10/10
Matrix Rank test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Discrete Fourier Transform test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	9/10	9/10	9/10
Non-Overlapping Template Matchings test	0/10	0/10	10/10	9/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Overlapping Template Matchings test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Universal test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Linear Complexity test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Serial test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	9/10	10/10
Approximate Entropy test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Cumulative Sums test	0/10	0/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
Random Excursions test	0/10	0/10	9/10	10/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10
randomExcursionsVariant test	0/10	0/10	8/10	9/10	0/10	0/10	0/10	0/10	10/10	9/10	9/10	10/10

Table A.1: Results of the stochastic tests on a Self-Timed-Ring True-Random-Number-Generator of Sizes 16 and 32. We show results for the default Vivado placement and routing, manual placement and routing, and our automated placement and routing. All configurations are tested ten times. Bias is the probability of each bit being '1'. The *rngtest* produces 1677 pass or fails for each run. The remaining test cases produce a single pass or fail for each run. The number given in the table is the average probability of the test passing. A value of 100% means the configuration passed the test in all ten iterations.

STR Size	128											
Strategy	Vivado						Algorithm					
Missing Tokens	1	3	5	7	9	11	1	3	5	7	9	11
Bias												
Probability of logic '1'	59.70%	48.78%	51.42%	51.01%	49.41%	50.38%	50.00%	50.00%	50.00%	50.00%	50.00%	50.00%
FIPS 140-2												
rngtest	0.00%	18.16%	39.91%	76.39%	73.25%	89.79%	99.92%	99.93%	99.91%	99.92%	99.92%	99.92%
NIST 800-22												
Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Block Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	9/10	10/10
Runs test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Longest Run Of Ones test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	9/10	10/10
Matrix Rank test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Discrete Fourier Transform test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	9/10	10/10	9/10	10/10	10/10
Non-Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	9/10	8/10	10/10	9/10
Universal test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Linear Complexity test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	8/10	10/10	10/10
Serial test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Approximate Entropy test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Cumulative Sums test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	9/10	10/10	10/10
Random Excursions test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	8/10	8/10	9/10	8/10	8/10
randomExcursionsVariant test	0/10	0/10	0/10	0/10	0/10	0/10	9/10	8/10	9/10	9/10	7/10	7/10

Table A.2: Results of the stochastic tests on a Self-Timed-Ring True-Random-Number-Generator of Sizes 128. We show results for the default vivado placement and routing, manual placement and routing, and our automated placement and routing. All configurations are tested ten times. Bias is the probability of each bit being '1'. The *rngtest* produces 1677 pass or fails for each run. The remaining test cases produce a single pass or fail for each run. The number given in the table is the average probability of the test passing. A value of 100% means the configuration passed the test in all ten iterations.

STR Size	192											
Strategy	Vivado						Algorithm					
Missing Tokens	1	5	7	11	13	17	1	5	7	11	13	17
Bias												
Probability of logic '1'	46.03%	49.75%	49.72%	50.07%	50.21%	50.33%	50.00%	50.00%	50.00%	50.00%	50.00%	50.00%
FIPS 140-2												
rngtest	0.00%	99.35%	88.29%	99.90%	99.61%	99.60%	99.95%	99.92%	99.90%	99.96%	99.95%	99.93%
NIST 800-22												
Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Block Frequency test	0/10	0/10	0/10	0/10	0/10	0/10	9/10	8/10	10/10	10/10	10/10	10/10
Runs test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Longest Run Of Ones test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	9/10	10/10
Matrix Rank test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Discrete Fourier Transform test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Non-Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Overlapping Template Matchings test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	9/10	10/10	10/10	10/10	10/10
Universal test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Linear Complexity test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	9/10	10/10
Serial test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Approximate Entropy test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	9/10	10/10	9/10	10/10	10/10
Cumulative Sums test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	10/10	10/10	10/10	10/10
Random Excursions test	0/10	0/10	0/10	0/10	0/10	0/10	9/10	9/10	9/10	9/10	9/10	6/10
randomExcursionsVariant test	0/10	0/10	0/10	0/10	0/10	0/10	10/10	10/10	8/10	10/10	8/10	9/10

Table A.3: Results of the stochastic tests on a Self-Timed-Ring True-Random-Number-Generator of Sizes 192. We show results for the default vivado placement and routing, manual placement and routing, and our automated placement and routing. All configurations are tested ten times. Bias is the probability of each bit being '1'. The *rngtest* produces 1677 pass or fails for each run. The remaining test cases produce a single pass or fail for each run. The number given in the table is the average probability of the test passing. A value of 100% means the configuration passed the test in all ten iterations.

Bibliography

[1] L. Afflerbach. "Criteria for the assessment of random number generators". en. In: *Journal of Computational and Applied Mathematics* 31.1 (July 1990), pp. 3–10.

[2] Altera. *Quartus II TimeQuest Timing Analyzer, Quartus II 9.0 Handbook, Volume 3*. Ed. by Altera. Mar. 2019.

[3] M. Bakiri, C. Guyeux, J.-F. Couchot, and A. K. Oudjida. "Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses". en. In: *Computer Science Review* 27 (Feb. 2018), pp. 135–153.

[4] J. K. Barr, B. A. Bradley, B. T. Hannigan, A. M. Alattar, and R. Durst. "Layered security in digital watermarking". en. US8190901B2. May 29, 2012.

[5] M. Baudet, D. Lubicz, J. Micolod, and A. Tassiaux. "On the Security of Oscillator-Based Random Number Generators". In: *Journal of Cryptology* 24.2 (Apr. 1, 2011), pp. 398–425.

[6] K. Bhattacharjee, K. Maity, and S. Das. "A Search for Good Pseudo-random Number Generators : Survey and Empirical Studies". In: *arXiv:1811.04035 [cs]* (Nov. 2018). arXiv: 1811.04035.

[7] Y. Bi, K. Shamsi, J.-S. Yuan, P.-E. Gaillardon, G. D. Micheli, X. Yin, X. S. Hu, M. Niemier, and Y. Jin. "Emerging Technology-Based Design of Primitives for Hardware Security". In: *J. Emerg. Technol. Comput. Syst.* 13.1 (Apr. 2016).

[8] R. G. Brown. *Dieharder, A Random Number Test Suite. version Version 3.31. 1, Duke University Physics Department.* https://webhome.phy.duke.edu/~rgb/General/dieharder.php. 2004. (Visited on 02/02/2022).

[9] A. Cherkaoui, V. Fischer, A. Aubert, and L. Fesquet. "A Self-Timed Ring Based True Random Number Generator". In: *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*. 2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems. May 2013, pp. 99–106.

[10] A. Cherkaoui, V. Fischer, A. Aubert, and L. Fesquet. "Comparison of Self-Timed Ring and Inverter Ring Oscillators as entropy sources in FPGAs". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2012, pp. 1325–1330.

[11] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert. "A Very High Speed True Random Number Generator with Entropy Assessment". In: *Cryptographic Hardware and Embedded Systems - CHES 2013*. Ed. by G. Bertoni and J.-S. Coron. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 179–196.

[12] R. Cofer and B. F. Harding. *Chapter 9 - Design Constraints and Optimization*. Ed. by R. Cofer and B. F. Harding. Embedded Technology. Burlington: Newnes, 2006. Chap. 9, pp. 137–154.

[13] J. Day and H. Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (Dec. 1983), pp. 1334–1340.

[14] O. Elissati, S. Rieubon, E. Yahya, and L. Fesquet. "Self-Timed Rings: A Promising Solution for Generating High-Speed High-Resolution Low-Phase Noise Clocks". In: *VLSI-SoC: Forward-Looking Trends in IC and Systems Design*. Ed. by J. L. Ayala, D. Atienza Alonso, and R. Reis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 22–42.

[15] European Commission. *Art. 32 GDPR: Security of processing*. Ed. by European Commission. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679. Apr. 27, 2016. (Visited on 05/10/2021).

[16] R. Forgrave. *The Man Who Cracked the Lottery*. en. https://www.nytimes.com/interactive/2018/05/03/magazine/money-issue-iowa-lottery-fraud-mystery.html. May 2018. (Visited on 06/24/2021).

[17] M. Green. *The many flaws of Dual_EC_DRBG*. https://blog.cryptographyengineering.com/2013/09/18/the-many-flaws-of-dualecdrbg/. 2013. (Visited on 02/19/2021).

[18] J. Hamon, L. Fesquet, B. Miscopein, and M. Renaudin. "High-Level Time-Accurate Model for the Design of Self-Timed Ring Oscillators". In: *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*. ISSN: 1522-8681. Apr. 2008, pp. 29–38.

[19] M. Y. Hsiao. "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes". In: *IBM Journal of Research and Development* 14.4 (July 1970), pp. 395–401.

[20] D. Hurley-Smith, C. Patsakis, and J. Hernandez-Castro. "On the unbearable lightness of FIPS 140-2 randomness tests". In: *IEEE Transactions on Information Forensics and Security* (2020), pp. 1–1.

[21] JEDEC. *2.5 V ± 0.2 V (Normal Range) and 1.8 V – 2.7 V (Wide Range) Power Supply Voltage and Interface Standard for Nonterminated Digital Integrated Circuits*. Tech. rep. JEDEC, June 2006.

[22] JEDEC. *Interface Standard for Nominal 3 V/3.3 V Supply Digital Integrated Circuits*. Tech. rep. JEDEC, June 2006.

[23] Keysight. *Infiniium UXR-Series Oscilloscopes*. Ed. by Keysight. May 11, 2021.

[24] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2019, pp. 1–19.

[25] C. Krieg, C. Wolf, and A. Jantsch. "Malicious LUT: A stealthy FPGA Trojan injected and triggered by the design flow". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ISSN: 1558-2434. Nov. 2016, pp. 1–8.

[26] C. Krieg, C. Wolf, A. Jantsch, and T. Zseby. "Toggle MUX: How X-optimism can lead to malicious hardware". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2017, pp. 1–6.

[27] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. PhD thesis. Gusshausstrasse 27–29/384, 1040 Wien: Vienna University of …, 2019.

[28] Lattice. *Timing Closure*. Ed. by Lattice. Oct. 2013.

[29] J. W. Lee, Daihyun Lim, B. Gassend, G. E. Suh, M. v. Dijk, and S. Devadas. "A technique to build a secret key in integrated circuits for identification and authentication applications". In: *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*. June 2004, pp. 176–179.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. "Meltdown". In: *arXiv:1801.01207 [cs]* (Jan. 2018). arXiv: 1801.01207.

[31] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules*. en. Tech. rep. Dec. 2002.

[32] O. Petura, U. Mureddu, N. Bochard, V. Fischer, and L. Bossuet. "A Survey of AIS-20/31 Compliant TRNG Cores Suitable for FPGA Devices". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–10.

[33] T. Rahman. "Hardware-based security primitives and their applications to supply chain integrity". PhD Dissertation. University of Florida, 2017.

[34] M. Rostami, F. Koushanfar, and R. Karri. "A Primer on Hardware Security: Models, Methods, and Metrics". In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1283–1295.

[35] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, A. Rukhin, J. Soto, M. Smid, S. Leigh, M. Vangel, A. Heckert, J. Dray, and L. E. B. Iii. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. 2002.

[36] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423.

[37] G. E. Suh and S. Devadas. "Physical Unclonable Functions for Device Authentication and Secret Key Generation". In: *2007 44th ACM/IEEE Design Automation Conference*. ISSN: 0738-100X. June 2007, pp. 9–14.

[38] B. Sunar, W. J. Martin, and D. R. Stinson. "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks". In: *IEEE Transactions on Computers* 56.1 (Jan. 2007), pp. 109–119.

[39] R. Tarjan. "Depth-first search and linear graph algorithms". English. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE Computer Society, Oct. 1971, pp. 114–121.

[40] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: (Jan. 1, 1976), pp. 31–42.

[41] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/. (Visited on 02/02/2022).

[42] Xilinx. *UG1224 VCU118 Evaluation Board User Guide*. Ed. by Xilinx. 1.4. Oct. 17, 2018.

[43] Xilinx. *UG903 Vivado Design Suite User Guide: Using Constraints*. Ed. by Xilinx. 2019.1. June 21, 2019.

[44] Xilinx. *UG945 Vivado Design Suite Tutorial: Using Constraints*. Ed. by Xilinx. 2019.1. June 24, 2019.

[45] M. Yildiz, J. Abawajy, T. Ercan, and A. Bernoth. "A Layered Security Approach for Cloud Computing Infrastructure". In: *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*. ISSN: 2375-527X. Dec. 2009, pp. 763–767.

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria February 2022

Benedikt Tutzer