Transforming Write Collisions in Block RAMs into Security Applications

Tim Güneysu, Christof Paar

Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany
{gueneysu,cpaar}@crypto.rub.de

***Abstract*—Due to their versatile and generic structure, Field Programmable Gate Arrays (FPGA) allow dynamic reconfiguration of their logical resources just by loading configuration files. However, this flexibility also opens up the threat of theft of Intellectual Property (IP) since these configuration files can be easily extracted and cloned. In this context, the ability to bind a configuration to a specific device is an important step to prevent product counterfeiting.**

In this paper, we present a novel strategy to identify and authenticate FPGAs in applications using intrinsic, device-specific information (also known as Physically Unclonable Functions). Our solution is based on the output of intentionally induced write collisions in synchronous dual-port block RAM (BRAM). We show that the output of such write collisions can be used to create unique device signatures. In addition to applications for chip identification and authentication, we also propose a solution to efficiently create secret keys on-chip. As a last contribution, we outline how to transform our idea into a circuit for True Random Number Generation (TRNG).

I. Introduction

When Field Programmable Gate Arrays (FPGA) were first introduced in the 1980s, this was a revolutionary step from static ASIC and VLSI solutions to flexible and maintainable hardware applications. It has become possible to avoid the individual and static designs of standard VLSI technology, and instead to compile electrical circuits for arbitrary hardware functions into configuration bit files used to program a fabric of reconfigurable logic. However, the flexibility of SRAM-based FPGAs also brings up the issue of protecting the Intellectual Property (IP) of such circuit layouts from unauthorized duplication or reverse engineering. Unfortunately, a configuration bit file of an FPGA can easily be retrieved from a product and used to clone a device with only little effort. In this context, the generic FPGA technology lacks an intrinsic and unique identification mechanism which allows to bind a given configuration to a specific device. Recognizing this issue, Xilinx recently proposed the Device DNA feature for Spartan-3 A/AN FPGAs, which is a 57 bit unique value which is integrated in each chip during manufacturing [16]. This limited source of unique information can be sufficient for simple authentication scenarios based on check-values, but is definitely not satisfactory from a cryptographic point of view. Note that in this context the cryptanalytic COPACOBANA machine, which is a low-cost FPGA-based cluster consisting of 128 Spartan-3 XC3S1000 FPGAs, can determine a 56 bit key of a DES encryption on average in less than a week [7]. Hence, with only 57 bit of entropy from the Device DNA, brute-force attacks with similar complexity are likely. An alternative approach to integrate a unique feature with *variable* length into FPGAs is based on so called Physically Unclonable Functions (PUF). These PUFs respond to a given challenge c with a hardware-based response $r = \mathrm{PUF}(c)$ [14] which can also be used to identify and authenticate chips [15]. In the last years, different variants of PUFs have been proposed, for example based on gate delays [4], capacitative coatings [18] or initial bit configurations in asynchronous memories [5], [6]. Based on these PUF-based challenge-response protocols, more secure (but complex) authentication protocols were presented as well as methods to derive secret keys for cryptographic protection schemes.

Our Contribution. In this paper, we propose a a novel and intrinsic scheme to identify and authenticate FPGA devices that contain synchronous dual-port Block RAMs (BRAM). We exploit the fact that write collisions in BRAMs (i.e., writing with both ports different values to the same BRAM cell) show a deterministic behavior for a device-specific subset of memory bits. This can be used either to generate a fixed device identification number or, alternatively, as a PUF in a challenge-response-based authentication scenario. We also show how to employ this technique to derive a secret key on-chip as alternative source to enable configuration encryption[1]. A third contribution of this paper is the proposal to use identified memory bits with non-deterministic behavior after write collisions to build a True Random Number Generator (TRNG).

Outline. We start with a short review of existing previous work in this field. Next, we shortly explain the concept how to create write-collision with dual port BRAMs in Section III. We present our initial results obtained from experiments in Section IV. Based on our finding, we outline three scenarios for chip identification, key derivation and random number generation in Section V. Section VI discusses the aspects of validity with respect to our experiments and how to employ our proposal in practical real-world implementations, before we conclude in Section VII.

[1]For example, the feature to protect configuration files using 3DES or AES is supported by Xilinx Virtex-II Pro/Virtex-4/Virtex-5 and Altera Stratix II/III/VI FPGAs. For more information please consider Section II.

II. Previous Work

To cope with problems such as device cloning and configuration counterfeiting on FPGAs, various internal and external (board-based) approaches have been proposed. On high-end devices, IP vendors can install a secret to decrypt a configuration bit file on encryption-enabled FPGA devices using a previously inserted secret key. FPGA types like Virtex 2, Virtex 4 and Virtex 5 from Xilinx [20] as well as Altera's Stratix II-IV [2] devices provide decryption cores based on symmetric 3DES and AES blockciphers. With an encrypted configuration file, the IP can only be used on a device that has knowledge of the appropriate secret key. But here the issue of key transfer arises. One approach is to ship all FPGAs to the IP owner for on-site key installation, a smarter idea is presented in [8] based on a public-key based protocol.

However, this only works for high-end FPGAs with integrated configuration encryption feature. Less powerful devices like Xilinx Spartan-3 or Altera Cyclone FPGAs need to take other actions, such as Xilinx Device DNA [16]. Other solutions, for example by Altera, are based on separate security chips that dongle the IP to a specific FPGA by exchanging cryptographic handshake tokens between the components [1]. However, this approach is costly and rather inefficient since it requires modification to the customer's board layout and also additional hardware.

There are several academic proposals of this IP protection issue. In [9], [15], complex protocols have been proposed which require require the participation of the FPGA manufacturer (as a trusted party to distribute the IP) and the implementation of additional security features in the FPGA. The generic PUF-based approach in [15] was later picked up by Guajardo *et al.* [5], [6] and extended to preferred state analysis of SRAM-cells in uninitialized memories on Altera FPGAs. A similar, intrinsic idea for PUFs based on flip-flops (called *Butterfly-PUFs*) was presented by nearly the same group of researchers in [11], [10]. However, to intentionally induce write collisions in dual-port BRAMs as a foundation for PUFs, chip authentication protocols or any other constructive purpose was – to the best of our knowledge – not presented yet.

III. Write Collisions in Dual-Port BRAMs

In this work, we propose the constructive use of write-collisions in integrated memory blocks (BRAM) of FPGAs. In all recent FPGA devices, hardware manufacturers integrate dual ported dedicated memory blocks that can access the same portion of memory using two independent ports A and B. While reading the same memory cell with both ports (at the same time) is perfectly acceptable, writing different values to the same cell obviously results in a data collision (cf. Figure 1). After such a write collision, either the value in port A, the value of port B, or a mixture of both can be the output of this memory cell [19]. Although the inner structure of SRAM-based memory is well-known and described in many text books, the exact implementation of data arbitration by the dual-port logic is vendor-specific. We expect to find a symmetric structure inside the dual-port logic so that device-specific characteristics on the shared data input bus connecting each SRAM cell will show a preference of one or the other port value. To examine the behavior of BRAM cells during write collisions more thoroughly, we created a test setup for a series of different devices.

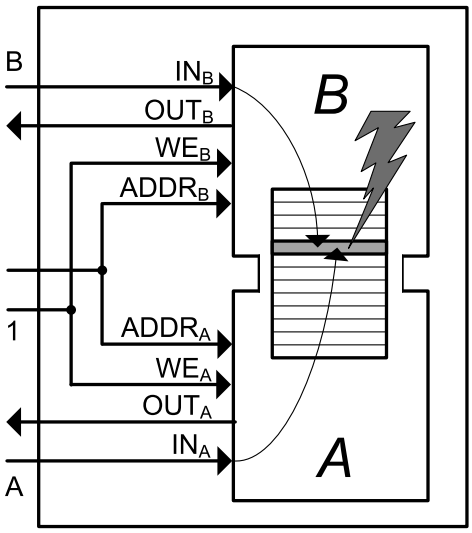

Fig. 1. Constructing a write collision in a generic dual port BRAM.

In general, we would expect two possible outcomes from our experiment:

1) The result could be biased by internal wiring or control logic so that the value at one port dominates on all devices under test. However, such a predetermined behavior would not be very helpful to identify unique characteristics of the device.
2) The outcome is influenced by chance or variances on wires and capacitances due to the production process. In this case, the individual bits of each cell may either store the value of one or the other port with a distinct probability. Exploiting these probabilities can enable several interesting application scenarios on FPGAs as more thoroughly discussed in the next sections.

IV. Experimental Setup

In several experiments, we examined the behavior of memory write collisions on Xilinx Spartan-3 XC3S200-4 devices, which are part of Digilent's Spartan-3 boards [3]. As a preliminary sample set, we tested eight different Spartan-3 boards at room temperature. Spartan-3 XC3S200 devices contain 12×18-kBit BRAMs which we configured all in a 512×32 bit configuration. We wrapped each BRAM with control logic that generates a write-collision sequentially in each 32 bit memory cell. We connected both input ports of the BRAM to 32 bit registers that can each contain either all ones (`WE` with input tied to V_{CC}) or all zeros (by enabling the reset signal `CLR`). Note that to generate a meaningful write collision, we need to set all bits on the first port and clear

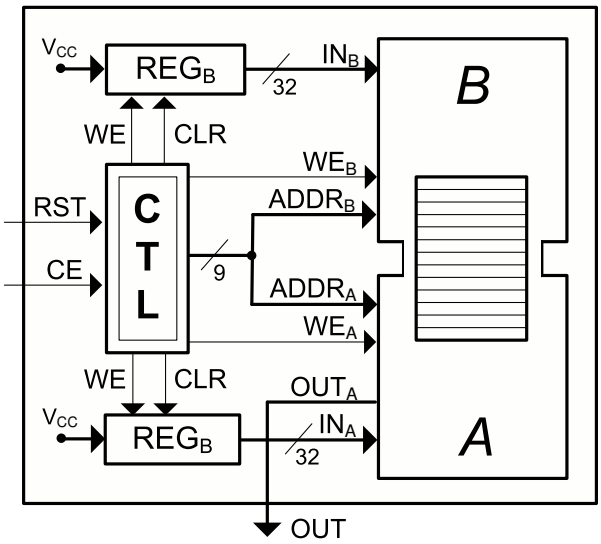

Fig. 2. Basic module to subsequently generate write-collisions in BRAMs.

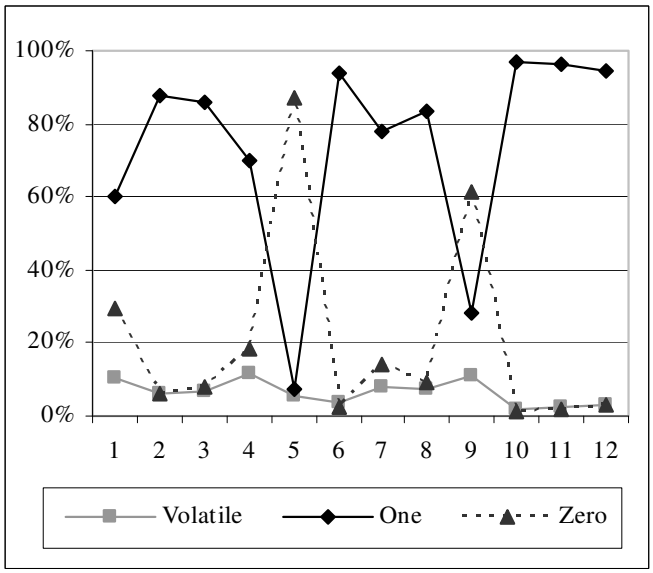

Fig. 3. Relative output results from write collisions for all 12 BRAMs of one FPGA.

all bits on the other. Then, we can decide by analyzing the output which port eventually dominates for each individual bit. Due to the flexible settings for registers REG_A and REG_B, the basic structure of our test module can be used to examine different configuration to generate collisions. For example, the memory cell can be initialized with zeros or different collisions can be created in which either port A or port B is logic high, respectively. The basic module is shown in Figure 2. We instantiated the basic module 12 times, that means for each BRAM contained in the Spartan-3 XC3S200 device. In this specific FPGA, the BRAMs are located in two columns with six BRAMs each and are denoted by corresponding `Xi` and `Yj` coordinates. For simplicity, we enumerate all BRAMs in ascending order beginning with `X0Y0` corresponding to BRAM #1 (e.g., `X1Y3` denotes the forth BRAM in the second column and is equivalent to BRAM #10). We connected all basic modules to a central controller with a connected UART transceiver. This controller subsequently queries all collision modules to return 32 bit results from collisions in each memory cell and sends the results to a PC via serial interface (115200 baud, 8N1). Additionally, the controller supports a predefined number of query iterations to allow evaluations on the persistence of collision results between different runs. In this first experiment, we chose a frequency of 100MHz, $I = 32$ iterations for each test run and generated collisions on the first $C = 256$ cells of each BRAM only[2]. In addition to the 32 iterations per run, we validated our individual results by repeating all tests three times.

Our results are promising: we encountered that after inducing write collisions the total hamming weight (e.g., the number of ones returned from the memory cells) differs for most BRAMs. In particular, after evaluation of the 32 iterations, there are many bits in each BRAM which show a rather determined behavior and flip preferably to either zero or one. On the other hand, there are also few *volatile* bits which sometimes flip to zero or one with some distinct probability. We then chose a simple model based on a three-partition threshold scheme to classify determined and volatile bits in the first place (i.e., if a bit is high in more than 24 out of 32 iterations, it is considered *one*, less than 8 iterations *zero*, and in all other cases it is denoted *volatile*). According to this model, Figure 3 shows our results for all BRAMs of a single FPGA. Due to the presence of volatile bits, our second hypothesis turns out to be the correct one since the results from our experiments seem not to be predictable. However, to be useful for applications (e.g., for chip identification), we need to verify that our results are also *distinct* among *different* FPGAs. We rerun the experiment for each of the eight different Spartan-3 boards and obtained the results depicted in Figure 4. Note that for the sake of clarity, we only provide graphs describing the hamming weight of the BRAMs contained in *four* FPGAs. The complete set of results can be found in Table I. As can be seen, all FPGAs generally show a different BRAM signature with respect to the hamming weight of all determined bits in each BRAM. We can summarize our results as follows:

- Among all FPGAs, some BRAMs show a strong bias so that all returned bits of this memory are constantly one or zero, respectively. In general, these BRAMs will not be useful in further applications and thus will not be part of further investigation.
- All tested FPGAs possess at least two BRAMs (some FPGAs even 6 or more) which revealed a preference for *both one and zero bits* of which each bit has a distinct probability. We will denote these BRAMs by the term

[2]We selected these parameters since they are convenient for transmission with the byte-wise serial interface. However, temporary setups with alternative values (i.e., $I = 100$,$C = 512$) did not show any significant divergence with respect to the results presented here.

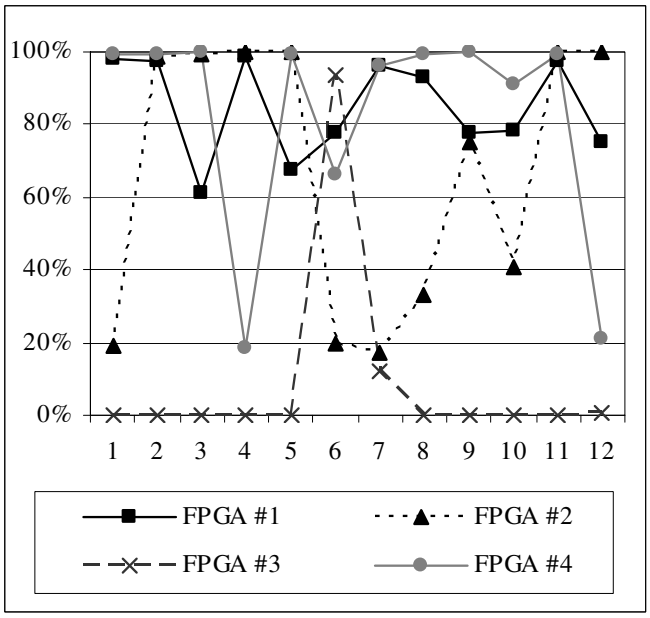

Fig. 4. Relative hamming weights of collisions in BRAMs of four different FPGAs.

Distinctive BRAM (DB) for the remainder of this paper.

- Most memory bits (usually more than 90%) of these DBs can be considered *determined* and have a strong tendency to always flip to the same bit value. This tendency to a preferred state is also persistent after FPGA reconfiguration and power-down.
- According to to our threshold scheme, up to 10% of memory bits in a DB are considered *volatile* (i.e., these bits flipped between 9 and 23 times either to one or zero). Of these bits, some still have a slight bias towards one value, and a few flip about 50% between zero and one. These good statistical properties of the latter might be useful to build a hardware-based True Random Number Generator (TRNG).
- Modifications to the collision generating module (e.g., removing the register REG_A and REG_B and tieing inputs of BRAMs to ground and V_{CC}, respectively) did have an effect on the signatures of DBs. This means, the number of determined one and zero bits differ with respect to the original implementation. We attribute this effect to the different drivers on the BRAM's input. Hence, the basic module to generate the write collisions should also remain identical in different implementations to preserve each DB's signature. In addition to that, our very basic determination scheme based on static thresholds is also contributing to this effect. In this context, a more advanced fuzzy extraction algorithm as proposed in [12] should be used for practical applications providing also a more fault-tolerant recovery of the state of each individual memory bit (however, at the cost of additional redundancy).

Based on these promising results, we can now continue to use the location and signature of DBs, for example to identify different FPGAs, to create device-specific secrets or the generate random numbers. These different scenarios are discussed in the next sections.

V. Applications and Scenarios

We now present possible scenarios how to employ the distinctive collision results of DBs in practical applications. Note that all outlined scenarios require knowledge about the location of DBs in an FPGA. Hence, DBs need to be identified first in an *enrollment* or *setup* phase. This can be done by configuring the FPGA with a configuration which examines all BRAMs (similarly to our experimental setup as seen above) and return all BRAMs signatures generated from the output of memory write collisions. Alternatively, this FPGA configuration can also contain some smarter logic to provide processed information (e.g. a computed FPGA hash or footprint) that directly can be used in subsequent applications. Further aspects concerning an efficient implementation of applications are discussed in Section VI.

A. First Scenario: Chip Identification and PUF-based Authentication

A first potential use case of the signatures obtained from write collisions is located in the field of chip identification and authentication. The distinctive location and signature (e.g., the number and position of determined ones) of DBs can be used to uniquely identify a chip (cf. Section IV). The application designer could create a specific configuration of his application that directly relies on the location and content of known DBs. Based on the location of one or several DBs, the designer can use the data acquired in the enrollment phase to identify *determined* positions of ones and zeros in the DB's signature, i.e., bit positions in the DBs which always flip either to zero or one. This information can be used to compute an *Auxiliary Identification Tag* (AIT) that contain exact positions and the corresponding preference of several determined bits (e.g., 128 or 256 bits) of one or more DBs. The AIT is encoded/encrypted using a scheme that is only known to the application developer and stored in a permanent memory, for example a dedicated location in the system flash. A corresponding decoder/decryptor for the AIT is placed inside the application's FPGA configuration. After loading the configuration at startup time, it can access the AIT from memory, decode/decrypt it and compare the designated bit positions of the AIT with the memory signature. In case of a positive match, the FPGA can be considered identified and the application can be certain to operate on a valid device.

Alternatively, instead of a startup identification procedure, a similar technique can be used as an interactive two-party authentication protocol with a *verifier* (who wants to check the authenticity of the device) and a *prover* (i.e., the FPGA device). In this context, we consider the distinct location and value of each bit as a PUF which can be queried using a challenge-response protocol. The verifier has access to all

TABLE I
RELATIVE NUMBER OF ONE (1), VOLATILE (V) AND ZERO (0) BITS OBTAINED FROM WRITE COLLISIONS IN ALL BRAMS ON EIGHT DIFFERENT XC3S200 FPGAS.

BRAM	**FPGA Device**															
	#1	*(1/v/0)*	#2	*(1/v/0)*	#3	*(1/v/0)*	#4	*(1/v/0)*	#5	*(1/v/0)*	#6	*(1/v/0)*	#7	*(1/v/0)*	#8	*(1/v/0)*
#1	0.98/0.01/0.01		0.19/0.08/0.73		0.00/0.00/1.00		0.99/0.01/0.00		0.60/0.10/0.30		0.00/0.00/1.00		0.95/0.03/0.02		0.25/0.09/0.66	
#2	0.97/0.02/0.01		0.99/0.00/0.01		0.00/0.00/1.00		1.00/0.00/0.00		0.88/0.06/0.06		0.82/0.08/0.10		0.73/0.10/0.17		1.00/0.00/0.00	
#3	0.61/0.11/0.28		0.99/0.01/0.00		0.00/0.00/1.00		1.00/0.00/0.00		0.86/0.06/0.08		1.00/0.00/0.00		0.94/0.03/0.03		0.27/0.07/0.66	
#4	0.98/0.01/0.01		1.00/0.00/0.00		0.00/0.00/1.00		0.18/0.06/0.76		0.70/0.12/0.18		0.81/0.08/0.11		0.13/0.05/0.82		0.90/0.05/0.05	
#5	0.68/0.11/0.21		1.00/0.00/0.00		0.00/0.00/1.00		0.99/0.01/0.00		0.08/0.05/0.87		0.30/0.10/0.60		0.20/0.09/0.71		0.93/0.04/0.03	
#6	0.78/0.08/0.14		0.19/0.08/0.73		0.93/0.04/0.03		0.66/0.11/0.23		0.94/0.03/0.03		1.00/0.00/0.00		1.00/0.00/0.00		0.65/0.12/0.23	
#7	0.96/0.03/0.01		0.17/0.05/0.78		0.12/0.10/0.78		0.96/0.02/0.02		0.78/0.08/0.14		1.00/0.00/0.00		0.27/0.06/0.67		1.00/0.00/0.00	
#8	0.93/0.04/0.03		0.33/0.11/0.56		0.00/0.00/1.00		0.99/0.01/0.00		0.83/0.08/0.09		0.74/0.09/0.17		0.94/0.03/0.03		0.27/0.09/0.64	
#9	0.78/0.09/0.13		0.75/0.10/0.15		0.00/0.00/1.00		1.00/0.00/0.00		0.28/0.11/0.61		0.95/0.02/0.03		0.95/0.02/0.03		0.65/0.12/0.23	
#10	0.78/0.08/0.14		0.41/0.09/0.50		0.00/0.00/1.00		0.91/0.05/0.04		0.97/0.02/0.01		0.88/0.05/0.07		0.15/0.05/0.80		0.23/0.07/0.70	
#11	0.97/0.02/0.01		1.00/0.00/0.00		0.00/0.00/1.00		0.99/0.01/0.00		0.96/0.02/0.02		0.98/0.01/0.01		0.99/0.01/0.00		0.97/0.02/0.01	
#12	0.75/0.10/0.15		1.00/0.00/0.00		0.01/0.01/0.98		0.21/0.10/0.69		0.94/0.03/0.03		1.00/0.00/0.00		0.91/0.05/0.04		0.95/0.03/0.02	

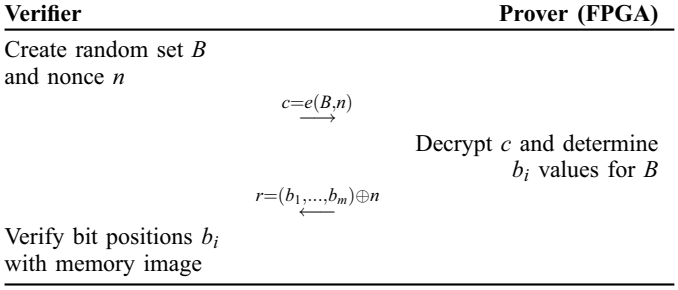

Fig. 5. Challenge-response protocol for device authentication.

information obtained during the enrollment phase, hence he knows the memory signatures of each FPGA device. The verifier can use this information to request the FPGA to arbitrarily return the values at some determined bit positions for comparison to verify if the device is indeed authentic.

Instead of storing an AIT with bit positions and corresponding bit values in memory, the verifier sends an encrypted challenge consisting of an $m-bit$ nonce (random number to avoid replay-attacks) and an arbitrary, freshly selected set B of m bit positions of determined bits to the FPGA device. The FPGA decrypts the challenge, determines the values of the bit positions in the corresponding DBs (i.e., querying the PUF with $(b_1,\ldots,b_m) = PUF(B)$), masks the results with the nonce (e.g., using XOR) and returns the response to the verifier. In case the response is correct, the FPGA proved to be authentic. For better visualization, the full protocol is shown in Figure 5.

B. Second Scenario: Secret Key Derivation

Secure storage of secret keys in volatile FPGA devices is still an issue and an active field of research. As mentioned in the introduction, there is the secret key store in high-end FPGA devices which supports encrypted configurations. Xilinx uses battery-buffered memory to permanently store the secret key for the decryptor on chip which comes at the cost of a limited product lifetime. Altera prefers a permanent fuse-based solution for the secret key which is one-time programmable and limits the reusability of the device after programming. Similar to proposals in [5], we can introduce an initial phase (configuring the FPGA with a startup configuration first, before loading the actual encrypted application) to install a secret key K in a RAM-based key store. This startup configuration on the FPGA reads an *Auxiliary Bit Position* (ABP) information from permanent Flash memory which specify m positions of determined bits of one or several DBs (where $m \geq |K|$). Again, the ABP is stored in a specially encoded/encrypted way only known to the application designer[3]. The startup configuration then determines the bit values in the DBs according to the ABP information and assembles a corresponding secret key which is finally stored in the (volatile) key store. This key can then used in a second step by the internal decryptor to load the actual, encrypted application configuration.

Despite the use for providing key material for configuration protection, the same procedure can also be applied to generate secrets directly within the application logic. Instead of storing a static secret inside the application's configuration file, the application can also use (externally or internally stored) ABP information to generate such a secret directly out of DBs. These secrets are highly individual for each FPGAs and also implicitly bind the confidential data uniquely to each device (cf. TPM functionality [17]).

C. Third Scenario: True Random Number Generation

A further idea to exploit write-collision on FPGA could target random number generation. In this case, we first need the enrollment phase for an FPGA to identify potential DBs and the positions of determined and volatile bits. This time, however, we focus our attention on volatile bits of which we will use those with appropriate statistical properties (e.g.,

[3]The startup configuration and ABP layout does not need to be necessarily created by application designers but could also be globally deployed by the FPGA manufacturer. The global startup configuration can come as an encrypted configuration file which is encrypted with a key that is statically built in FPGA devices by the FPGA manufacturer. However, such a generic configuration must be personalized in the startup phase for each application designer (cf. [8]).

return about 50% ones and zeros on average after long sample runs). In the 512x32 BRAM configuration of our experiment, we encountered that even two or more bits of a single 32 bit output are likely to show these desired characteristics. By generating repetitivly write collisions in such a cell and extracting these two (or more) volatile bits from the returned 32 bit output, we can accumulate 'random' bits in a cyclic shift register. To generate a write collision takes two cycles (one cycle resets the cell and another performs the actual collision), so that we could yield a random number generator with a theoretical throughput of 100 MBit/s (based on a single BRAM running at 100 MHz). Note that before using this source of randomness for practical applications, it is mandatory to ensure that the entropy satisfy security standards (i.e., passes the NIST recommendation for random number generation [13]) and is also sufficiently resistant against physical manipulations like voltage or temperature fluctuations. However, this extensive evaluation is not in the scope of this work.

VI. Evaluation and Implementation

As shown in Section IV, our initial results to exploit write collisions in synchronous BRAMs were promising and raised ideas for different application scenarios. However, we like to stress that our first results need further validation with respect to the following aspects:

- *Size of the sample set*: Our experiments were based on the small sample set of only eight Spartan-3 XC3S200 devices. Experiments should be repeated for larger and different test sets (i.e., more and different FPGA types) to validate our result according the uniqueness of DB signatures.
- *Iteration count*: In a trade-off of accuracy and serial transmission speed, we opted to perform 32 collision iterations on each memory bit. Although this number is perfectly acceptable to provide an idea about a BRAM's write collision footprint, a more thorough analysis with hundreds or thousands of iterations is possibly more suitable for practical applications.
- *Operational environment*: We did not yet perform measurements under different operational conditions and environments, e.g., different temperatures or under influence of radiation sources. However, according to [5], it was reported that a changing environment in the temperature range of $-20°$C to $80°$C has only an influence of 12% on the volatility (based on the hamming weight) of all memory cells compared to a reference measurement obtained at $20°$C. Hence, assuming a similar result, we expect the use of write-collisions to be feasible also in more challenging operational environments.

Another important aspect is the resource consumption which is required to allow the generation of write collision with a BRAM. Note that BRAMs are a valuable resource in most designs and thus should be only exclusively used for collision generation when they are abundantly available. Since this does not hold for most applications, we propose an alternative implementation in which the write collision logic is designed transparently. In other words, after the chip identification or secret value generation has been finished (cf. Section V), the BRAM can be re-used by application logic again. This alternative design is applicable for all memory blocks which do not (completely) contain ROM values that are preloaded during device configuration. Obviously, any of these ROM values would be overwritten during generation of write collisions. However, if only parts of the BRAM are used for ROM values, the remaining memory cells might be still available as target for the collision generating circuit. The alternative implementation based on multiplexed inputs (and transparent control logic) to the BRAM is shown in Figure 6.

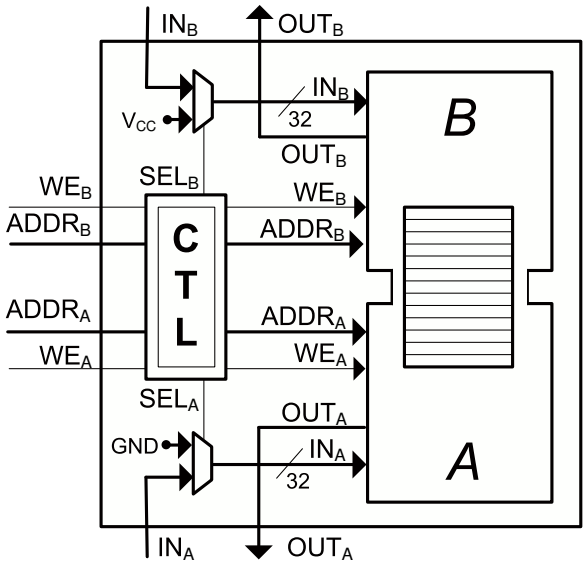

Fig. 6. Alternative implementation for a transparent collision generation.

As a last issue, we now discuss how the application logic for the different scenarios should be implemented. Since potentially every BRAM can also be a DB, a first naïve approach could be to place the transparent collision logic adjacent to *each* BRAM. However, despite the high logic overhead on chips which contain several dozens or even hundreds of BRAMs, all the results have to been aggregated and controlled in a central place on the device. This introduces additional overhead and can even lead to long signal routes with a negative impact of the system performance. Hence, it is preferable to pick only a subset of BRAMs at a time which should augmented with the transparent collision logic. Then, in a next step, different configuration files of the same application can be created that contain individual (distinct) subsets of augmented BRAMs. For example, for one FPGA we would use a first configuration that contains collision logic at, say, BRAM 1,3 and 6, and for another FPGA a second configuration with collision logic at BRAM 2,4 and 5.

VII. Conclusions and Outlook

In this paper, we examined write collisions in synchronous dual-port BRAMs of recent FPGA devices and conducted several experiments using eight Spartan-3 XC3S200 FPGAs. Due to our promising results, we proposed three different application scenarios for chip identification and authentication, secret key derivation and random number generation. We like to emphasize at this point that our results still need validation with respect to larger device sets and varying operational environments, e.g., temperature and voltage fluctuations. Hence, we are looking forward to further work extending and implementing our ideas for new security application based on memory write-collisions.

References

[1] Altera Corporation. FPGA design security using MAX II reference design. URL http://www.altera.com/end-markets/refdesigns/sys-sol/indust_mil/ref-des-secur.html.

[2] Altera Corporation. Stratix II/III/IV FPGAs, 2009. URL www.altera.com/products/devices/.

[3] Digilent Inc. Spartan-3 Board, populated with an XC3S200 FPGA, 2009. http://www.digilentinc.com/Products/Detail.cfm?Prod=S3BOARD.

[4] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *9th ACM Conference on Computer and Communications Security*, pages 148–160. ACM New York, NY, USA, 2002.

[5] J. Guajardo, S. Kumar, G. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In *Cryptograpic Hardware and Embedded Systems (CHES 2007)*, volume 4727, page 63. Springer, 2007.

[6] J. Guajardo, S. Kumar, G. Schrijen, and P. Tuyls. Physical Unclonable Functions and public-key crypto for FPGA IP Protection. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 189–195, 2007.

[7] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.

[8] T. Güneysu, B. Möller, and C. Paar. Dynamic Intellectual Property Protection for Reconfigurable Devices. In *IEEE International Conference on Field-Programmable Technology (ICFPT 2007)*, pages 169–176. IEEE Computer Society, 2007.

[9] T. Kean. Cryptographic Rights Management of FPGA Intellectual Property Cores. In *10th international Symposium on Field-Programmable Gate Arrays (FPGA 2002)*, Monterey, CA, 2002.

[10] S. Kumar, J. Guajardo, R. Maes, G. Schrijen, and P. Tuyls. Extended abstract: The butterfly PUF protecting IP on every FPGA. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2008)*, pages 67–70, 2008.

[11] R. Maes, P. Tuyls, and I. Verbauwhede. Intrinsic PUFs from Flip-flops on Reconfigurable Devices. In *3rd Benelux Workshop on Information and System Security (WISSec 2008)*, 2008.

[12] R. Maes, P. Tuyls, and I. Verbauwhede. Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *Cryptographic Hardware and Embedded Systems (CHES 2009)*, volume 5747 of *LNCS*, pages 332–347. Springer-Verlag, 2009.

[13] NIST. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication SP 800-90, 2007.

[14] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[15] E. Simpson and P. Schaumont. Offline hardware/software authentication for reconfigurable platforms. In *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249, pages 311–323, 2006.

[16] M. Smerdon. *Security Solutions Using Spartan-3 Generation FPGAs*. Xilinx Inc., April 2008. http://www.xilinx.com/support/documentation/white_papers/wp266.pdf.

[17] Trusted Computing Group (TCG). TPM specification, version 1.2 revision 94, March 2006. URL www.trustedcomputinggroup.org/specs/TPM/.

[18] P. Tuyls, G. Schrijen, B. Skoric, J. van Geloven, N. Verhaegh, and R. Wolters. Read-proof hardware from protective coatings. volume 4249, page 369. Springer, 2006.

[19] Xilinx Application Note. *XAPP463 – Using Block RAM in Spartan-3 Generation FPGAs*, March 2005. http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf.

[20] Xilinx Corporation. Virtex2, Virtex 4 and Virtex 5 FPGAs, 2006. URL www.xilinx.com/products/silicon_solutions/.