Efficient On-Chip Randomness Testing Utilizing Machine Learning Techniques

Vojtech Mrazek, Lukas Sekanina, *Senior Member, IEEE*, Roland Dobai, Marek Sys, and Petr Svenda

***Abstract*— Randomness testing is an important procedure that bit streams, produced by critical cryptographic primitives such as encryption functions and hash functions, have to undergo. In this paper, a new hardware platform for the randomness testing is proposed. The platform exploits the principles of genetic programming, which is a machine learning technique developed for the automated program and circuit design. The platform is capable of evolving efficient randomness distinguishers directly on a chip. Each distinguisher is represented as a Boolean polynomial in the algebraic normal form. The randomness testing is conducted for bit streams that are either stored in an on-chip memory or generated by a circuit placed on the chip. The platform is developed with a Xilinx Zynq-7000 All Programmable System on Chip that integrates a field programmable gate array with on-chip ARM processors. The platform is evaluated in terms of the quality of randomness testing, performance, and resources utilization. With power budget less than 3 W, the platform provides comparable randomness testing capabilities with the standard testing batteries running on a personal computer.**

***Index Terms*— Evolutionary computation, field-programmable gate arrays (FPGAs), random sequences.**

I. Introduction

CRYPTOGRAPHIC primitives, such as encryption functions, one-way hash functions, pseudorandom number generators, and hardware random number generators [exploiting a suitable physical process to obtain a truly random bit stream (RBS)], are often implemented as electronic circuits directly on a chip. Ideally, the output bits they produce should be statistically indistinguishable from the outputs of a truly random number generator. However, this crucial property can be partly lost because of various unpredictable faults, changes in the environment or hidden design flaws. Randomness testing is, thus, an important procedure that critical cryptographic primitives should regularly undergo. Various statistical test suites (STSs) such as National Institute of Standards and Technology (NIST) STS, Dieharder, and TestU01 are routinely employed for randomness testing. However, they are primarily intended for an offline testing conducted on a common processor.

Manuscript received January 22, 2019; revised May 13, 2019; accepted June 8, 2019. Date of publication July 10, 2019; date of current version November 22, 2019. This work was supported by the Czech Science Foundation under Grant GA16-08565S. *(Corresponding author: Lukas Sekanina.)*

V. Mrazek, L. Sekanina, and R. Dobai are with the IT4Innovations Centre of Excellence, Faculty of Information Technology, Brno University of Technology, 612 66 Brno, Czechia (e-mail: mrazek@fit.vutbr.cz; sekanina@fit.vutbr.cz; dobai@fit.vutbr.cz).

M. Sys and P. Svenda are with the Faculty of Informatics, Masaryk University, 602 00 Brno, Czechia (e-mail: syso@mail.muni.cz; xsvenda@mail.muni.cz).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2923848

Recently, a new randomness testing procedure has been proposed [1]. As this procedure is based on creating specific Boolean functions acting as randomness distinguishers, there is a great potential to develop its efficient circuit implementation. The design of desired distinguishers is based on searching in the space of Boolean functions represented in an algebraic normal form (ANF), i.e., as polynomials with product terms summed by means of an exclusive-OR operator. A heuristic approach utilizing a brute force search method was employed to obtain desired Boolean functions by means of a software implementation [1]. Recent work [2] showed that machine learning techniques, particularly genetic programming (GP), can also provide high-quality Boolean distinguishers, but with a lower computational effort invested in comparison with the brute force algorithm.

In this paper, we propose a hardware implementation of the GP method [2] that enables us to evolve efficient randomness distinguishers directly on a chip. The objective is to find Boolean functions (so-called distinguishers) that successfully distinguish a given bit stream from an RBS. These bit streams are either stored in an on-chip memory (as a result of some on-chip computation) or generated "on the fly" by a cryptographic primitive (circuit) implemented on the chip. The fitness function, determining the quality of candidate solutions, is based on the so-called Z-score (see Section III-B) which is a numerical outcome of a statistical randomness test comparing two data sequences.

The platform has been developed with a Xilinx Zynq-7000 All Programmable System on Chip that integrates a field programmable gate array (FPGA) with on-chip ARM processors. The platform consists of three main components: 1) configurable Boolean distinguishers (CBDs) implemented as digital circuits in the FPGA; 2) a search algorithm implemented in an on-chip ARM core and used to generate candidate distinguishers that are evaluated in CBDs; and 3) a memory subsystem storing the data sequences undergoing the randomness testing if they are not directly generated by a circuit component. If multiple instances of CBDs exist, several candidate distinguishers can be evaluated in parallel. The platform was evaluated in terms of the quality of randomness testing (by means of the data sets produced by several cryptographic primitives), performance, and resources utilization, and compared with a relevant hardware implementation of randomness testing.

The main contribution of this paper is that the algorithm and the software implementation described in [2] can be implemented as an embedded solution for sensitive on-demand statistical test requiring a fraction of energy in comparison to

a personal computer. The design tool reports less than 3 W for the Zynq chip. This paper also provides a detailed analysis of the tradeoffs between the area occupied in the FPGA and the randomness testing time.

The rest of this paper is organized as follows. Section II introduces the principles of statistical randomness testing of cryptographic primitives and its hardware implementations, evolutionary algorithms (EAs), GP, hardware implementations of GP (by means of evolvable hardware), and relevant applications of EAs in the domain of this paper. The idea of Boolean distinguishers and the methods based on the brute force search and the evolutionary search of Boolean distinguishers are presented in Section III. The proposed hardware platform, capable of evolving efficient Boolean distinguishers in FPGA, is introduced in Section IV. Section V deals with the experimental evaluation of the platform. Section VI provides the use case in which randomness testing is conducted online, for a pseudorandom number generator implemented on a chip. Conclusions are given in Section VII.

II. Related Work

A. Cryptographic Primitives and Randomness Testing

Cryptographic primitives such as encryption functions, one-way hash functions, and pseudorandom number generators (RNGs) are well-established, low-level cryptographic algorithms (or hardware components) that are used as building elements of cryptographic protocols in computer applications. Designing a new cryptographic primitive is a very time-consuming job even for experts. These primitives have to be tested well by the cryptologist community before their routine deployment. The history carries many examples of serious flaws in cryptographic algorithms [3]–[5]. Cryptanalysis conducted by a skilled human cryptanalyst is by far the most successful approach to assess the overall security of an algorithm.

Some automation is, however, possible in the first phases of the cryptanalysis, e.g., by using randomness testing suites such as the *NIST STS* [6] or *Dieharder* [7]. These tests can be applied to check whether the produced bits are correct in terms of statistical properties and to reveal a potentially undesired behavior indicating a deviation from randomness. Such a defect signalizes a potential flaw in the algorithm design. As these testing suites are limited only to the testing of predefined patterns on certain statistical defects, others potential flaws will remain unnoticed.

In some cases, statistical testing is also periodically conducted during the deployment of the primitives. For example, superseded standard FIPS 140-1 [8] required on-demand statistical RNG test using four statistical tests. New standard FIPS 140-3 [9] requires additional continuous testing of the RNG using simple continuous random bit generation (RBG) test or RBG entropy source test. As summarized in [2], a standard statistical test examines randomness of data by looking at a specific feature (e.g., the number of ones, the number of ones in blocks, and so on). The empirical tests of randomness are typically based on the statistical hypothesis testing.

These tests evaluate the null hypothesis, i.e., "data being tested are random." Each test computes a specific statistic of bits or block of bits. A test checks whether the observed test statistic for analyzed bit stream happens to be in the extreme (tail) parts of the null distribution (distribution of test statistic of random data). In such a case, the hypothesis is rejected, and the data are considered as nonrandom. Formally, a test statistic is transformed to a p-value (using the null distribution) representing the probability that a perfect random number generator would have produced a bit stream "less random" (i.e., more extreme according to the analyzed feature) than the tested bit stream [6]. A small p-value (below 0.01) is typically interpreted as the tested data not being random.

In general, the empirical tests of randomness are based on the following steps. First, a histogram of patterns for the given data set is computed by the test. Then, the histogram is reduced into a single value representing its "randomness" according to the analyzed feature. Finally, p-value is typically calculated from the observed test statistic using the null distribution.

Tests are grouped to testing suites (also called batteries) to provide more complex randomness analysis. NIST STS [6], Dieharder [7] (an extended version of the Diehard), and TestU01 [10] are the most commonly used batteries for the statistical randomness testing. Although test batteries consist of a set of tests, their testing ability is limited since the feature being examined is fixed for each test. However, it is believed that by using suitable type and amount of features, randomness can be confirmed with a desired level of confidence.

Recent work [1], [2], [11] showed that carefully constructed Boolean functions can provide comparable results, yet much faster and using lower data volumes in comparison with the commonly used STSs. This approach will be elaborated on in detail in Section III.

B. Evolutionary Algorithms and Cryptography

Genetic algorithms, GP, evolutionary strategies, and other search methods inspired in the Darwinian theory of evolution and in the principles of neo-Darwinism are collectively referred to as EAs. They are traditionally used to solve hard optimization problems by means of a parallel search in the space of all feasible solutions. In order to do so, EAs employ a set of candidate solutions (the so-called population), new candidate solutions are created by bioinspired operators such as crossover and mutation, and the search is driven by an objective function, called the fitness function. GP can also be seen as a machine learning technique that can automatically design computer programs without requiring the user to know or specify the form or structure of the solution in advance [12]. As logic networks can also be treated as candidate solutions, GP has been employed to automatically design digital circuits, for example, using a version of GP denoted as Cartesian GP (CGP) [13]. By evolvable hardware, we mean a circuit evolution conducted directly on a chip, typically by means of GP [14].

In the context of cryptography, EAs have been applied to solve quite a diverse set of problems, yet all showing a common property—they can be formulated as a search problem.

Picek's tutorial [15] provides a list of tasks where EAs proved successful in cryptology.

Pseudorandom generators have traditionally been evolved by the EA community (e.g., [16], [17]). One of the requirements for their use for cryptographic purposes is a low implementation cost. CGP was employed to produce such type of pseudorandom generators [18].

Boolean functions with specific properties (for example, highly nonlinear and balanced functions) are introduced to cryptographic primitives because they make a possible attack on them more difficult. Genetic algorithms (e.g., for S-Box generation [19]) as well as GP (e.g., for bent function [20] and high correlation immunity function [21] designs) were successfully applied, improving the state-of-the-art results. A general scheme for the design of block ciphers by means of GP was introduced in [22]. To best of our knowledge, there has been no research related to evolvable hardware dealing with randomness testing.

C. Circuit Evolution in FPGAs

FPGAs have always been an attractive platform for the on-chip circuit evolution because they provide electronic circuits that can relatively easy be configured and evolved by means of EAs (see [14, Ch. 2]).

Evolvable hardware systems based on FPGAs have been developed for more than 2 decades, see a recent survey [23]. In the FPGA-based evolvable hardware, EA (GP) generates candidate configurations (the so-called chromosomes in the EA terminology) that are used to configure the reconfigurable blocks of the FPGA. Once a new candidate circuit is established on the basis of this configuration, it is evaluated to obtain the fitness score. The evaluation is performed for all candidate circuits in the population either sequentially or in parallel. New populations are created using genetic operators which can be implemented in the software or as circuits. The process is repeated until a required solution is obtained or a predefined number of generations is exhausted. From the designer perspective, the key decisions (determining the area on a chip, performance, and flexibility) are how candidate circuits will be implemented and reconfigured and where EA will be implemented.

The circuit reconfiguration can directly be performed at the level of the configuration bit stream for some FPGA families. The concept of dynamic partial reconfiguration (DPR) is then used which allows designers to modify a part of the FPGA while other parts of the FPGA can operate unaffected [24]–[27]. There is a configuration port (e.g., Internal Configuration Access Port in Xilinx Virtex chips) that enables to accomplish the partial reconfiguration from a device (e.g., a processor) located inside the FPGA. The elementary unit of the FPGA which can undergo the reconfiguration is represented by the so-called frame. As one frame typically contains thousands of bits, the reconfiguration time is long even if only a minimal change in the FPGA configuration is requested. There are many constraints imposed on the configuration process in order to perform the reconfiguration safely. Hence, DPR is relatively a time-demanding operation.

A different approach is to reconfigure a virtual reconfigurable circuit (VRC) that is built on the top of the FPGA using multiplexers and application-specific processing elements [28], [29]. Here, the reconfiguration means just writing a set of registers that holds the control signals for the multiplexers. VRCs have been developed in order to avoid slow and not-well-supported reconfiguration mechanisms existing in former FPGAs. An obvious disadvantage is that the multiplexers needed to ensure the reconfiguration introduce the additional area and delay overhead in the resulting circuits. A recent trend is to combine DPR with VRC to gain benefits from both the approaches and eliminate their disadvantages [26], [30].

The EA is implemented either outside the FPGA (e.g., in a personal computer [24]) or inside the FPGA. The second option is currently a preferred solution as the EA is, in fact, a software which can be executed in on-chip processors such ARM or MicroBlaze [27], [30], [31]. Another approach is to implement the EA as a specialized circuit using resources available in the FPGA. However, this solution is useful only in very specific applications. The EA implemented in software is easy to modify and configure. Its performance is usually sufficient because executing the genetic operators typically requires a fraction of the time with respect to the candidate circuit evaluation which is typically accelerated in the programmable logic (PL).

In the context of FPGA-based evolvable hardware systems, various implementation options for the circuit evaluation, fitness calculation, and genetic operators were surveyed and analyzed [23], [27], [30]. The state-of-the-art approach is to implement the EA as a program executed in an on-chip processor and evaluate candidate circuits by means of DPR, VRC, or their combination. This approach is also suitable for Xilinx's reconfigurable Zynq chip [32]. Zynq integrates a PL, a dual-core ARM Cortex A-9 processor and numerous I/O subsystems and memories. It also utilizes new reconfiguration options by means of a processor configuration access port (PCAP) which can manage reconfigurations from the embedded ARM cores. PCAP has been used for the purposes of evolvable hardware, for example, in [26] and [30].

D. Hardware Implementation of Randomness Testing Methods

General-purpose state-of-the-art randomness testing frameworks (such as FIPS, NIST STS, and Dieharder) are typically implemented on common processors and used for the off-line randomness testing. Full hardware implementations of these frameworks are quite rare. Four tests of the first version of the FIPS 140-2 standard were implemented on the Actel Fusion FPGA [33]. Four overlapping tests out of 16 Diehard battery tests required 16 518 LUTs of the Xilinx Virtex 5 FPGA chip running at 151 MHz. Almost complete hardware implementation of the NIST STS suite was presented in [34], where the authors employed DPR to squeeze 14 tests of NIST STS into a Xilinx Virtex II Pro FPGA V2P30 chip. Their implementation of all tests would require 32 230 LUTs (estimated according to [34]); thanks to the use of DPR, the tests were executed on an FPGA containing only 27 392 LUTs.

Other FPGA-based implementations of randomness testing are focused on revealing specific weaknesses in the (pseudo or true) random data streams produced directly on a chip. One of these approaches is to implement only a subset of a commonly used test battery (such as NIST STS tests [35], [36]). The objective is to provide fast and compact, but, in principle, simplified on-the-fly randomness testing. It has to be noted that although the hardware implementation of some tests is simplified with respect to the standard software version, no experimental analysis of the quality of testing on the real data was reported in these papers. Another approach in this direction is developing completely new hardware architectures for randomness testing. A new architecture suitable for learning cumulative distribution functions that were applied to nonparametric runtime testing for bias in RNGs was presented in [37]. The authors pointed out that "a notable drawback of our algorithm is that in order to function optimally it requires the stream under analysis to generate variates in a time-independent manner." In addition to statistical randomness testing, some implementations also test the internal quality of the entropy source in the case of True RNGs [38], but these approaches are out of the scope of this paper.

We can conclude this section by emphasizing the fact that the on-the-fly simplified implementations discussed in the previous paragraph cannot be considered as implementations of general-purpose randomness testing frameworks. As the randomness testing showing the quality comparable with NIST STS is needed in various on-chip applications and in low-cost portable systems and existing implementations of this type are obsolete [33], [34], [39], we propose a new FPGA implementation based on Boolean distinguishers in this paper.

III. Boolean Functions as Randomness Distinguishers

Based on constructing suitable Boolean functions, Sys *et al.* [1] introduced a method looking for distinguisher of a (tested) bit stream (TBS) and an RBS. These Boolean functions are represented in an ANF and evaluated by means of the Z-score. This section introduces ANF for Boolean functions, Z-score and two methods developed for constructing desired Boolean distinguishers.

A. Algebraic Normal Form

ANF is a canonical polynomial representation of a Boolean function. Formally, every Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can uniquely be represented by a polynomial in n variables

$$f(x_1, \ldots, x_n) = \bigoplus_{I \in \mathcal{P}(M)} a_I x^I = \bigoplus_{I \in \mathcal{P}(M)} a_I \prod_{i \in I} x_i$$

where $\oplus$ is the exclusive-OR operator, $\mathcal{P}(M)$ denotes the powerset of $M = \{1, \ldots, n\}$, n denotes the number of variables, and $a_I \in \{0,1\}$. The product x^I is denoted as monomial. The algebraic degree of f, denoted by $deg(f)$, is the maximum of the degrees of the monomials in f. Considering ANF representation and lexicographically ordered powerset $\mathcal{P}(M)$, every Boolean function can uniquely be represented by the binary sequence $(a_0, a_1, \ldots, a_M)$ of length 2^n, where the coefficient a_0 corresponds with the empty set and a_M corresponds to the set M.

For example, Boolean function $g(x_1, x_2, x_3) = x_1 \oplus x_1x_3 \oplus x_2x_3$ is in ANF, its algebraic degree is $deg(g) = 2$ and it consists of three monomials ($|I| = 3$). This function is uniquely determined by $I = \{\{1\}, \{1,3\}, \{2,3\}\} \subseteq \mathcal{P}(\{1,2,3\})$ and it can be represented as $(a_0, a_1, a_2, a_3, a_{12}, a_{13}, a_{23}, a_{123}) = (0,1,0,0,0,1,1,0)$.

The reasons for choosing ANF for representing a Boolean distinguisher are: 1) a logic expression in ANF can easily be interpreted by human (which is important for cryptanalysts) and 2) the search space induced by the ANF representation is naturally constrained for a given n by the requirements imposed on the ANF.

B. Z-Score

The quality of a Boolean distinguisher is measured in terms of the so-called Z-score, which is a generalization of the Monobit test that counts the number of ones (#1) and zeros (#0) in the analyzed TBS and examines whether the numbers are close to each other as it would be expected for random data. The N-bit data stream to be analyzed is divided to B nonoverlapping blocks consisting of n bits ($N = B \cdot n$). The blocks serve as inputs for Boolean function $f(x_1, \ldots, x_n)$. The sum #1 of results of f when applied to the blocks, is computed. The #1 together with the probability of evaluating f to one for random input (denoted as p) is used to compute Z-score

$$\text{Z-score} = \frac{\#1 - pB}{\sqrt{p(1-p)B}}. \quad (1)$$

Fig. 1 shows how Z-score is computed by a three-input Boolean distinguisher when sequences TBS (T) and RBS (R) contain $B = 4$ blocks. Z-score was employed as it normalizes a binomial distribution of #1 [1]. In addition to that, it defines the statistical distance between observed and expected numbers of ones for random data.

Sys *et al.* [1] proposed to use the computed test statistic directly as the measure of the strength of distinguishers. The Z-score of the tested data is compared with the Z-score calculated for a random data that serves as a baseline for randomness comparison. A bigger (absolute) difference of these Z-scores then indicates a stronger randomness distinguisher and conversely.

C. Brute Force Method

A heuristic method, based on the brute force search, was proposed to find suitable Boolean distinguishers, i.e., n-input Boolean functions represented in ANF [1]. The method consists of two phases. The goal of the first phase is to enumerate all monomials whose degree is less or equal to d, where d is very small (usually $d = 3$) to make the enumeration feasible. From all these results, a set of top 10 (or 100 or 1000, depending on experiment) monomials showing the best Z-score is selected. In the second phase, more complex distinguishers are constructed from the top monomials, again by enumeration. The monomials are combined together to

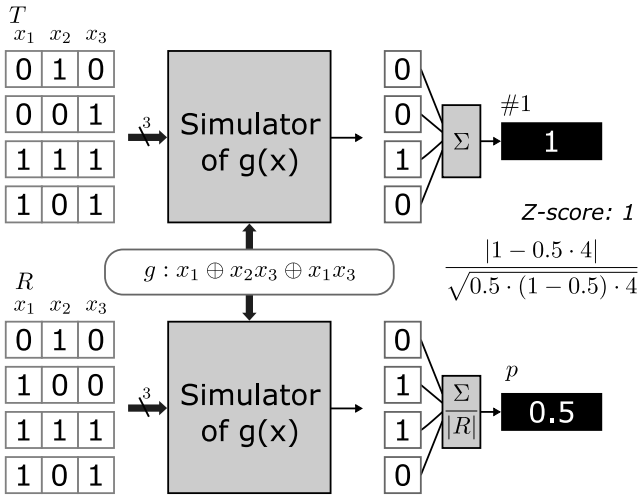

Fig. 1. Determining the fitness score (Z-score) of Boolean distinguisher g represented using logic function $x_1 \oplus x_2x_3 \oplus x_1x_3$. Test sequence T and random sequence R consist of $B = 4$ samples.

obtain a Boolean distinguisher in the form $f = b_1 \oplus \cdots \oplus b_k$, where b is a preselected monomial and $k \leq 3$. Although this algorithm significantly reduced the set of distinguishers that are considered by the evaluation, it was shown in [1] that resulting distinguishers can reveal nonrandomness of various difficult TBSs.

BoolTest[1] [11] is a software implementation of the brute force search method. The performance (strength and speed) of the BoolTest significantly depends on the used parameters (k, d). BoolTest provides comparable randomness testing to the NIST STS test suite in terms of strength (comparable number of rounds for 20 tested cryptographic functions) as stated in [1], but the time is radically decreased. The randomness analysis of 100 Mib of data (regardless of its randomness quality) with the NIST STS (default setting) takes 8.5 min on ordinary laptop.[2] BoolTest with four very simple configurations ($k, d \in \{1, 2\}$, but with larger $n \in \{384, 512\}$) used in [1] takes 5, 6.6, 3.8, and 3.9 s, respectively. However, for a more challenging configuration with $n = 128$, $k = 3$, and $d = 3$, the time needed to process 100 Mib of data is about 1.1 min. This underlines the necessity to employ an efficient heuristics such as EA.

In summary, BoolTest can be used as a fast alternative to existing batteries and/or to complement their results. Moreover, with decreasing amount of the available data, BoolTest outperforms NIST STS battery more and more. The direct interpretability of a Boolean function-based distinguisher adds benefit for a human cryptologist interested in the more detailed analysis of weaknesses present in inspected cryptographic function. Among others, the previously unknown bias in the output of C rand() and Java Random generators was discovered using BoolTest [1].

D. Evolutionary Design of Distinguishers

The EA presented in [2] is based on a $(1 + \lambda)$ search strategy known from CGP [13]. In this algorithm, λ offspring individuals are created by a point mutation from one parent—the highest scored individual of the previous population.

In order to encode a Boolean function containing up to k monomials with degree d, a string of genes (integers) consisting of $d \times k$ items is employed. The items are divided into k tuples. Each tuple is associated with a single monomial and defines its inputs. The positive value of a gene determines the index of the input variable involved in the monomial. The negative value means that this item is ignored. For example, a candidate Boolean function

$$g = x_1x_5x_{40} \oplus x_{20}x_{75} \oplus x_{99}x_{76} \oplus x_{56}$$

is encoded as

$$(1, 5, 40; 20, 75, -30; 99, -55, 76; -1, 56, -112)$$

for $n = 128$, $d = 3$, and $k = 4$. When the mutation operator is applied, up to h genes can be modified. The mutation either inverts the sign of a chosen gene or replaces its value by a randomly generated, but valid integer.

Mrazek *et al.* [2] compared the aforementioned EA (denoted EVO in [2]), the two-stage brute force approach (denoted BF/BF) and the brute force approach combined with EA, in which either the first or the second stage was performed by EA (denoted EVO/BF and BF/EVO). The best results were reported for the EVO approach that was tested for $d \leq 10$ and $k \leq 20$. On seven complex data sets analyzed in [2], EVO provided better distinguishers and the execution time was, in practice, reduced 40 times with respect to the BF approach. Moreover, EVO enabled to construct more complex distinguishers utilizing higher polynomial degrees, which is intractable by the brute force approach.

IV. Evolutionary Design of Boolean Distinguishers in FPGA

This section presents a new Zynq-based platform for the evolutionary design of randomness distinguishers. As Fig. 2 shows, after the random initialization, the platform automatically generates candidate polynomials (randomness distinguishers) that are evaluated (simulated) using the test and RBSs available on the platform. The evaluation is accelerated in the FPGA. The Z-score is calculated for each candidate distinguisher and the best of them (α) is compared against the best-scoring distinguisher (p) obtained so far. The globally best-scoring distinguisher is used as the parent for the next generation of distinguishers. The process is repeated for a predefined number of iterations. Details of the mapping of this process on the resources available in the Zynq chip are provided in Sections IV-A–IV-D. Together with the hardware implementation, a purely software implementation in C has been developed in order to find suitable parameters of EA and tune the performance before a real hardware design is conducted.

We have developed two use cases for the platform. It can be applied for:

1) offline randomness testing of bit streams stored in a local memory (use case I) or

[1] github.com/crocs-muni/booltest

[2] Processor Intel(R) Core(TM) i5-8250U CPU at 1.60 GHz.

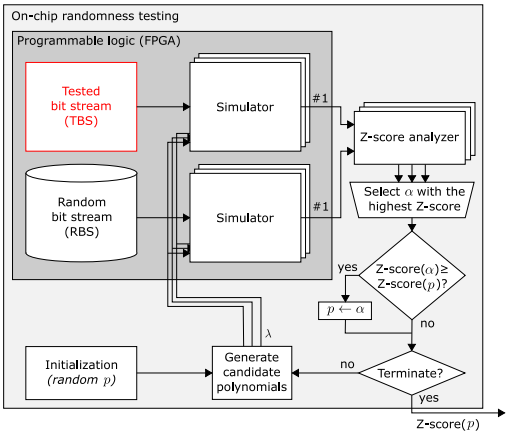

Fig. 2. Basic functionality of the platform developed for on-chip randomness testing.

2) online randomness testing of bit streams generated by an on-chip component (use case II).

Use case I is motivated by the need to create a portable and low-power general-purpose randomness testing system (i.e., neither laptop-based testing nor simplified test batteries are acceptable) that could be used as a component in an FPGA (or eventually in an application specific circuit—ASIC) for randomness testing of various sequences produced or processed on a complex chip. Hence, we suppose that the data are created somewhere on a chip and available in a local memory. One of the applications of such a system is online testing of (pseudo) random data generated by RNGs which is the use case II (i.e., no local memory is used). However, the local memory can also work as a buffer if the data generated on the chip are produced much faster or much slower than Boolean distinguishers can process. While this section deals with use case I, use case II is presented in Section VI.

A. Platform Overview

The concept of the Boolean distinguishers is, in principle, well suited for a hardware implementation because a candidate Boolean distinguisher can straightforwardly be implemented in a reconfigurable combinational circuit in order to ensure its fast evaluation. As the structure of the distinguishers is strictly dictated by ANF, we propose to implement the distinguisher as a combinational pipelined circuit and employ the multiplexers to select the inputs of the monomials. This strategy, corresponding to the VRC principles, supports fast data processing (because of pipelined circuits) and fast reconfiguration (by means of writing to the configuration register). Moreover, if the FPGA is sufficiently large, several VRCs can be implemented on the chip to enable a parallel evaluation of the entire population.

In use case I, both bit streams—TBS (tested) and RBS (reference random)—are stored in a local memory (called BlockRAM or BRAM) of the FPGA. With respect to available resources on the Zynq chip, the chosen size of TBS as well as RBS is $N = 800$ kbits, i.e., the total memory capacity allocated for bit streams is 2×100 kB.

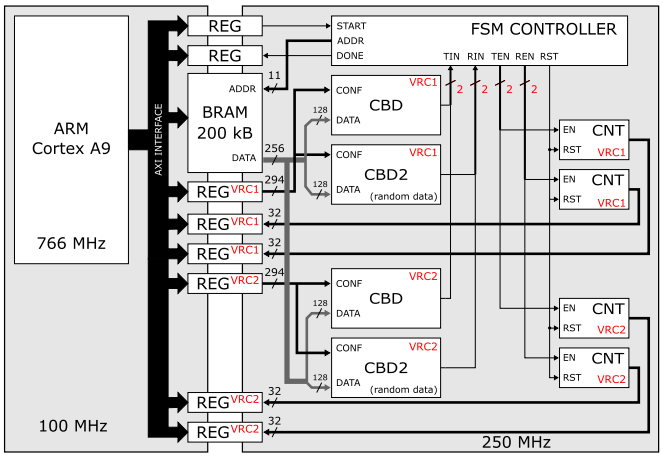

Fig. 3. Platform for evolutionary design of Boolean distinguishers in the XC7Z020 Zynq chip. A pair of CBDs is devoted to the evaluation of a single candidate distinguisher. Multiple CBD pairs are used to evaluate more distinguishers in parallel (shown for $\lambda = 2$).

Fig. 3 shows the proposed implementation of the platform, which can evolve Boolean distinguishers with up to 128 inputs in the FPGA. In order to process TBS and RBS in parallel, there are two independent copies of the configurable Boolean distinguisher (CBD and CBD2) implemented in the PL. They are configured to operate identically, i.e., both copies implement the same candidate Boolean distinguisher whose configuration is generated by EA. Each of them is equipped with a simple counter (CNT) of ones as the number of ones is needed to calculate the Z-score. These counters are enabled by TEN and REN signals if CBD is outputting logic 1. The memory holding TBS and RBS is organized in such a way that 128 bits can be fetched from each bit stream in one clock cycle.

If the FPGA capacity is sufficient, multiple CBD pairs can be instantiated to evaluate more individuals in parallel as shown in Fig. 3. A common utilization of the platform is that the number of CBD pairs equals the number of offspring (λ). TBS and RBS stored in BRAM are then shared by the CBD pairs (see Section IV-C).

The ARM core is employed to calculate the Z-score from the results produced by candidate distinguisher(s). The ARM core also implements the EA and configures CBDs according to evolved configurations.

B. Configurable Boolean Distinguisher

According to its configuration, CBD is designed to implement one Boolean distinguisher from a set of Boolean distinguishers that are determined by three parameters (n, d, k), where n is the number of input bits of Boolean distinguisher, d is the maximum degree, and k is the maximum number of monomials in the corresponding ANF.

With respect to available resources in Zynq and typical sizes of Boolean distinguishers analyzed in [2], CBD is implemented as a combinational circuit for $n = 128, d = 6$, and $k = 6$. Fig. 4 shows that six monomials $(k_1, \ldots, k_6)$ are processed in parallel and their outputs are summed in a six-input exclusive-OR gate. A monomial is implemented

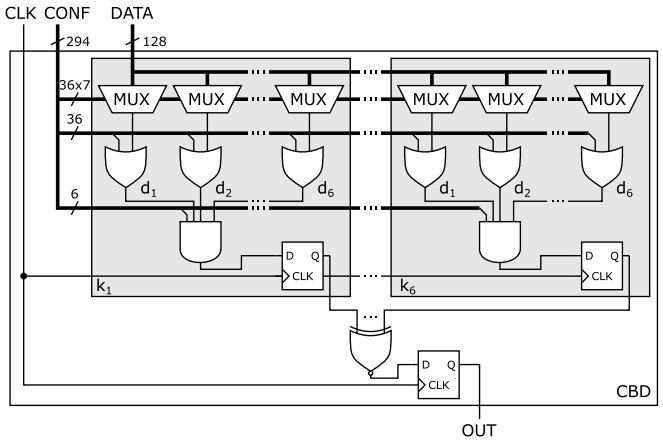

Fig. 4. Configurable Boolean distinguisher for $n = 128$, $d = 6$, and $k = 6$. Configuration register CONF controls the inputs of the exclusive-OR (6 bits) and six monomials ($6 \times (6 + 6 \times 7)$ bits). One monomial needs 6 bits to control the inputs of the AND gate and, for each input, 7 bits are needed to select one of the 128 DATA inputs by the multiplexers. The OR gates enable to disconnect a particular variable in a monomial.

with a six-input AND gate whose inputs are selected from the 128-bit input vector by means of six 128-input multiplexers. A single 128-input multiplexer is implemented using 34 LUTs and 28 embedded multiplexers (the so-called F7 and F8 multiplexers in Zynq).

Six OR gates used in each monomial are capable of disconnecting appropriate inputs from the AND gates. This ensures that each AND can effectively process from 0 to 6 inputs, i.e., the whole monomial can eventually be disconnected. If all the inputs are disconnected, the AND generates logic 1. The inputs of the exclusive-OR can analogically be disconnected. If all the inputs are disconnected, the exclusive-OR gate generates logic 0. In order to support pipelined processing, several stages of D registers are inserted into CBD. One stage of D registers is also included to the implementation of 128-input multiplexers.

One pair of CBDs is configured using a configuration register containing the control bits for multiplexers and configuration gates. The configuration register size is 294 bits, where 6 bits control the inputs of the exclusive-OR and $6 \times (6+6 \times 7)$ bits control six monomials. One monomial needs 6 bits to control the inputs of the AND gate and, for each input, 7 bits are needed to select one of the 128 primary inputs by the multiplexer. If λ CBD pairs are employed, the total configuration register size is 294λ bits.

C. Evolutionary Algorithm

We adopted the search method $(1+\lambda)$ used in [2] and similar FPGA-based accelerators of EA (see [13], [26]). As Algorithm 1 shows, the search method that is implemented in the ARM core, starts with a randomly generated population of candidate Boolean distinguishers (line 1), each of them represented using 294 bits. The population is evaluated in CBDs either sequentially or in parallel, depending on the number of CBD pairs on the chip (line 2). The Z-score, representing the fitness value, is assigned to each individual. The best-scored individual (α) is selected as a parent of the new population

Algorithm 1 Search Algorithm

```
1  P ← RandomlyCreateInitialPopulation;
2  EvaluatePopulation(P) in FPGA;
3  fitness(p) ← −∞;
4  i = 0;
5  while (i < n_g) do
6      α ← Select Highest-scored-individual(P);
7      if fitness(α) ≥ fitness(p) then
8          p ← α;
9      P ← {p} ∪ {λ offspring of p created by mutation};
10     EvaluatePopulation(P) in FPGA;
11     i = i + 1;
12 return fitness(p);
```

(lines 6–8). λ offspring are then generated from the parent by means of the mutation operator modifying h genes (integers) of the chromosome and evaluated (lines 9–10). The body of the while loop is repeated n_g times, where n_g is a predefined number of generations. The highest scored individual is the result of the EA.

The execution time of EA is dominated by the number of evaluations ($n_g \times \lambda$) because creating a new population and other supporting tasks conducted by the processor are negligible in the total execution time.

D. Accelerated Evaluation of Distinguishers

Let us assume that only one CBD pair is instantiated in the FPGA. In order to evaluate a candidate Boolean distinguisher, the following steps are performed. The processor sends the chromosome to the configuration registers of both CBDs in 294/32 $\approx$ 11 clock cycles. It has to be noted that the processor can send only 32 bits/clock cycle (at 100 MHz) to PL. In PL, the evaluation is finished in $10+N/n$ clock cycles, where 10 is the initial latency of CBD (i.e., the number of stages in the pipeline), N is the bit stream length and n is the number of inputs bits of the distinguisher. The outcome of the evaluation (i.e., the number of ones in TBS and in RBS) is sent back to the processor (in 2 clock cycles) in which the resulting Z-score is determined and assigned as the fitness value to the chromosome.

If λ CBD pairs are available in the FPGA, the evaluation time ($10+N/n$ clock cycles) remains unchanged, but λ distinguishers instead of one distinguisher are now evaluated within this time. The configuration time and the result storing time are, however, increased proportionally with λ (see Section V-A for detailed evaluation).

V. Results

This section summarizes the results of synthesis and experimental evaluation of the proposed implementation. It deals with use case I in which the bit streams are stored in on-chip BRAM memory blocks.

A. Implementation and Synthesis Results

The proposed platform was described in VHDL language, synthesized with Xilinx Vivado 2018 and implemented on the

TABLE I

NUMBER OF CLOCK CYCLES (REQUIRED FOR CONFIGURATION, EVALUATION, AND STORING THE RESULT), THE EVALUATION TIME FOR λ DISTINGUISHERS, AND THE TOTAL EA EXECUTION TIME IF 40 000 CANDIDATE INDIVIDUALS ARE GENERATED AND EVALUATED USING 1–5 CBD PAIRS

CBD pairs	Clock Cycles			Eval. time	EA time	Speedup
($=\lambda$)	Configuration	Evaluation	Store result	[μs]	[s]	[-]
1	11	6260	2	25.17	1.0068	1.00
2	21	6260	4	25.29	0.5058	1.99
3	31	6260	6	25.41	0.3388	2.97
4	41	6260	8	25.53	0.2553	3.94
5	51	6260	10	25.65	0.2052	4.91

ZC702 Evaluation Kit containing XC7Z020-CLG484-1 chip. The EA was implemented in the ARM core with parameters adjusted according to [2]: $\lambda = 4$, $h = 5$, and $n_g = 10^4$. If not explicitly stated otherwise, all results are reported for distinguishers that can fit into a CBD configured with $n = 128$, $d \leq 6$, and $k \leq 6$.

Due to the pipeline processing, PL can operate at 250 MHz. Up to five CBD pairs can be instantiated in PL. For $N = 800\,000$ and $n = 128$, the evaluation time of a single candidate distinguisher using one CBD pair (see $\lambda = 1$ in Table I) requires $10 + 800\,000/128 = 6260$ clock cycles which corresponds with 25 μs. Some additional time is required for the CBDs configuration (11 clock cycles) and storing the result to the processor (2 clock cycles). The execution time of EA that produces and evaluates $n_g \times \lambda = 40\,000$ candidate distinguishers is then 1.0068 s. Table I summarizes these numbers if 1–5 CBD pairs evaluating 1–5 candidate distinguishers in parallel are instantiated in PL. The obtained speedup with respect to the baseline implementation (i.e., $\lambda = 1$ corresponding with Speedup = 1.00 in Table I) scales almost perfectly with the number of CBD pairs.

Table II gives the resources utilization for $k = 6$, $d = 6$, $N = 800\,000$, and $n = 128$, where the dominant consumer is BRAM needed to store TBS and RBS. Depending on the number of CBDs, our design requires from 12.80% to 33.26% slice LUTs. We were unable to fit more than five CBD pairs to PL because of complicated routing associated with the requirement of high operational frequency. Increasing the complexity of CBD can in some cases slightly improve the quality of randomness testing as reported in [2], but it immediately reduces the speedup in terms of Table I. For our FPGA, only one CBD pair can be instantiated in the PL if $d = k = 10$. This configuration leads to a 66% resources overhead (with respect to $d = k = 6$ and "CBD pairs" = 1 reported in Table II) and power consumption is also higher (2.7 W). A compromise configuration with $d = 6$ and $k = 10$ enables instantiating up to three CBDs with a 25% increase in resources.

B. Data Sets

The method is evaluated on the data (i.e., TBSs) generated by means of the stream cipher (RC4), block cipher (AES), and hash functions (SHA-256, MD6, Keccak). In order to detect some nonrandom sequences in our experiments, we intentionally limited the number of rounds that are performed by these primitives. The test data were generated as the standard keystream for RC4. For the remaining cases, a special stream consisting of 128-bit blocks of minimal Hamming weight was employed at the inputs. This sequence starts with block $B_0 =$ "00...0." Each of next 128 blocks consists of 127 bits "0" and 1 bit "1" on different positions, i.e., $B_1 =$ "00...01,"$B_2 =$ "00...10,"..., $B_{128} =$ "100...." Next blocks consist of 126 bits with "0" and 2 bits of the value "1," etc. The test data were generated as blocks of the sequence processed by one of the functions (SHA-256, etc.) separately. In addition to that, a sequence of random data obtained from /dev/urandom (which is a special file that serves as a pseudorandom number generator in Unix-like operating systems) is considered. The generator is cryptographically secure [40] which means that the numbers it produces can be used in cryptographic applications where high-quality random data are required. In all cases, 100 kB of data is used as this memory capacity is supported in the proposed implementation.

TABLE II

RESULTS OF SYNTHESIS FOR 1–5 CBD PAIRS ($k = d = 6$, $N = 800\,000$, AND $n = 128$) INSTANTIATED IN THE XC7Z020 CHIP

	Available	Chip utilization for 1–5 CBD pairs [%]				
Site type		1	2	3	4	5
Slice LUTs	53200	12.80	17.52	23.17	28.19	33.26
Flip Flops	106400	6.61	7.97	9.33	10.68	12.08
F7 Muxes	26600	4.55	8.88	13.21	17.54	21.87
F8 Muxes	13300	4.79	9.12	13.45	17.78	22.11
Block RAMs	140	47.14	47.14	47.14	47.14	47.14
Power [W]	–	2.35	2.54	2.76	2.94	3.24

C. Results of EA and BF

For the EA setting given in Section V-A, we analyzed how the Z-score depends on d ($d = \{3, 6\}$), k ($k = 1, \ldots, 6$) and selected data streams. Fig. 5 shows the box plots of the Z-score constructed from ten independent runs of EA. For $d \leq 3$, it is possible to run the BF method whose results are shown as the crosses. For example, BF evaluated 349 632 monomials with $d \leq 3$ in the first phase of the BF algorithm introduced in Section III-C. If $k > 1$, the number of distinguishers evaluated by BF is $\binom{s}{k}$, where s is the number of top monomials selected in the first phase.

Our results are consistent with [2] because we implemented the same EA. If $d \leq 3$ and $k \leq 3$, then BF is able to find better solutions than EA. The reason is that there are usually only a few "the best solutions" which BF can always detect while EA does not usually reach them, but EA is very close. If the number of monomial is increased ($d = 3, k = \{4, 5, 6\}$), the monomials selected by BF in the first phase are not good

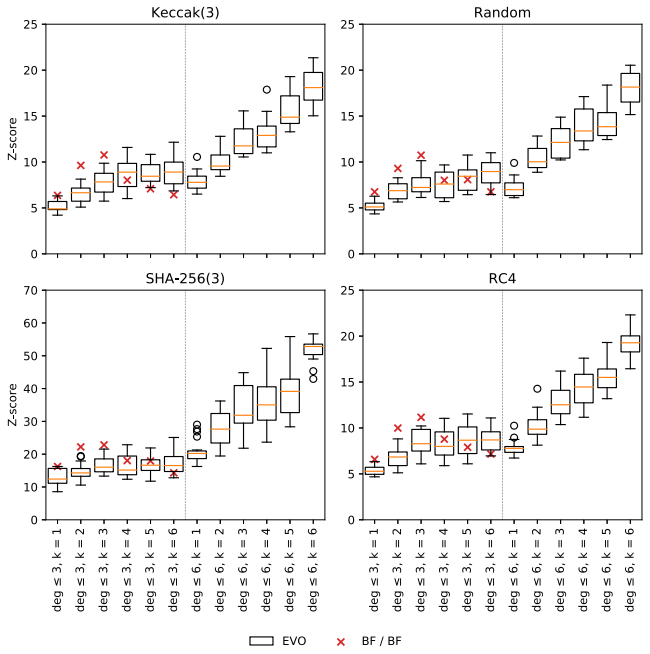

Fig. 5. Z-score obtained for four tested data streams using evolved Boolean distinguishers with various settings of d and k. Red cross: best solution from the brute force method.

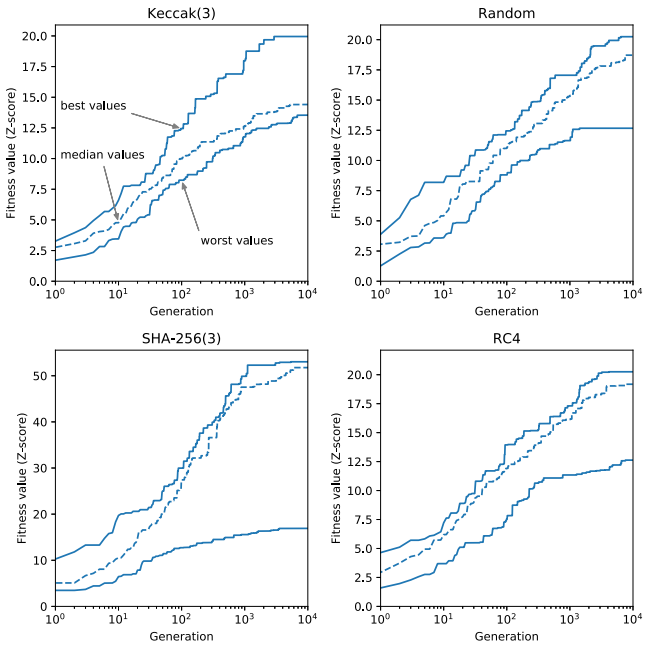

Fig. 6. Z-score of the best, median, and worst distinguishers from the EA runs conducted for four analyzed data streams and $d = k = 6$.

TABLE III

Z-SCORE OF THE BEST DISCOVERED DISTINGUISHERS FOR DATA STREAMS PRODUCED BY VARIOUS CRYPTOGRAPHIC FUNCTIONS

Source of data (rounds)	EVO $d \leq 6, k \leq 6$	EVO $d \leq 3, k \leq 6$	BF $s = 100$
AES(3)	21.51	12.6	12.2
Keccak(2)	84.92	58.7	64.0
Keccak(3)	21.35	12.2	10.8
Keccak(4)	23.12	13.4	11.2
MD6(8)	22.33	13.2	11.2
MD6(9)	20.83	11.5	10.8
MD6(10)	22.57	11.8	12.9
MD6(11)	23.10	12.3	11.4
RC4	22.31	11.5	11.2
SHA-256(3)	56.67	25.1	22.8
SHA-256(4)	21.43	11.8	12.4
random	20.53	11.0	10.7

enough to constitute good distinguishers in the second phase of BF. Hence, EA provides much better results. These results are then significantly improved for $d = 6$ and $k \geq 2$. We do not report any results for BF with $d = 6$ because it is intractable to run BF in these cases.

Fig. 6 shows the fitness (i.e., Z-score) of the best, median, and worst distinguishers from the EA runs conducted for four data streams and $d = k = 6$. It can be seen that 10^4 is a sufficient number of generations to converge as the best solutions stop improving well before this limit (it has to be noted that we used the logarithmic scale on the Generation axis). This behavior is visible in both the basic cases that we investigate, i.e., for testing of: 1) high randomness bit streams (Keccak(3), RC4, Random) in which the Z-score is around 20.0 for our setup of CBD and 2) low randomness bit streams [SHA-256(3)] in which the Z-score is much higher than 20.0.

D. Performance Analysis

We can observe that the Z-score is improving with increasing the number of monomials (k) and degree (d). Higher k and d permit the evolution to search for more complex distinguishers capable of discovering more tricky dependencies among the bits. This is the main advantage of the EA with respect to the BF algorithm.

For 12 data sets, Table III summarizes the Z-scores of the best distinguishers obtained using: 1) the EA operating in the space of all possible functions permitted by CBD; 2) the EA operating in a restricted search space ($d \leq 3$); and 3) BF in which the best 100 monomials were used in the second phase of the brute force method introduced in Section III-C. Based on the comparison with the baseline Z-score of the random sequence (denoted "random"), one can observe that all three methods give consistent results, but EVO ($d \leq 6, k \leq 6$) provides the strongest distinguishers. For example, the data streams produced by three rounds of SHA-256 cryptographic function or two rounds of Keccak are far from random.

For $d = 6$, $k = 6$, $n = 128$, 100 kB data streams, and $\lambda \times n_g = 4.10^4$, the execution time is 1 s, if a single pair of CBDs is instantiated. If four pairs of CBDs are employed the proposed system is faster than a highly optimized single-core software implementation running on a 64-bit Xeon E5-2670 at 2.6 GHz whose execution time is 0.3 s. However, the main contribution of this paper is that the whole system can be implemented as an embedded solution for sensitive on-demand statistical test requiring a fraction of energy in comparison to a personal computer.

E. Comparison With State of the Art

The estimated complexity of the almost complete NIST STS battery implementation is 32 230 LUTs on a Xilinx Virtex II Pro FPGA V2P30 chip [34]. However, the state-of-the-art implementations of randomness testing typically employ a simplified subset of commonly used test batteries [35], [36].

For example, eight NIST STS simplified tests (denoted as NIST-STS-8 in this section) implemented on the Xilinx Spartan-6 XC6SLX45 chip required between 44 and 757 LUTs (1920 LUTs in sum, no other circuits such as controllers or memory interfaces considered) and operated at frequencies between 121 and 203 MHz [35]. Although this approach is clearly less complex than the proposed Boolean distinguishers (requiring 6808 LUTs for one CBD pair running at 250 MHz as given in Section V-A), it cannot provide randomness testing of the same quality as the proposed Boolean distinguishers. In order to demonstrate this property, we generated data streams (TBSs) using various cryptographic functions (23 block ciphers and 13 hash functions). These functions process different types of inputs (such as blocks of counter values, blocks of minimal Hamming weight, and random blocks where consecutive blocks differ in one bit), but reduce the number of internal rounds to get data with bias on the edge of detectability. In total, we generated 443 TBSs whose randomness was tested using the proposed Boolean distinguishers and NIST-STS-8, but NIST-STS-8 was implemented according to the original software version of NIST STS. Therefore, we did not consider the simplifications introduced in [35] because the corresponding hardware implementation is not fully documented.

Results of Boolean distinguishers (with a very basic configuration $d \leq 4, k \leq 2, n = 128$) and NIST-STS-8 are evaluated according to the most significant results for 100 kB TBSs, i.e., the biggest Z-score and the smallest p-value are counted. We consider that a TBS failed NIST-STS-8 test if at least one of computed p-values is below the significance level (0.001). In order to find critical Z-score (for significance level 0.001), we generated 1000 random TBSs (using "/dev/urandom") and computed corresponding maximal Z-scores. Based on this experiment, we can consider that a TBS failed a Boolean distinguisher-based test if the Z-score is greater than 7.66. Note that Z-score ≤ 7.66 reliably represents a random sequence for $d \leq 4, k \leq 2, n = 128, N = 100$ kB. Z-score of TBS generated by, for example, AES(3) is 5.15 and we classified this sequence as "random" which is consistent (as for other cases) with more detailed testing reported in Table III. Despite a very basic configuration of Boolean distinguishers, the proposed method was able to detect nonrandomness for 24 different TBSs which successfully passed NIST-STS-8 [i.e., ARIA(2), CAST(3), Grostl(1), Grostl(2), IDEA(1), KUZNYECHIK(1), MD5(7), MD5(8), MD6(8), RIPEMD160(8), SERPENT(3), SHA1(11),SHA2(6), SIMON(13), SIMON(14), SPECK(7), 3DES (2), TWOFISH(2); CAMELLIA-hw(3), Grostl-hw(2), MD6-hw(8), SHACAL2-hw(7), SIMON-hw(13), SPECK-hw(6)]. On the other hand, NIST-STS-8 detected ten nonrandom TBSs that successfully passed Boolean distinguisher-based testing. The remaining TBSs were evaluated identically by both approaches.

VI. Online Randomness Testing of On-Chip Pseudo RNGs

In order to evaluate the proposed implementation in use case II, we replaced the BRAMs storing TBS and RBS with a hardware implementation of two pseudo RNGs (PRNGs): 1) PRNG_{ref} that is a reference PRNG producing uniformly distributed values and 2) $\text{PRNG}_{\text{test}}$ that is a subject of randomness testing. PRNG_{ref} is implemented according to [41] as a linear feedback shift register (LFSR) producing a 128-bit vector in one clock cycle. In particular, a new input value of PRNG_{ref} is generated by means of XNOR-ing of bits 99, 101, 126, and 128 of a maximum length cycle LFSR.

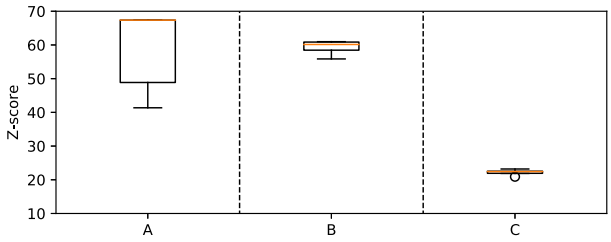

Fig. 7. Z-score obtained from randomness testing of (A) PRNG_{ref} with a stuck-at-1 at bit 7. (B) Low-quality short-cycle PRNG. (C) PRNG_{ref} seeded using a different initial value.

PRNG_{ref} requires 2145 LUTs of the PL, which is 29% of the whole design if a single CBD pair is instantiated. If both PRNGs were identical then they would occupy 58% of the whole design. As the usage of PRNGs leads to less complex routing than for the BRAM-based solution, we were able to instantiate up to seven CBD pairs in PL. It has to be noted that the execution time (and the number of clock cycles) is the same as for the BRAM-based solution if the same number of CBD pairs is instantiated. For four pairs of CBDs, power consumption is 2.79 W, which is comparable to the BRAM-based solution (2.94 W).

The evaluation was performed with the EA having the same setting as reported in Section V-A. We tested three PRNGs on the position of $\text{PRNG}_{\text{test}}$.

1) A PRNG working as PRNG_{ref}, but with a stuck-at-1 at bit 7 (this setup emulates a faulty circuit).
2) A low-quality short-cycle 128-bit PRNG working with bits 2, 4, 5, and 8 as generators of a new bit in next clock cycle.
3) PRNG_{ref} seeded using a different initial value.

In cases 1) and 2), a very high Z-score is expected as these PRNGs are of a low quality. On the other hand, the Z-score obtained for the case 3) should be around 20.0. Fig. 7 showing the box plots of Z-score calculated from ten independent runs of EA, confirms this hypothesis.

VII. Conclusion

An evolvable hardware platform was proposed which is capable of evolving efficient randomness distinguishers directly in the FPGA. We investigated two use cases: 1) the platform is used to evaluate bit streams that are stored in local BRAMs and 2) the bit streams are online generated and immediately processed by evolving distinguishers. In both cases, we analyzed the quality of randomness testing, performance, and resources utilization. The main contribution of this paper is that the software implementation developed for randomness testing and described in [2] can be implemented as an

embedded solution for more sensitive on-demand statistical test (compared to NIST FIPS 140-2 tests) of randomness. The proposed tests are also energy efficient requiring a fraction of energy in comparison to a personal computer. Finally, evolved distinguishers can easily be interpreted by an expert to identify sources of nonrandomness, which is almost impossible to find out when other randomness testing methods are employed.

Our future work will focus on using our platform for randomness testing of bit streams that are produced by various components on a chip. In particular, we plan to integrate the proposed solution to new implementations of hardware random number generators, which generate genuinely random numbers, in order to test their properties in various physical environments.

REFERENCES

[1] M. Sýs, D. Klinec, and P. Švenda, "The efficient randomness testing using Boolean functions," in *Proc. 14th Int. Joint Conf. e-Business Telecommunications (ICETE)*, vol. 4, 2017, pp. 92–103.

[2] V. Mrazek, M. Sýs, Z. Vasicek, L. Sekanina, and V. Matyas, "Evolving Boolean functions for fast and efficient randomness testing," in *Proc. Genetic Evol. Comput. Conf.*, Jul. 2018, pp. 1302–1309.

[3] M. Nemec, M. Sýs, P. Svenda, D. Klinec, and V. Matyas, "The return of coppersmith's attack: Practical factorization of widely used RSA moduli," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct./Nov. 2017, pp. 1631–1648.

[4] M. Vanhoef and F. Piessens, "All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS," in *Proc. 24th USENIX Secur. Symp. (USENIX Security)*, 2015, pp. 97–112.

[5] X. Wang, H. Yu, and Y. L. Yin, "Efficient collision search attacks on SHA-0," in *Advances in Cryptology—CRYPTO*. Berlin, Germany: Springer, 2005, pp. 1–16.

[6] A. Rukhin, "A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications," NIST, Gaithersburg, MD, USA, Tech. Rep. Version STS-2.1., 2010.

[7] R. G. Brown, D. Eddelbuettel, and D. Bauer. (2013). *Dieharder: A Random Number Test Suite 3.31.1*. [Online]. Available: http://www.phy.duke.edu/∼rgb/General/dieharder.php

[8] "FIPS PUB 140-1, security requirements for cryptographic modules," U.S. Dept. Commerce, Nat. Inst. Standards Technol., 1994.

[9] "FIPS PUB 140-3, security requirements for cryptographic modules," U.S. Dept. Commerce, Nat. Inst. Standards Technol., 2007.

[10] P. L'Ecuyer and R. Simard, "TestU01: A C library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, p. 22, Aug. 2007.

[11] M. Sýs, D. Klinec, K. Kubíček, and P. Švenda, "Booltest: The fast randomness testing strategy based on Boolean functions with application to DES, 3-DES, MD5, MD6 and SHA-256," in *E-Business and Telecommunications*. Cham, Switzerland: Springer, 2019, pp. 123–149.

[12] R. Poli, W. B. Langdon, and N. F. McPhee. 2008. *A Field Guide to Genetic Programming*. [Online]. Available: http://lulu.com and http://www.gp-field-guide.org.uk

[13] J. F. Miller, *Cartesian Genetic Programming* (Natural Computing Series). Berlin, Germany: Springer-Verlag, 2011.

[14] M. A. Trefzer and A. M. Tyrrell, Eds., *Evolvable Hardware—From Practice to Application* (Natural Computing Series). Berlin, Germany: Springer, 2015.

[15] S. Picek, "Evolutionary computation and cryptology," in *Proc. Genetic Evol. Comput. Conf. Companion (GECCO)*, Jul. 2017, pp. 1066–1093.

[16] Y. Wang, H. Wang, A. Guan, and H. Zhang, "Evolutionary design of random number generator," in *Proc. Int. Joint Conf. Artif. Intell.*, Apr. 2009, pp. 256–259.

[17] M. Tomassini, M. Sipper, and M. Perrenoud, "On the generation of high-quality random numbers by two-dimensional cellular automata," *IEEE Trans. Comput.*, vol. 49, no. 10, pp. 1146–1151, Oct. 2000.

[18] S. Picek, D. Sisejkovic, V. Rozic, B. Yang, D. Jakobovic, and N. Mentens, "Evolving cryptographic pseudorandom number generators," in *Parallel Problem Solving from Nature*. Cham, Switzerland: Springer-Verlag, 2016, pp. 613–622.

[19] G. Ivanov, N. Nikolov, and S. Nikova, "Reversed genetic algorithms for generation of bijective s-boxes with good cryptographic properties," *Cryptogr. Commun.*, vol. 8, no. 2, pp. 247–276, 2016.

[20] R. Hrbacek and V. Dvorak, "Bent function synthesis by means of Cartesian genetic programming," in *Parallel Problem Solving from Nature–PPSN XIII*. Cham, Switzerland: Springer-Verlag, 2014, pp. 414–423.

[21] S. Picek, C. Carlet, S. Guilley, J. F. Miller, and D. Jakobovic, "Evolutionary algorithms for Boolean functions in diverse domains of cryptography," *Evol. Comput.*, vol. 24, no. 4, pp. 667–694, 2016.

[22] J. C. Hernandez-Castro, J. M. Estevez-Tapiador, A. Ribagorda-Garnacho, and B. Ramos-Alvarez, "Wheedham: An automatically designed block cipher by means of genetic programming," in *Proc. IEEE Int. Conf. Evol. Comput.*, Jul. 2006, pp. 192–199.

[23] R. Salvador, "Evolvable hardware in FPGAs: Embedded tutorial," in *Proc. Int. Conf. Design Technol. Integr. Syst. Nanoscale Era (DTIS)*, Apr. 2016, pp. 1–6.

[24] A. Thompson, "Silicon evolution," in *Proc. 1st Annu. Genetic Program. Conf. (GP)*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Cambridge, MA, USA: MIT Press, 1996, pp. 444–452.

[25] A. Upegui and E. Sanchez, "Evolvable FPGAs," in *Reconfigurable Computing*, S. Hauck and A. DeHon, Eds. San Mateo, CA, USA: Morgan Kaufmann, 2008, pp. 725–752.

[26] R. Dobai and L. Sekanina, "Low-level flexible architecture with hybrid reconfiguration for evolvable hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 3, p. 20, 2015.

[27] J. Mora, R. Salvador, and E. De la Torre, "On the scalability of evolvable hardware architectures: Comparison of systolic array and Cartesian genetic programming," *Genetic Program. Evolvable Mach.*, vol. 20, no. 2, pp. 155–186, 2019.

[28] R. Porter, K. McCabe, and N. Bergmann, "An applications approach to evolvable hardware," in *Proc. 1st NASA/DoD Workshop Evolvable Hardw.*, Jul. 1999, pp. 170–174.

[29] L. Sekanina, "Virtual reconfigurable circuits for real-world applications of evolvable hardware," in *Evolvable Systems: From Biology to Hardware*. Berlin, Germany: Springer-Verlag, 2003, pp. 186–197.

[30] O. Garnica, K. Glette, and J. Tørresen, "Comparing three online evolvable hardware implementations of a classification system," *Genetic Program. Evolvable Mach.*, vol. 19, nos. 1–2, pp. 211–234, 2018.

[31] K. Glette and J. Torresen, "A flexible on-chip evolution system implemented on a Xilinx Virtex-II Pro device," in *Evolvable Systems: From Biology to Hardware*. Berlin, Germany: Springer 2005, pp. 66–75.

[32] *Zynq-7000 All Programmable SoC Overview DS190*. Xilinx, San Jose, CA, USA, 2016.

[33] M. Drutarovský, M. Varchola, and O. Vancák, "Optimized FIPS 140 statistical tests IP core embedded in FLASH based ACTEL FPGA," in *Proc. 53rd Int. Wissenschaftliches Kolloquium*, 2009, pp. 1–5.

[34] D. Hotoleanu, O. Cret, A. Suciu, T. Gyorfi, and L. Vacariu, "Real-time testing of true random number generators through dynamic reconfiguration," in *Proc. 13th Euromicro Conf. Digit. Syst. Design, Architectures, Methods Tools*, Sep. 2010, pp. 247–250.

[35] F. Veljković, V. Rožić, and I. Verbauwhede, "Low-cost implementations of on-the-fly tests for random number generators," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2012, pp. 959–964.

[36] B. Yang, V. Rožić, N. Mentens, W. Dehaene, and I. Verbauwhede, "Embedded HW/SW platform for on-the-fly testing of true random number generators," in *Proc. Design, Automat. Test Europe Conf. Exhib.*, Mar. 2015, pp. 345–350.

[37] A. Althoff and R. Kastner, "An architecture for learning stream distributions with application to RNG testing," in *Proc. 54th Annu. Des. Automat. Conf.*, Jun. 2017, p. 15.

[38] S. U. Hussain, M. Majzoobi, and F. Koushanfar, "A built-in-self-test scheme for online evaluation of physical unclonable functions and true random number generators," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 2, no. 1, pp. 2–16, Jan./Mar. 2016.

[39] A. Vaskova, C. López-Ongil, E. S. Millán, A. Jiménez-Horas, and L. Entrena, "Accelerating secure circuit design with hardware implementation of diehard battery of tests of randomness," in *Proc. IEEE 17th Int. On-Line Test. Symp.*, Jul. 2011, pp. 179–181.

[40] P. Lacharme, A. Rock, V. Strubel, and M. Videau, (2012). *The Linux Pseudorandom Number Generator Revisited*. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01005441

[41] P. Alfke, "Efficient shift registers, LFSR counters, and long pseudo-random sequence generators," Xilinx, San Jose, CA, USA, Tech. Rep. XAPP 052, 2016.