Noname manuscript No.
(will be inserted by the editor)

How to Generate Repeatable Keys Using Physical Unclonable Functions

Correcting PUF Errors with Iteratively Broadening and Prioritized Search

Nathan E. Price · Alan T. Sherman

Received: date / Accepted: date

Abstract We present an algorithm for repeatably generating keys using entropy from a Physical Unclonable Function (PUF). PUFs are logically identical physical constructs with Challenge-Response Pairs (CRPs) unique to each device. Applications include initialization of server keys and encryption of FPGA configuration bitstreams. One problem with PUFs is response errors. Our algorithm corrects PUF errors that inhibit key repeatability.

Our approach uses a PUF to generate an error-free PUF value in three steps. First, we repeatedly sample the PUF to determine the most likely value. Second, we apply an iteratively-broadening search to search up to some number of bit errors (in our experiments we use two). Third, we apply exhaustive search until the correct value is found or failure is declared. The searches are prioritized by the known bit error rates in decreasing magnitude. We assume the application includes a test for the correct value (e.g., some matching plaintext-ciphertext pairs).

Previous algorithms often omit noisy PUF bits or use error-correcting codes and helper data. Our algorithm can use all PUF bits regardless of noise. Our approach is simple, and for appropriate parameter choices, fast. Unlike previous approaches using error-correcting codes, when used for public-key cryptography our method requires storing only the public key and no other helper data in non-volatile storage.

We implemented a latch-based PUF on FPGAs and measured PUF characteristics to analyze the effectiveness of the algorithm. Tests for a 1024-bit PUF show 351 samples reduce the probability of errors to less than 10^{-6}. The iterative broadening and exhaustive searches further reduce failure rates.

Keywords Cryptography · cryptographic key generation · physical unclonable function (PUF) · entropy · error correction · FPGA

N. Price
E-mail: np1@umbc.edu

A.T. Sherman
E-mail: sherman@umbc.edu

Cyber Defense Lab
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21784

1 Introduction

In 2012, Heninger et al. [1] showed substantial operational flaws resulting from weak keys generated with poor entropy. Hardware devices used insufficient entropy and generated the same keys. Even worse, some devices start with low but increasing entropy such that the first prime number (P) used for RSA key generation is shared and the second (Q) is not, exposing both P and Q.

Physical Unclonable Functions (PUFs) provide a convenient mitigation to some of the vulnerabilities described by Heninger. When used to key devices that require repeatable keys, however, PUFs must overcome the difficulty that sometimes their outputs are non-deterministic. This paper presents a new approach for correcting such PUF errors. We present a simple algorithm with low runtime and minimal additional costs external to the IC containing a cryptographic module that implements the algorithm. Our approach is an alternative to well-known algorithms such as excluding noisy bits or correcting their errors with BCH codes. Our approach works for any application of PUFs that requires repeatable outputs, including repeatable key generation and device identification.

PUFs implement a set of challenge-response pairs (CRPs) such that separate logically-identical devices produce different CRPs, unpredictable even to the designer before sampling the PUFs [2][3][4][5]. When used for repeatable key generation, one would typically take the PUF response from a particular challenge to seed a deterministic cryptographic pseudorandom number generator to produce a sequence of cryptographic keys. Our focus is reliably producing this PUF response.

PUF value generation begins with repeated cycling, generating the most likely value of bits based on average bit values. Repeated sampling creates the PUF value for the initial key generation and the starting candidate PUF value for subsequent PUF value reproductions. We call this starting value the “correct” value. A test determines if the correct PUF value is reproduced. If the most likely PUF value has errors, they are corrected using an iterative broadening search.

For one choice of parameters, the algorithm first assumes a single error and tests possible single bit corrections in order based on bit noise. If the single error assumption fails, the search expands by assuming two errors. If these steps fail, a prioritized exhaustive search corrects the initial PUF value errors. Bit noise prioritizes the exhaustive search by testing corrections to noisiest bits first. After finding the correct value or reaching a predefined point of failure, the prioritized exhaustive search stops.

Contributions of this work include:

- Applying iteratively broadening and exhaustive search to correct PUF errors while prioritizing the search by bit noise, and
- Demonstrating and quantifying PUF error rates using latch-based PUFs implemented on FPGAs.

In comparison with existing approaches based on error-correcting codes and excluding noisy bits, our approach for correcting PUF errors is simple, fast, robust, storage-space efficient, and does not require excluding noisy bits. Our approach is more robust by not restricting error correction to some fixed maximum number of errors specified at initialization. In our approach, the helper data stored in non-volatile memory can be simply a public key, which for many public-key applications requires no extra storage, reducing costs. By contrast, approaches based on error correcting codes require additional specialized helper data. The running times of both approaches depend on implementation parameters; for our FPGA implementation, our approach achieves excellent error rates very quickly (with 351 samples, the probability of error is less than 10^{-6}).

2 Physical Unclonable Functions (PUFs)

Physical Unclonable Functions (PUFs) use uncontrollable variations in the fabrication process to provide unclonable challenge-response functionality [2]. A PUF accepts a challenge (a bit sequence) and produces a response (another bit sequence) unique to that PUF. Two logically identical PUFs possess, with high probability, different Challenge-Response Pairs (CRPs). Applications of PUFs include unique identification and cryptographic key generation (repeatable and non-repeatable).

A PUF is a physical device providing functionality physically infeasible to duplicate by the original manufacturer or by others with or without design secrecy [6]. Such circuits produce responses dependent on physical characteristics and uncontrollable variations in the manufacturing process of integrated circuits. A specific example is doping variations of transistors in integrated circuits [2]. PUFs include delay-based ring oscillators, arbiter PUFs and memory-based SRAM and Latch PUFs [4]. Latch circuits with cross-coupled NAND or NOR gates form the PUFs examined here.

Two PUFs of the same design will, with high probability, produce different CRPs. For a given challenge, each PUF produces a response that is unpredictable before querying the PUF. Two basic properties desired in CRPs of PUFs used for identification are robustness and unpredictability [4]. A robust PUF requires unchanging CRPs over time, producing the same response every time the corresponding challenge is queried. This repeatability enables repeatable cryptographic key generation with the PUF. However, noise often causes unstable responses causing erroneous bit values within the response. Fuzzy extractors overcome the noise to reproduce values [2]. Noise causes a repeatedly queried challenge to produce different responses [7]. Robustness is not desired for random number generation.

Beyond response robustness, Guajardo [3] specifies three assumptions regarding PUF behavior:

1. The response to a challenge reveals no significant information about a response to any other challenge.
2. Without control of the PUF, a response to a challenge can only be guessed (with negligible probability).
3. PUFs are tamper evident. Attempts to analyze a PUF’s unique characteristics are assumed to alter the PUF’s functionality (CRPs).

3 Problem Statement

We present error correction for PUF noise to enable using PUF values for repeatable cryptographic key generation.

3.1 Test for Valid Key

Testing a PUF value for correctness varies by application. Options include plaintext-ciphertext samples stored in non-volatile memory (locally or remotely) to test the key generated by a PUF value. For public-key cryptography, assuming the public key is published or saved locally, the local system generates private and public key pairs with the PUF and compares the public keys. If a trusted public repository for public keys is available, additional stored data are not required for key reproduction.

3.2 PUF-Based Repeatable Key Applications

An example application for repeatable PUF generated keys is encryption of FPGA bitstreams (see Figure 1). FPGAs are often implemented with SRAM for configuration storage. When turned off, the volatile SRAM will lose the FPGA's configuration. Every power-up requires a configuration bitstream transfer from non-volatile memory to the FPGA. Since the configuration bitstream defines the device's functionality, copying the bitstream enables device cloning using another FPGA [2][8].

Encrypting the configuration protects against cloning. The FPGA internally decrypts the bitstream and uses it for configuration. The FPGA must have a key to decrypt the bitstream. Options for FPGA designers include non-volatile memory, battery backed volatile memory and intrinsic PUFs [2][8]. Developers using production FPGAs must use resources selected by FPGA designers, which may exclude non-volatile memory. Without internal non-volatile memory, an FPGA is unable to store the key if powered off, regardless of how the key was originally created or supplied. Example applications include devices with SRAM-based FPGAs, such as routers, televisions and set top boxes [8].

3.3 Advantages Over Externally Supplied Keys

Keys generated outside the FPGA have numerous vulnerabilities. The entity that generates the key must be trusted. Externally generated keys introduce risks, such as stored, shared or stolen keys.

The proposed PUF-based key generation occurs inside the FPGA and, if the FPGA is properly designed, never leaves the FPGA [9][2]. A module inside the FPGA performs decryption without revealing the key as shown in Figure 1.

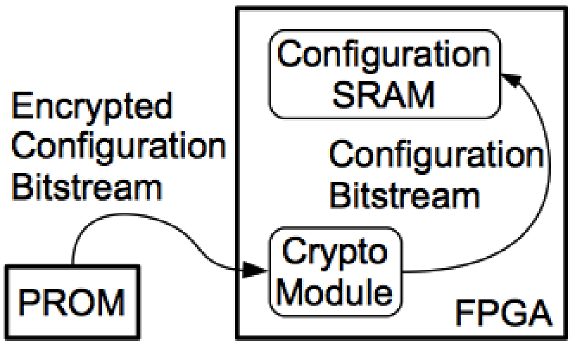

Fig. 1 The PROM stores the encrypted configuration bitstream. The FPGA decrypts the bitstream and uses the unencrypted bitstream to configure the SRAM-Based FPGA.

3.4 Adversarial Model and Trust Assumptions

The adversary is attempting to acquire the cryptographic key. An attacker can access hardware designs, transmission lines and component I/O pins. This includes ability to read memory devices through I/O pins. We assume the attacker is unable to observe internal component data and signals not supplied to I/O pins. We assume invasive attempts to characterize PUFs modify PUF functionality [2][3].

If the FPGA allows loading new configurations [2][8], the attacker can take over control of the FPGA by reconfiguring it. In practice, one must also consider the integrity of FPGA configuration bitstreams and helper data. We focus on protecting the secrecy of the key.

We assume proper design and fabrication of the FPGA, which includes a dedicated cryptographic module. The module acts as a black box capable of accepting commands and data for encryption or decryption and outputting the processed data. The cryptographic module contains the PUF and never outputs the secret key.

Trust assumptions include absence of trojan, backdoor or other malicious hardware in the FPGA. Adversaries can read helper data. We do not consider side channel attacks, such as monitoring power usage, or attacks that modify helper data.

4 Previous Work

We now discuss selected prior work on bit errors and previous algorithms to correct these errors for repeatable key generation. This paper is based on Price's Master's Thesis [10].

4.1 Previous Work by Others

Böhm [9] describes techniques proposed by Microvision and LSI to detect noisy bits with repeated sampling. Omitting these noisy bits from key generation enables repeatability. LSI further uses fuses to identify the noisy bits. However, excluding some bits reduces the number of PUF bits supplying entropy.

Böhm [9] describes stabilizing PUF bits with special circuit design features.

Böhm [9] and Guajardo [2] describe advantages of internally held keys (e.g., a key created and contained in an FPGA).

Much work applies fuzzy extractors and error correcting code to correct errors. For example, binary BCH code requires extra PUF bits and non-volatile storage of helper data specific to PUF response error correction, increasing costs [9][2][3].

Published test results show PUFs provide strong sources of device uniqueness and entropy for cryptographic key generation [9][2][11].

4.2 Previous Key Generation Algorithms

Numerous algorithms exist to correct PUF response errors and enable cryptographic key generation. Fuzzy extractors perform both error correction and privacy amplification (applying a hash function) [2]. Generally called fuzzy extractors, the algorithms produce the correct value from a noisy starting value. We present two examples below.

Error Correction with BCH Code. During initial generation, binary BCH code generates and stores helper data in non-volatile memory [2]. The helper data do not reveal sufficient information for practical PUF value reproduction without the PUF. The fuzzy extractor corrects errors by applying the BCH code and helper data to the noisy PUF response [2]. Implementation specific details determine the maximum number of correctable errors.

Multiple Sampling and Noisy Bit Omission. Böhm [9] describes PUF-based identification by repeatedly sampling a PUF. ID reproductions detect and omit unstable PUF bits. Omitting some bits requires the PUF to consist of more bits than the reproduced ID. Proposals for identifying unstable bits over time include fuses marking unstable bits. As discussed later, a limitation to this approach is that PUFs often exhibit aging effects that alter PUF bit characteristics causing bit stability changes over time [2].

5 PUF-Based Repeatable Key Generation

We propose repeatable key generation by correcting PUF errors using repeated sampling, iteratively broadening search and prioritized search. Individual samples require cycling the PUF. Repeated sampling determines the initial PUF value for both initial key creation and subsequent key recreations.

Repeated sampling during initialization determines the value treated as the PUF's correct value. Errors during the initialization stage must be minimized and warrant using more samples than subsequent PUF value recreations [11]. The initial key generation creates and stores required helper data externally in non-volatile memory.

Recreating the key begins by producing the most probable PUF value. Repeated PUF sampling is the primary strategy for error correction. If errors exist, we apply two additional error correction stages:

1. Iteratively broadening search (for up to a limited number (e.g., two) errors)
2. Prioritized exhaustive search.

Each search prioritizes by decreasing bit error rates.

Section 8 presents motivation for these two stages, and specifically, for separating single and double bit errors from the prioritized exhaustive search. Using bit noise rankings improves the search stages. For N samples, we count the number of times a bit is one. Bits with counts near 0 or N are the most stable. Bits with counts near $N/2$ are the noisiest. We try correcting the noisiest bits first.

Algorithm 1 implements prioritized exhaustive search. The counter bits indicate which PUF bits change during the exhaustive search. Without prioritization, a simple bitwise XOR flips the bits. To prioritize the search, $noiseRank[i]$ selects the noisiest bit in the PUF, which is XORed with counter bit i.

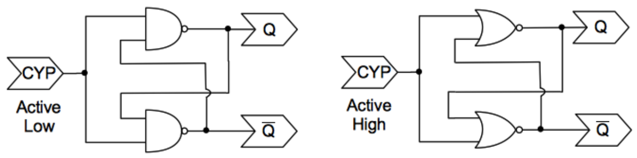

Fig. 2 Schematics for NAND and NOR latch-based PUFs. Q is the output value of the PUF bit.

Algorithm 1 Pseudocode for prioritized exhaustive search algorithm (not optimized to exclude previously tested candidates).

```
Inputs: (most likely PUF value), noiseRank[], iterationLimit
Output: candidatePUFvalue

for count = 1 to iterationLimit do
    candidatePUFvalue = (most likely PUF value)
    for i = 0 to (PUFsize - 1) do
        candidatePUFvalue[noiseRank[i]] =
            mostLikely[noiseRank[i]] xor count[i]
    end for
    if candidatePUFvalue passes test then
        return candidatePUFvalue
    end if
end for

return candidatePUFvalue
```

6 Latch PUF Implementation

We now describe the PUFs we implemented and characterize them in terms of bit distribution, stability and uniqueness.

6.1 Technical Details of the Latch PUFs

Our PUFs consist of two NAND gates (or two NOR gates) shown in Figure 2 and explained by Su [11]. Connecting one input from each gate provides a control line to cycle the PUF. Cycling the PUF (CYP) is the act of forcing the circuit into a stable state with an applied input, removing the input such that the circuit is in an unstable state, the PUF transitioning to an unpredictable stable state. The unpredictable stable state exhibits the unique PUF characteristics.

We implemented both NAND and NOR latch PUFs with 1024 bits per PUF. Implementation used Verilog with the Xilinx ISE Design Suite and four Avnet Xilinx Spartan-6 LX9 microboards. An advantage of PUFs is their simplicity, and as such, our designs are straightforward. We chose to implement PUFs with FPGAs (versus ASICs) for simplicity.

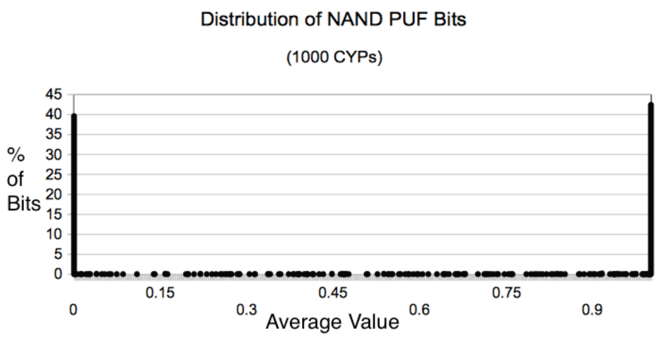

Fig. 3 Average bit values for the implemented NAND latch PUF.

6.2 PUF Characterization

The PUF bit distribution test performed 1000 CYPs, recording the value of each bit for each CYP.

Twenty tests over approximately 2.5 weeks tracked short term changes in PUF bit stability and distribution. Four FPGAs configured with the same PUF designs tested device uniqueness. Two tests of each device, with each test performing 100 CYPs, demonstrated device uniqueness and stability. Hamming distances quantify device uniqueness.

Bit value distribution varied for different PUF implementations. The NAND PUF is nearly equally distributed between 0 and 1 bits (see Figure 3). The NOR PUF (not shown) is biased towards 0 with roughly a 3:1 ratio. We do not know why this bias exists, but we suspect underlying circuitry implementing the logic causes the bias. Approximately 80% of PUF bits in both PUFs produced the same value over 1000 CYPs (i.e., no noise).

The stability experiment consisted of 20 runs of 1000 CYPs over roughly 2.5 weeks. PUF bit stabilities showed little change. The majority of bits were consistent over all CYPs. Unstable bits are bits not observed as always 0 or always 1. As seen in Figure 4, standard deviations for non-constant bits are relatively low.

Configuring four FPGAs with identical PUF bitstreams tested device uniqueness. Determining uniqueness and stability of each implementation consisted of two tests with 100 CYPs per test. Hamming distances between tests quantify device uniqueness. The NAND PUF exhibited a Hamming Distance of approximately 11% – 12% between devices and less than 3.5% between tests of the same device. Table 1 provides the exact values. As with PUF bit distribution, different PUF implementations produced different results not detailed here.

When using a PUF for random number or cryptographic key generation it would be useful to know the entropy of its responses. Unfortunately, there is no attractive way for us to do so for our PUFs. It is possible,

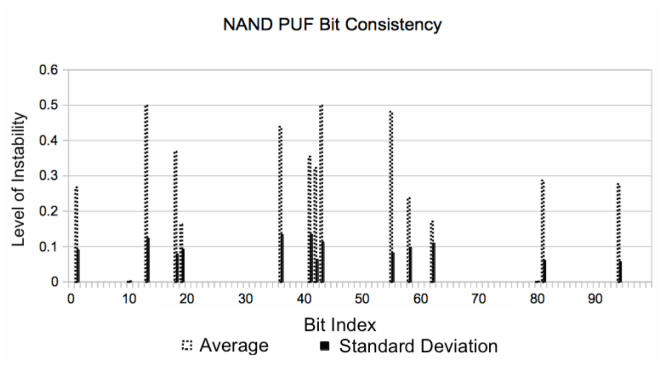

Fig. 4 The NAND PUF stability test performed 100 CYPs on 20 occasions. For each test, we calculated the average value and, for clarity, adjusted the average as $(0.5-|Average-0.5|)$. This figure displays results for the first 100 PUF bits.

Table 1 Hamming Distances between four FPGAs configured with the same NAND PUF bitstream. The Test numbers indicate the device number and the Test letter distinguishes between the two tests of each device.

Test	1a	1b	2a	2b	3a	3b	4a	4b
1a	0	13	125	118	116	119	114	113
1b	13	0	132	121	121	124	117	116
2a	125	132	0	33	119	118	109	114
2b	118	121	33	0	112	113	114	113
3a	116	121	119	112	0	5	112	121
3b	119	124	118	113	5	0	109	116
4a	114	117	109	114	112	109	0	13
4b	113	116	114	113	121	116	13	0

however, to calculate for each PUF the entropy of the noise in its responses. For one challenge, we did so using 200 samples for each bit in each PUF. We found that the single-PUF noise entropy for each of our four PUFs was 113, 116, 117, and 112 bits, respectively. By contrast, an ideal PUF would have zero noise entropy.

In principle one could estimate response entropy for a PUF in two ways. First, one could build many identical PUFs and, for a given challenge, sample the bit responses across the many PUFs. We did so across our four PUFs using 200 samples per bit per PUF. We found that the resulting response entropy across our four PUFs was approximately 239 bits (by contrast, an ideal PUF would have 1024 bits of entropy). Second, although we did not do so, one could consider many possible PUF instantiations of a given PUF circuit, using many different circuit layout locations within an FPGA.

7 Analysis of PUF and Algorithm

Sampled values indicate probabilities of bit errors. For example, a bit producing 1 for 80% of samples has a value of 1 and a 20% error rate. Similarly, a bit pro-

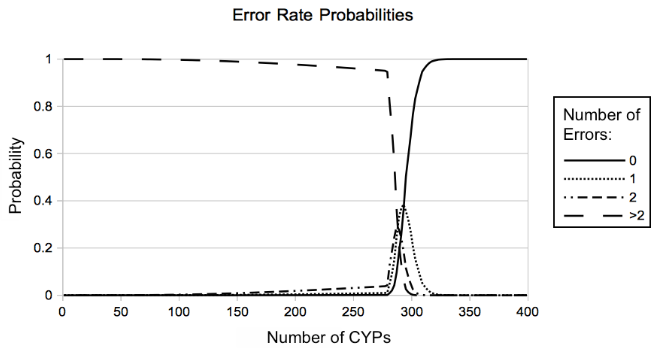

Fig. 5 Probabilities of various numbers of errors as functions of the number of samples.

ducing 0 for 80% of samples has a correct value of 0 and a 20% chance of an erroneous 1. Error probabilities greater than 50% are not possible unless performance characteristics of the PUF change (see Section 8.3).

Sampling the 1024-bit PUF 20,000 times and calculating average values for each bit computed error probabilities for individual bits. Figure 5 shows the probabilities of 0, 1, 2 and more than 2 errors in the 1024-bit PUF response calculated as functions of the number of CYP samples.

Figure 5 shows the probability of 0 errors approaching 100% as the number of CYP samples increases. With 325 CYP samples, the probability of even a single error is very small. The finite number of samples and numerical precision of the calculations result in approximately 0% probability of any errors with at least 353 samples used to regenerate the PUF response. Errors are still possible, but improbable. If repeated sampling does not reduce error rates to acceptable levels, the number of samples and the operation of the PUF should be scrutinized, as external factors can induce PUF noise [2][4][11]. Section 8 provides more explanation.

PUF value validation tests candidate values using saved helper data. If the initial PUF value is wrong, the single error correction step proceeds. For an N-bit PUF, N possible values exist with a single error. PUF bit noise prioritizes the single bit correction tests. For the two-bit error assumption, $N(N-1)/2$ possible values exist. Noise rank prioritizes the search order. Beyond two errors, a predefined limit bounds the number of tested values.

To demonstrate the algorithm's limits, let the most stable error bit be the n^{th} noisiest bit. For simplicity, Algorithm 1 makes no attempt to avoid retesting values with one or two errors. Therefore, the upper bounds for Algorithm 1 is $2^n - 1$. Testing all $2^{1024} - 1$ possibilities is not practical, so a point of failure is defined by the number of values tested or a predefined timeout.

For the tested PUF, most bits of the PUF exhibited complete stability. For analysis purposes, we assume these bits are correct. The one- and two-error correction algorithms presented above do not make this assumption, which may be included as an optimization. Only 202 possible one-bit errors remain for this PUF instance. For two-bit errors, $202(201)/2 = 20301$ possibilities exist. If these algorithms fail, Algorithm 1 is required. Practical computation limits the number of iterations.

Even if only 202 bits exhibit noise, an exhaustive search of $2^{202} - 1$ values is still not practical. A better option may be reducing errors by taking more samples. Sampling a CYP is fast. Even 1000 CYPs are practical and can reduce error correction work. As discussed, Figure 5 indicates the probability of at least one error approaches 0 with at least 353 CYP samples for our PUF. Our data are for our FPGA-based PUFs and are not necessarily representative of other PUFs.

8 Discussion

We now discuss our algorithms and experiments.

8.1 Algorithm Performance

The experiments and analysis in Sections 6 and 7 show repeated sampling prevents most errors. The exact number of samples depends on PUF characteristics.

The choice of how many errors to search for depends on the PUF. For our PUFs, we chose up to two. For the 1024-bit example, exhaustively testing all one and two error possibilities is practical.

The assumptions of one or two errors, made based on Figure 5, address scenarios with an error bit that appears relatively stable. If one of these assumptions is correct, at most $N(N+1)/2$ iterations are required instead of the worst case scenario of $2^{N-1} + 2^{N-2}$ iterations when exactly two errors exist and have the lowest priority in the exhaustive search. The prioritized exhaustive search provides a limited safety net.

To compare with other techniques, Guajardo [2] uses BCH code to generate a 128-bit key with approximately a 10^{-6} failure rate. The test results in Figure 5 show 351 samples lower the failure rate to less than 10^{-6} for the 1024-bit PUF. Accordingly, repeated sampling prevents most errors, and the iteratively broadening and exhaustive searches function as backups to repeated sampling.

The algorithm can perform the sampling fast. We did not optimize the PUF, but an optimized PUF can be sampled once every clock cycle. Therefore, an optimized PUF with performance characteristics similar to the implemented PUF can perform the 351 samples with 351 clock cycles. At 100 Mhz, sampling takes 3.51 microseconds to produce a PUF value with a failure rate less than 10^{-6}. Heninger [1] reports flawed attempts for seed generation occurring approximately 4 seconds after boot. Accordingly, the proposed algorithm reproduces the seed much earlier with a failure rate less than 10^{-6}.

Prioritized exhaustive searching lacks a guarantee of absolute error correction due to practical limitations. Failure to recreate a key is similar to failure of non-volatile memory storing a key and can be handled similarly.

Though absent from the presented design, excluding especially noisy bits [9] can further reduce failure rates. Including extra PUF bits prevent a bit shortage if several bits exhibit unacceptable noise [9][2]. Tracking which bits to exclude, and handling bit aging, must be considered. If a simple threshold during sampling identifies noisy bits, changes in bit noise must be considered. Causes of noise change include temperature and aging [2][11].

8.2 PUF Implementation

For our implementations, PUF bits are more evenly distributed for the NAND design than for the NOR design (see Section 6.2). Most bits showed complete stability over the test period. Unstable bits showed relatively low standard deviations, indicating the rate that a particular value is produced remained consistent. For example, if a bit produces 1 for 75% of all CYPs on one test, the results of other tests also produced a 1 for approximately 75% of CYPs.

One challenge we faced was the FPGA-based PUFs implemented for this project exhibit non-ideal uniqueness characteristics: The fractional Hamming Distances[1] between two of our devices was approximately 0.11 to 0.12, while 0.5 is ideal [2][11]. Multiple PUF implementations using the same basic latch-based PUF bits exhibited different characteristics. Without detailed knowledge of individual latch implementations, an explanation can only be guessed based on levels of implementation detail and corresponding PUF characteristics reported by others [4][11].

Other reported fractional Hamming Distances for latch-based PUFs vary from being notably biased [4] to near the ideal 0.5 [11]. The PUFs implemented for this project use FPGAs with IC details not known to us. We do not know if the FPGA circuitry maximizes PUF functionality. For comparison, Katzenbeisser [4] imple-

[1] Fractional Hamming Distance is the Hamming Distance divided by the length of the compared bit strings.

mented latch PUFs with ASICs using standard cells and states that IC details were not available. However, Su [11] created ASICs with knowledge of the IC layouts and reported more ideal results (e.g., a near 0.5 fractional Hamming Distance between devices). Latch-based PUF characteristics vary significantly between implementations. The FPGA implementations created here provide demonstrations but are not necessarily representative of other FPGA implementations.

8.3 Environment and Aging

Two significant concerns outside the scope of this project are environmental and long-term aging effects on PUFs [2][4][11]. Such effects increase error rates that may require additional error correction activity. Both remain as open problems that must be investigated before adopting the presented design and algorithm.

8.4 Key Replacement

A variety of situations require key replacement. Multiple options exist for key replacement. Implementing multiple sets of PUF bits and progressing through the sets as needed generates new keys.

If aging causes the PUF bit stabilities to change, refreshing the PUF value by repeating the initialization process deals with changing bits, but generating a new key with a seed varying only slightly from the previous seed (perhaps 10 of 1024 bits are flipped) may introduce cryptographic vulnerabilities.

8.5 Open Problems

A well-known optimization for FPGAs is to examine many possible layout locations of the circuit [12][13]. It would be interesting to determine how such layout positions affect bit error rates and device uniqueness.

It would also be interesting to compare the performance of FPGA and ASIC PUFs.

9 Conclusion

We presented and experimentally analyzed a new, simple and low-cost approach for recreating PUF generated keys that does not require exclusion of noisy bits. We tested the algorithm's effectiveness using latch-based PUFs. Repeated sampling reduces error probabilities significantly. Any existing errors can eventually be corrected by the prioritized search methods given enough time. Practical limits prompt specification of a point of failure with a low probability of being reached. For public-key cryptography, helper data for our algorithm consists of only the public key, which is likely already stored and will not add to device cost.

PUFs offer a simple and fast approach to generating entropy. Our work helps engineers to generate repeatable cryptographic keys.

Acknowledgements We thank Dhananjay Phatak, Chintan Patel, Russell Fink and Christopher Nguyen for helpful comments. Price and Sherman were supported in part by the National Science Foundation under SFS grant 1241576.

References

1. Heninger, Nadia, Zakir Durumeric, Eric Wustrow and J. Alex Halderman. "Mining your Ps and Qs: Detection of widespread weak keys in network devices." Proceedings of the 21st USENIX Security Symposium. Vol. 2. 2012.
2. Guajardo, Jorge, Sandeep S. Kumar, Geert Jan Schrijen and Pim Tuyls. FPGA Intrinsic PUFs and Their Use for IP Protection. In CHES (Cryptographic Hardware and Embedded Systems), 2007 pages 63–80.
3. Guajardo, Jorge, Sandeep S. Kumar, Geert Jan Schrijen and Pim Tuyls. Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection. Field Programmable Logic and Applications, 2007. FPL 2007 pages 189–195.
4. Katzenbeisser, Stefan, Ünal Kocabaş, Vladimir Rožić, Ahmed-Reza Sadeghi, Ingrid Verbauwhede and Christian Wachsmann. "PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon." Extended version of paper originally published at CHES 2012. Online http://eprint.iacr.org/2012/557.pdf.
5. Suh, G. Edward and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In Design Automation Conference 2007, ACM 978-1-59593-627-1/07/0006. 2007.
6. van Dijk, Marten and Ulrich Ruhrmair. Physical Unclonable Functions in Cryptographic Protocols: Security Proofs and Impossibility Results. Cryptology ePrint Archive, Report 2012/228.
7. O'Donnell, Charles W., G. Edward Suh and Srinivas Devadas. PUF-Based Random Number Generation. In MIT CSAIL CSG Technical Memo 481, 2004.
8. Maxfield, Clive. The Design Warrior's Guide to FPGAs: Devices, Tools and Flows. Burlington, Massachusetts: Elsevier, 2004.
9. Böhm, Christopher and Maximilian Hofer, *Physical Unclonable Functions in Theory and Practice.* Springer, 2013.
10. Price, Nathan. "How to Generate Repeatable Keys Using Physical Unclonable Functions: Correcting PUF Errors With Iteratively Broadening and Prioritized Search." Master's Thesis. Department of Computer Science and Electrical Engineering. University of Maryland, Baltimore County. April 2014.
11. Su, Ying, Jeremy Holleman, and Brian P. Otis. "A digital 1.6 pJ/bit chip identification circuit using process variations." Solid-State Circuits, IEEE Journal of Volume 43 Issue 1 (2008): 69–77.

12. Maes, Roel, Pim Tuyls, and Ingrid Verbauwhede. "Intrinsic PUFs from flip-flops on reconfigurable devices." 3rd Benelux workshop on information and system security (WISSec 2008). Vol. 17. 2008.
13. Maiti, Abhranil, et al. "Physical unclonable function and true random number generator: a compact and scalable implementation." Proceedings of the 19th ACM Great Lakes symposium on VLSI. ACM, 2009.