Increasing the Efficiency of Syndrome Coding for PUFs with Helper Data Compression

Matthias Hiller and Georg Sigl
Institute for Security in Information Technology
Technische Universität München, Munich, Germany
{matthias.hiller, sigl}@tum.de

Abstract—Physical Unclonable Functions (PUFs) provide secure cryptographic keys for resource constrained embedded systems without secure storage. A PUF measures internal manufacturing variations to create a unique, but noisy secret inside a device. Syndrome coding schemes create and store helper data about the structure of a specific PUF to correct errors within subsequent PUF measurements and generate a reliable key.

This helper data can contain redundancy. We analyze existing schemes and show that data compression can be applied to decrease the size of the helper data of existing implementations. We introduce compressed Differential Sequence Coding (DSC), which is the most efficient syndrome coding scheme known to date for a popular reference scenario.

Adding helper data compression to the DSC algorithm leads to an overall decrease of 68% in helper data size compared to other algorithms in a reference scenario. This is achieved without increasing the number of PUF bits and a minimal increase in logic size.

Index Terms—Physical Unclonable Functions (PUFs), Syndrome Coding, Data Compression, Differential Sequence Coding (DSC), Fuzzy Extractor, Run-Length Encoding (RLE), FPGA.

I. INTRODUCTION

Secure cryptographic keys for low prices are a prerequisite to protect simple embedded systems from theft of sensitive data and intellectual property. Silicon Physical Unclonable Functions (PUFs) enable secret key generation in standard CMOS technology for devices without secure storage. Similar to reading out a biometric pattern, PUFs generate a unique and relatively stable secret during runtime inside a chip. A PUF evaluates manufacturing variations within the circuit to create a unique signal within every device. A PUF output measures the physical properties inside the device and maps the measurement result into a PUF response, which is typically a digital signal. Since physical properties depend on environmental conditions and measurements always contain noise, PUF responses of the same device are highly correlated, but not fully identical.

Therefore, syndrome coding schemes such as Code-Offset Syndrome Coding [1], Index-Based Syndrome Coding (IBS) [2] or Differential Sequence Coding (DSC) [3] enable error correction for PUFs to increase the reliability of the output key. A syndrome coding scheme creates helper data from a PUF response and an external secret. The helper data permits to restore either the initial PUF response or a stable secret that can be directly used as key. The helper data must not leak any substantial information about the resulting key so that it can be placed in unsecured external storage. In this work, we will present a concept to improve the efficiency of syndrome coding schemes.

Typically, syndrome coding is concatenated with error-correcting codes (ECCs) to further reduce the output bit error probability. We focus on the helper data representation in this work, so ECC properties are not addressed in detail. A fuzzy extractor is a special case of syndrome coding, and optionally ECC. Instead of storing and reproducing a user-defined key, a fuzzy extractor generates a device-dependent key that cannot be chosen arbitrarily.

 Syndrome coding schemes usually use non-uniform helper data. This motivated us to investigate for the first time how data compression increases the efficiency of syndrome coding schemes. Information Theory offers a full toolbox of powerful data compression algorithms for different purposes [4]. If the elements in a sequence are not independent or the letters of the alphabet of the elements are not distributed uniformly, a data compression, or more general, a source coding algorithm can be applied to remove redundancy in the given sequence to represent the sequence by a shorter sequence. A decoder restores the initial sequence.

Our contributions: We show that helper data compression can be applied to several syndrome coding schemes. We use DSC, a very recent syndrome coding scheme, to give a sample implementation of helper data compression and discuss the efficiency. Applying Run-Length Encoding (RLE) [5], a data compression algorithm, to the DSC helper data reduces its size significantly with low implementation overhead. We study different parameter sets to provide an efficient solution for a real world scenario.

The FPGA implementation demonstrates the efficiency in practice. Our implementation uses 32% of the helper data of the best published non-DSC work. The hardware effort to decompress the helper data adds only 4% more slices.

Outline: Section II introduces related work. Section III demonstrates that helper data compression can be widely applied. We give a summary over DSC, a recent approach to store helper data, in Section IV. In Section V, RLE is presented. The efficiency of RLE is addressed in Section VI and the yield is analyzed in Section VII. We present and evaluate our FPGA implementation in Section VIII.
II. RELATED WORK

Quantitative comparisons of ASICs with popular PUFs are presented in [6]. A comprehensive introduction to the topic can be found in [7].

Code-Offset Syndrome Coding by Dodis et al. [1] XORs the PUF response with a random codeword of an ECC and stores the result as helper data. We will compare our results to the code-offset implementations of Bösch et al. [8], as well as Maes et al. [9]. Other implementations were presented by Maes et al. [10] and van der Leest et al. [11].

Index-Based Syndrome Coding (IBS) by Yu and Devadas [2] uses pointers for information hiding that refer to PUF outputs with specific properties. Complementary IBS by Hiller et al. [12] applied an iterative IBS encoding several times on the same set of PUF outputs to increase the reliability. In [13], Yu et al. compared previous work with focus on security and reliability.

In this work, we will design our implementation to fulfill the requirements defined by Guajardo et al. [14] with the PUF output distribution by Maes et al. [15]. A 128 bit key with key error probability $< 10^{-9}$ is derived from an SRAM PUF with average bit error probability 15%. During helper data generation, the SRAM cells are read out several times to estimate the reliability. During key reproduction the SRAM is read out once. Therefore, our results can be directly compared to the reference implementations in [9], [12], and partly [8].

III. MOTIVATION

Several implementations of syndrome coding schemes contain different parts of helper data: (nearly) uniform parts and also non-uniform parts. For PUFs with a high uniqueness, we can assume that the code-offset bits in [8] and [9], the IBS pointers in [2] and [12], and the inversion bits in DSC [3] as uniform and independent. Therefore, they cannot be compressed.

In contrast, e.g. the reliability information in [9] and [12] depends on the non-uniform reliability distribution of the PUF outputs. Therefore, the size can be reduced with data compression algorithms. DSC distance pointers are not uniformly distributed either. In [9], 9 helper data bits contain the code-offset and 8 bit reliability information. Every helper data block in [12] combines a 4 bit pointer and 5 bit reliability information. The distributions of the reliability information can be found in [15], [16]. For DSC, every code-sequence bit is mapped to a 7 bit distance value and one inversion bit.

We can calculate the entropy of the helper data easily since all distributions are known. The results in Table I show that existing fuzzy extractor implementations have relatively large amounts of non-uniform helper data that could be compressed. We are aiming for a low hardware complexity so the compression algorithm has to be chosen carefully according to distribution.

For the DSC implementation, the non-uniform part of the helper data has an entropy of 858 bit but occupies 1,904 bit of storage. In theory, this part could be compressed by 55%. In this work, we will show that we can achieve a compression of 49% of the non-uniform part in practice with a low hardware overhead. In the end, this leads to an overall decrease of 43% of the entire helper data. This shows that the theoretical idea of helper data compression can lead to profound benefits in practice.

IV. DIFFERENTIAL SEQUENCE CODING

Differential Sequence Coding (DSC) is a new syndrome coding scheme, introduced in [3]. It ensures that no unreliable PUF outputs will contribute to the key generation process. The reliability of specific PUF outputs varies due to the stochastic manufacturing process. We are able to predict the statistical properties of a long sequence of PUF outputs but it is hard to predict the properties of a small block. Instead of splitting a PUF response into fixed blocks, DSC assigns a variable number of PUF outputs to every code sequence bit.

In a two stage scenario, the helper data generation contains an ECC encoder and a DSC encoder. We concatenate a DSC encoder and a convolutional code [17]. A random input sequence is encoded to a code sequence $c^k$ with length $k$ in the convolutional encoder. The DSC algorithm encodes one code sequence bit $c_i (i \in \{1, ..., k\})$ in every iteration and executes the same loop multiple times. DSC indexes only reliable PUF outputs with an error probability smaller than a given maximum bit error probability $p_{max}$ with $0 < p_{max} < 0.5$.

The DSC encoder searches the PUF outputs $R^n$ sequentially and the offset counter $o$ keeps track of the absolute position of the last indexed PUF output. In iteration $i$, the DSC encoder searches the PUF outputs $R^n$ for the first PUF output $R_i$ and $j \in \mathbb{N}$ that fulfills

$$\Pr[r_{j+o} = 1] \geq 1 - p_{max} \lor \Pr[r_{j+o} = 0] \geq 1 - p_{max}$$

(1)

The helper data $v_i$ contains a distance pointer $u_i$ that points from the last indexed PUF output to the current indexed PUF output and an inversion bit $v_i$. The inversion bit marks if the expected PUF response and the code sequence bit are identical or inverse.

$$v_i = \begin{cases} 0 & \text{if } \Pr[r_{j+o} = c_i] \geq 1 - p_{max} \\ 1 & \text{else} \end{cases}$$

(2)

Figure 1 shows a small example for DSC encoding. White boxes represent a zero and black boxes represent a one. A PUF output $R$ with $\Pr[r = 0] \geq 1 - p_{max}$ is also denoted with a

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Helper Data</td>
<td>1,536</td>
<td>4,096</td>
<td>272</td>
</tr>
<tr>
<td>Non-Uniform Helper Data</td>
<td>12,228</td>
<td>5,120</td>
<td>1,904</td>
</tr>
<tr>
<td>Entropy of the Non-Uniform Helper Data</td>
<td>10,929</td>
<td>2,959</td>
<td>858</td>
</tr>
<tr>
<td>Max Compression of Non-Uniform Helper Data</td>
<td>11%</td>
<td>42%</td>
<td>55%</td>
</tr>
<tr>
<td>Max Overall Compression</td>
<td>9.5%</td>
<td>23%</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table I: Helper data structure and maximum compression rates in previous fuzzy extractor implementations.
white box. Accordingly, a black box indicates \( \Pr[r = 1] \geq 1 - p_{\text{max}} \) and gray boxed stand for unreliable PUF outputs with \( p_{\text{max}} < \Pr[r = 1] < 1 - p_{\text{max}} \).

The code sequence \( c^4 = (0, 1, 1, 0) \) and the PUF outputs \( R^{16} \) are encoded to four helper data blocks \( w^k = (u, v)^4 \). \( R_3 \) is the first reliable PUF output. We count the unreliable PUF outputs between two reliable ones as distance pointer, so \( u_1 = 1 \). For the first inversion bit, \( v_1 = 0 \) since both boxes have the same color. \( R_3 \) is the next reliable PUF output, so \( u_2 = 0 \) and again \( v_2 = 0 \). After skipping three unreliable PUF outputs, \( R_6 \) is indexed by \( u_3 = 3 \). Since a white box is indexed for a black code bit, \( v_3 = 1 \). \( u_4 \) and \( v_4 \) are computed accordingly, such that \( u^3 = (1, 0, 3, 4) \) and \( u^4 = (0, 1, 1, 1) \).

As a generalization of the example, the encoding algorithm presented in Algorithm 1 searches \( n \) PUF outputs \( R^n \) with PUF responses \( r^n \) for PUF outputs fulfilling Equation 1.

The helper data sequence \( w^k \) contains the syndrome. Every element \( w_i = (u_i, v_i), i = 1, \ldots, k, \) contains the distance \( u_i \in \{0, \ldots, 2^l - 1\} \) for a fixed \( l \in \mathbb{N} \), and an inversion bit \( v_i \in \{0, 1\} \).

In Algorithm 1, there are two possible sources of error, for which we decide that the PUF is not usable and discard the chip: \( \text{error}_1 \): the number of PUF bits meeting the required error probability smaller than \( p_{\text{max}} \) in the entire PUF is too small (with error probability \( \epsilon_1 \)) or \( \text{error}_2 \): the distance between any indexed PUF bits exceeds the maximum \( l \) bit counter value (with error probability \( \epsilon_2 \)).

For PUF outputs \( R^n \) with cumulative distribution function (cdf) over their expectations, the probability \( p \) that a PUF output is indexed is given by

\[
p = \text{cdf}(p_{\text{max}}) + 1 - \text{cdf}(1 - p_{\text{max}})
\]

Since \( u \) corresponds to the number of unsuccessful trials to find a good PUF output, the offset pointers \( u^k \) are distributed according to a geometric distribution \( P_p \) with parameter \( p \), such that

\[
P_p(u) = p(1-p)^u
\]

We evaluated different parameter sets of DSC and convolutional codes to find an efficient combination that fulfills the required key error probability of \( 10^{-6} \) in [15] for a 128 bit key using an SRAM PUF with average bit error probability of 15\%. Using \( p = 0.264 \) and a \((2, 1, [7])\) convolutional code [17] led to the best results. In this case, \( \epsilon_1 + \epsilon_2 < 10^{-3} \) for \( n/k = 4.5 \). Therefore, we will focus on an efficient representation for \( p = 0.264 \) in the following. See [3] for a detailed reliability and security analysis of DSC.

By default, the same storage size is assigned to all helper data elements \( u_i \) such that every element can store the highest specified pointer that only occurs with a small probability. So, the maximum pointer defines the size for all elements. However, almost all elements store smaller pointers that could be represented in a smaller memory.

In this work, we will tackle this open problem by applying a data compression algorithm that assigns variable size representations to the index pointers.

V. RUN-LENGTH ENCODING

Golomb’s Run-Length Encoding (RLE) [5] is a source coding algorithm designed for sequences with geometric probability distribution. An improved version was presented by Gallager and van Voorhis [18] in 1975.

The helper data distribution given in Equation 4 is geometric, so RLE can be applied to our problem. Note that [5] treats runs of successful draws ended by an unsuccessful one and in our case, the successful draw ends a run.

RLE represents any integer number \( u \) by a series of ones followed by a small number of a finite alphabet \( A \) with elements \( a_j \in A, j = 0, \ldots, m - 1 \) and \( |A| = m \). For the run-length part, \( m \) determines how many unsuccessful trials are represented by every 1 and \( a \) gives the number in the
remaining \( u \mod m \) trials. As straight-forward approach, the binary representation of \( j \) is chosen as \( a_j \).

The algorithm can be interpreted as Euclidean division of \( u \) by \( m \) with different representations of the quotient and remainder. The quotient is represented in a series of ones, followed by a zero, and the remainder in the finite alphabet \( \mathcal{A} \). Therefore, the compressed representation \( q(u) \) of \( u \) is given by

\[
q(u) = 1 \left\lfloor \frac{u}{m} \right\rfloor 0a_{(u \mod m)} \quad (5)
\]

According to [18], optimal codes can be constructed for an integer \( m \) chosen in dependency of \( p \in [0,1] \) such that

\[
(1-p)^m + (1-p)^{m+1} \leq 1 < (1-p)^{m-1} + (1-p)^m \quad (6)
\]

As an example, Table II shows RLE representations of small integers for \( m = 1, m = 2 \) and \( m = 4 \). For different parameters \( m \), the size of the fixed length part and the overall length \( l \) show large variations. Therefore, the selection of \( m \) will be addressed later in this work.

<table>
<thead>
<tr>
<th>integer</th>
<th>uncoded</th>
<th>( m = 1 )</th>
<th>( m = 2 )</th>
<th>( m = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>10</td>
<td>01</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>110</td>
<td>100</td>
<td>0100</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>111</td>
<td>101</td>
<td>0111</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>1111</td>
<td>1100</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>11111</td>
<td>1110</td>
<td>11001</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>111111</td>
<td>11100</td>
<td>11001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1111111</td>
<td>11110</td>
<td>11100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>11111111</td>
<td>111100</td>
<td>111000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>111111111</td>
<td>111110</td>
<td>111100</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>1111111111</td>
<td>1111110</td>
<td>1111100</td>
</tr>
</tbody>
</table>

Table II: Run-length encoding with \( m = 1, m = 2 \) and \( m = 4 \) according to [5]

The length \( l(u) \), i.e. the number of bits to represent \( u \) is given by the individual length of the run-length part and the fixed length of the finite alphabet.

\[
l(u) = \left\lfloor \frac{u}{m} \right\rfloor + 1 + \log_2 m \quad (7)
\]

Therefore, we try to minimize the expectation \( E(l) \), where \( l(u) \) is distributed with the geometric probability distribution \( P_p(u) \) given in Equation 4.

VI. HELPER DATA COMPRESSION

Goal of this section is to identify a suitable parameter set to increase the efficiency of DSC with helper data compression. The pointer lengths in RLE differ depending on the input and the code parameters. This section shows compressions for different parameter sets to identify the optimal parameter for our problem. For increasing \( p \) in geometric distributions, higher integer numbers \( u \) are selected less likely, so the average amount of information per symbol decreases 1.

In Figure 2, \( p \) is plotted on a logarithmic x-axis and the average length is shown on the linear y-axis. The entropy (solid cyan line) is the lower bound for every lossless compression, so we try to approach this limit as closely as possible. The other plots show the average pointer sizes for different RLE parameters.

For low \( m \), the fixed part is rather small whereas the RLE part increases rapidly with increasing source entropy. In contrast, high \( m \) have a large fixed part and only slowly increasing RLE parts.

For \( p = 0.264 \), \( m = 2 \) approaches the entropy closest. With an average pointer size of 3.18, the encoded representation is only 0.03 bit higher as the entropy.

VII. YIELD ANALYSIS

The last section demonstrated that in average at least 3.18 bit have to be assigned to each pointer and also the inversion bit has to be stored. For an embedded system, we have to assign a specific amount of memory for the helper data pointers to generate one single key.

We aim for a yield greater 99.9%, thus tolerating helper data overflow errors with a probability of \( \epsilon_2 \leq 5 \cdot 10^{-4} \).

Figure 3 shows the empirical helper data size distributions \( S_p(l) \) with RLE parameter \( m \) for helper data size \( l \) and \( 10^7 \) simulated samples. The distribution for \( m = 2 \) has the lowest mean value.

Figure 4 demonstrates that the size of the helper data can be reduced significantly compared to the 2,176 bits of

1For very low \( p \), the expression \( (1-p)^u \) only decreases very slowly with increasing \( u \). Therefore, many \( u \) have similar probabilities which results in a high entropy. For high \( p \), small \( u \) are chosen with a high probability and \( (1-p)^u \) decreases much faster. This leads to a low entropy.
the uncompressed version without reducing the yield. For uncompressed helper data, dedicated storage is assigned to every pointer, so no overflow can be tolerated for any pointer.

Here, several small pointers can compensate one extremely large one. The compensation permits to assign a smaller storage overhead, because now only a large number of large pointers can cause an overflow instead of one single large pointer.

At least 1150 bits should be assigned for a reasonable yield. However, $\epsilon_2(l)$ decreases rapidly around 1200 helper data bits. The error probability decreases by several orders of magnitude for spending 5% to 10% more helper data bits. As a result, $\epsilon_2(l) \leq 5 \cdot 10^{-4}$ can be achieved by $l = 1224$.

VIII. IMPLEMENTATION

The DSC hardware implementation with helper data compression is similar to a DSC implementation without helper data compression with major modifications in the DSC decoder.

The new DSCmp Dec module provides the joint functionality of the basic DSC decoder and the RLE decoder. The new module permits to remove the large counters in the DSC module. Through this synergy, the number of registers decreased and also required logic decreased slightly. A small serial parallel converter (SE PA) is added to the helper data path to convert the serially incoming helper data into chunks of 8 for the Spong Ent hash function.

Table III compares our new hardware implementation with previous work. The exact DSC cycle counts depend on the pointers in a specific device. To get a pessimistic estimate, both DSC cycle counts were obtained by setting half of the distance pointers to 4 and the other half to 5 so that all PUF response bits are read in. The DSC helper data is compressed by 43% with a minimal hardware overhead. Thus, our new DSC fuzzy extractor with helper data compression requires only 9% to 32% of the helper data of the comparable non-DSC approaches.

Loading in the data bitwise instead of bytewise increases the overhead for the communication at the input side. This leads to the 10% increase in execution time.

However, the implementation is still highly efficient. Our compressed DSC approach leads to the lowest PUF and helper data sizes, which are very important measures. The reference implementation by Bösch et al. [8] requires about $3 \times$ more PUF outputs, helper data and slices with a slight benefit in execution time. The fast soft decision implementations by Maes et al. [9] and Hiller et al. [12] require about one third of the execution time and are about 10% smaller. However, they require more PUF outputs and more substantial, $6.5 \times$ and over $11 \times$ more helper data.

Tables IV presents detailed synthesis results for a Xilinx Spartan-3E xc3s1200e-5fg320 legacy FPGA to provide numbers that are comparable to the reference implementations. The Spartan-6 (xc6slx45-3fgg484) implementation in Table V shows the complexity on an entry level state-of-the-art FPGA and can be used as reference point for future implementations.

Designing DSC in simple loops for low complexity and choosing a very specialized data compression algorithm pay
Table III: FPGA implementations of reproduction procedures of the DSC and reference implementations synthesized for Xilinx Spartan 3E FPGAs

<table>
<thead>
<tr>
<th></th>
<th>PUF Output Bits</th>
<th>Helper Data Bits</th>
<th>Slices</th>
<th>Block RAM Bits</th>
<th>Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code-Offset Golay [8]</td>
<td>3,696</td>
<td>3,824</td>
<td>≥ 900</td>
<td>0</td>
<td>&gt; 24,024</td>
</tr>
<tr>
<td>C-IBS RM-GMC [12]</td>
<td>2,304</td>
<td>9,216</td>
<td>250</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>Compressed DSC Conv. Code</td>
<td>1,224</td>
<td>1,224</td>
<td>272</td>
<td>11,264</td>
<td>33,925</td>
</tr>
</tbody>
</table>

Table IV: Synthesis results of the DSC reproduction procedure implementation for Xilinx Spartan-3E FPGA

<table>
<thead>
<tr>
<th></th>
<th>DSCcmpr Dec</th>
<th>Viterbi Dec</th>
<th>SPONGENT</th>
<th>XOR Mod</th>
<th>Entire Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices Total</td>
<td>17</td>
<td>91</td>
<td>28</td>
<td>61</td>
<td>272</td>
</tr>
<tr>
<td>Registers</td>
<td>9</td>
<td>32</td>
<td>109</td>
<td>43</td>
<td>219</td>
</tr>
<tr>
<td>Logic LUTs</td>
<td>26</td>
<td>164</td>
<td>146</td>
<td>109</td>
<td>140</td>
</tr>
<tr>
<td>Block RAM Bits</td>
<td>–</td>
<td>11,264</td>
<td>–</td>
<td>–</td>
<td>11,264</td>
</tr>
</tbody>
</table>

Table V: Synthesis results of the DSC reproduction procedure implementation for Xilinx Spartan-6 FPGA

<table>
<thead>
<tr>
<th></th>
<th>DSCcmpr Dec</th>
<th>Viterbi Dec</th>
<th>SPONGENT</th>
<th>XOR Mod</th>
<th>Entire Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices Total</td>
<td>7</td>
<td>50</td>
<td>15</td>
<td>24</td>
<td>92</td>
</tr>
<tr>
<td>Registers</td>
<td>9</td>
<td>32</td>
<td>21</td>
<td>36</td>
<td>121</td>
</tr>
<tr>
<td>Logic LUTs</td>
<td>18</td>
<td>115</td>
<td>34</td>
<td>67</td>
<td>237</td>
</tr>
<tr>
<td>SB Block Rams</td>
<td>–</td>
<td>4</td>
<td>11</td>
<td>–</td>
<td>15</td>
</tr>
</tbody>
</table>

IX. CONCLUSIONS

Helper data compression is a new concept to increase the efficiency of popular syndrome coding schemes. For the example of an improved coding scheme called compressed DSC, we have shown that the helper data size can be reduced with RLE data compression. We change the fixed storage assignment for each helper data element to the size that is actually required for every single element.

In the benchmark SRAM scenario, RLE decreases the size of a single helper data element from 8 to in average 4.18. Our compressed DSC implementation is the most efficient FPGA implementation for this scenario known to date. The overall helper data size was lowered by 43% compared to the uncompressed DSC helper data. Further, our implementation requires 32% of the helper data of the best comparable non-DSC candidate.

Acknowledgements: This work was partly funded by the German Federal Ministry of Education and Research in the project SIBASE through grant number 01S13020A.

REFERENCES