Hardware SLE Solvers: Efficient Building Blocks for Cryptographic and Cryptanalytic Applications

Andy Rupp [a] Thomas Eisenbarth [b] Andrey Bogdanov [c]
Oliver Grieb [d]

[a] *Dept. of Computer Science, University of Trier, Germany.*
[b] *Dept. of Mathematical Sciences, Florida Atlantic University, Boca Raton, USA.*
[c] *ESAT/COSIC and IBBT, Katholieke Universiteit Leuven, Belgium.*
[d] *Horst Görtz Insitute for IT Security, Ruhr-University Bochum, Germany.*

Abstract

Solving systems of linear equations (SLEs) is a very common computational problem appearing in numerous research disciplines and in particular in the context of cryptographic and cryptanalytic algorithms. In this work, we present highly efficient hardware architectures for solving (small and medium-sized) systems of linear equations over $\mathbb{F}_{2^k}$. These architectures feature linear or quadratic running times with quadratic space complexities in the size of an SLE, and can be clocked at high frequencies. Among the most promising architectures are one-dimensional and two-dimensional systolic arrays which we call triangular systolic and linear systolic arrays. All designs have been fully implemented for different sizes of SLEs and concrete FPGA implementation results are given. Furthermore, we provide a clear comparison of the presented SLE solvers. The significance of these designs is demonstrated by the fact that they are used in the recent literature as building blocks of efficient architectures for attacking block and stream ciphers [5,17] and for developing cores for multivariate signature schemes [2,6].

Key words: Cryptanalytic Hardware, Cryptographic Hardware, Linear Equations, Gauss-Jordan Elimination, SLE Solver, SMITH, GSMITH, Systolic Array.

1 Introduction

From a complexity-theoretical point of view, the solution of an SLE is efficiently computable. For instance, using Gaussian elimination any SLE can be solved in at most cubic time. However, software implementations of current

algorithms for solving SLEs may turn out insufficient in practice, especially for applications where a real-time response is desirable or many tasks have to be processed in parallel.

Based on the parallel nature of hardware, we propose a family of high-speed hardware algorithms for Gaussian elimination. We focus on dense SLEs over binary finite fields arising in symmetric cryptanalysis and in asymmetric cryptography, where fast algorithms from numerical analysis computing approximative solutions and well-elaborated algorithms from asymmetric cryptanalysis for solving sparse systems are either not applicable or become inefficient.

Though this paper mainly studies algorithms for solving SLEs, the three fundamental problems of linear algebra (matrix multiplication, matrix inversion, finding solutions to SLEs) are computationally equivalent: Once there is an efficient algorithm for one of these problems, any of the others can be solved with essentially the same complexity.

There are numerous applications of linear algebra in nearly any area of computer science. Cryptology is no exception with matrix problems arising both in cryptanalysis and cryptography. In this introductory part, we briefly outline the areas where our hardware algorithms are of most relevance.

1.1 Motivation: Small and Medium-Sized Dense SLEs in Cryptology

Medium-sized to large dense SLEs typically appear in the cryptanalysis of symmetric ciphers. In particular, this type of SLE results from algebraic attacks which are generically applicable to any block or stream cipher. Moreover, fast SLE solvers for small to medium-sized dense SLEs are needed for high-performance implementations of the broad class of MQ cryptosystems which are currently among the most promising post-quantum asymmetric schemes.

Algebraic Cryptanalysis of Symmetric-Key Ciphers. The overwhelming majority of symmetric ciphers can be represented as finite state machines (FSMs) with or without memory whose state transition and output functions are defined over some binary fields. The output of such an FSM depends on input bits (which are often known) and the secret internal state that should be determined. In other words, for each output bit one obtains a (sometimes rather complicated) boolean equation in terms of the state and input bits. In the course of an algebraic attack, one tries to exploit some weakness of the considered cipher to derive advantageous non-linear equations of preferably low degree.

Solving a generic system of nonlinear equations is a well-known NP-hard problem. However, depending on the structure of the resulting nonlinear systems,

they might turn out solvable with a complexity lower than a brute-force search over the key space. Several methods of solving such nonlinear systems have been proposed, including the F4 [14] and F5 [15] Gröbner basis finding algorithms by Faugère which are indeed applicable to a subset of ciphers. Other approaches include SAT solvers recently applied to DES [10] and KeeLoq [11] by Courtois and Bard as well as MRHS [23] by Raddum and Semaev applied to DES.

Linearization methods [12,27] have received much attention during the last decade and are still widely used. Here, the nonlinear system is first simplified, then linearized and solved as an SLE. To make cryptanalysis with linearization methods feasible, one is reliant upon efficient SLE solvers being the object of study of the work at hand.

Enhancing Other Types of Symmetric Cryptanalysis. Solving systems of equations has recently turned out to be a useful toolbox to enhance other types of cryptanalysis for symmetric constructions. In [1], the *differential cryptanalysis* of PRESENT [8] was applied to a higher number of rounds by solving simple systems of equations instead of doing brute-force guessing. In [11], the *slide attack* [4] on KeeLoq was used to reduce the effective number of rounds to be analyzed where solving systems of equations with a simple algebraic structure was again applied to mount a more efficient attack. In [7] and [24], the nonlinear systems for AES were significantly simplified and successfully solved by adding equations resulting from collisions and Hamming weights of intermediate variables, respectively, obtained by observing the *side-channel* information (such as power consumption or electromagnetic radiation) of the implementation.

In all these approaches, a highly time-efficient solver for medium-sized SLEs would potentially increase the real-world efficiency of the attacks.

Two Case Studies. In fact, some of the engines described in this paper have already found application in cryptanalysis.

In the context of purely algebraic analysis, our high-speed hardware engines turn out vital for implementing the MRHS algorithm [23] that involves solving medium-sized systems of linear equations. For instance, in a recent paper [17] by Geiselmann, Matheis, and Steinwandt proposing a scalable hardware architecture, which implements the MRHS method, SLE solvers (slight variants of the GSMITH architecture presented in this article) for handling systems over $\mathbb{F}_2$ up to a dimension of 4096×4096 are employed.

In the context of enhancing other types of cryptanalysis using fast SLE-solvers, the authors of [5] present a hardware-assisted algebraic attack on the GSM stream cipher A5/2 based on the analysis by Barkan, Biham, and Keller [3]. Here we face the problem of solving 2^{16} systems of 555 linear equations over

$\mathbb{F}_2$.[1] The A5/2 session key can be immediately derived from a solution being consistent with the output stream of the cipher.

Both of these applications rely on the high performance of our architectures when solving *medium-sized* SLEs. However, we expect that also solving *larger* SLEs with essentially the same efficiency can become feasible when several chips are connected and I/O issues can be resolved, which is, however, not the topic of this paper.

Implementing Asymmetric Cryptography. Apart from the field of cryptanalysis, SLEs also play a central role in some cryptographic applications. For example, the performance of public-key signature schemes based on multivariate quadratic polynomials highly depends on the efficiency of solving small SLEs over finite extension fields. This class of signature schemes is of special interest due to its resistance to quantum computer attacks. To get an impression, for the generation of a signature using the Rainbow signature scheme [13] with recent recommendations for parameter choices, two SLEs of dimension 12×12 over $\mathbb{F}_{2^8}$ need to be solved. In [6], a generic hardware architecture for this kind of signature schemes is proposed with an SLE solver as described in this paper being a major building block. In the work at hand, we propose new area-optimized architectures (linear systolic SLE solvers) for this application that can significantly improve the hardware performance of signature schemes based on multivariate quadratic polynomials.

1.2 Existing Approaches to Hardware-Based SLE Solving

Among the algorithms suitable for the class of small and medium-sized dense SLEs, that we are considering, Gaussian elimination quickly turns out to be the algorithm of choice when aiming at a hardware implementation. This is due to its simplicity and generality, where the latter means that in contrast to other methods, SLEs are not required to satisfy any special properties.[2] In the following we give a brief overview of results dealing with parallelized implementations of Gaussian elimination.

In [22], an implementation of Gaussian elimination over $\mathbb{F}_2$ for the ICL-DAP (Distributed Array Processor by ICL) with an overall quadratic running time is described. In [16], the implementation of an adapted algorithm for computing LU-decompositions (e.g., see [26]) on an NVidia GeForce 7800 GPU is

[1] Note that the SLEs involved in this attack are actually underdetermined and thus cannot be solved in a common sense. However, it is already sufficient to determine a certain subset of 61 variables.

[2] For instance, in order to apply the conjugate gradient method [18] the coefficient matrix needs to be symmetric positive-definite.

presented exhibiting a cubic asymptotic time complexity.

There are a number of publications dealing with systolic array implementations of Gaussian elimination over prime fields. A systolic array is a network of simple processors, called cells, where each processor is connected to its direct neighbors. The array acts much like a filter on data passing through it. In [19], a triangular-shaped systolic architecture for Gaussian elimination over $\mathbb{F}_2$ is described. Wang and Lin [25] describe a triangular systolic array similar to the one in [19] realizing matrix triangularization over $\mathbb{F}_2$ by Gaussian elimination and backward-substitution. For solving SLEs of dimension $n \times n$ following both approaches an architecture consisting of n pivot cells (these are the cells on the main diagonal of the triangular array) and $\frac{n(n+1)}{2}$ basic cells is required. Furthermore, approximately n^2 delay elements (storing and delaying the transfer of intermediate results) are needed in addition. The solution of the first SLE is finished after $4n$ cycles. Assuming new SLEs can be fed in continuously row by row, a new solution vector is ready every n cycles. Unfortunately, important figures of an actual implementation like the complexity of the different cells, the area consumption of the systolic array, and the latency are missing completely.

1.3 Our Contributions

Based on the work in [19] we present and evaluate a triangular systolic architecture solving SLEs over finite extension fields $\mathbb{F}_{2^k}$ in Section 4. We consider slight variants of this systolic array, called triangular systolic network and triangular systolic lines solvers. The systolic network architecture is similar to the systolic array but exhibits global connections in both directions of the array. This allows the initial latency to be reduced to $2n$ clock cycles for the first result. However, clearly the critical path now depends on the size of the whole array. This significantly slows down the design already for small n as will be shown in Section 4. In contrast to that, in the systolic line architecture we only have global connections in one direction and local connections in the other. In this way, the initial running time can be reduced to $3n$ while still the architecture can be clocked at a high frequency. Furthermore, we propose a new systolic SLE solver architecture that requires only n arithmetic cells, but has a quadratic runtime. Hence, it allows for a different tradeoff of size versus throughput. These linear systolic solvers can also be implemented following the array or the lines approach described above. We implemented all variants of the triangular as well as of the linear systolic SLE solvers for different parameters. The implementation figures are provided in Section 5.

In addition to the systolic architectures, we present a different hardware approach realizing Gaussian elimination including backward-substitution over

$\mathbb{F}_{2^k}$. By slightly modifying the flow of the Gauss algorithm and parallelizing row operations, we are able to develop a highly efficient architecture. This architecture, called GSMITH, is described in Section 3. It consists of a rectangular array of simple cells exhibiting local as well as some global connections. We distinguish between pivot cells, pivot row and column cells, and basic cells. An architecture consisting of a single pivot cell, n pivot row and $n-1$ pivot column cells as well as $(n-1)n$ basic cells are required to solve SLEs of dimension $n \times n$. The GSMITH design requires only a single $\mathbb{F}_{2^k}$-inverter (contained in the pivot cell) whereas the triangular systolic architectures described above comprise n such inverters. These are very expensive building blocks (also in comparison to multipliers) in terms of area and latency when considering large extension fields. In contrast to all systolic architectures we present in this paper, the running time of GSMITH depends on the probability distribution of the matrix entries. For coefficient matrices with uniformly distributed entries and sufficiently large k (where $k \geq 8$ can already be considered sufficiently large), the running time for loading and solving the first SLE is $2n$. If the concrete application scenario allows to continuously load new SLEs column-wise into the architecture then the loading phase causes no additional overhead in terms of running time and a new solution vector can be output every n clock cycles. For the special case $\mathbb{F}_2$ both running times increase by n. We provide concrete implementation figures for the GSMITH design in Section 5.

Each of the architectures to be presented exhibits some unique features. Which architecture eventually provides the best performance depends on the concrete application scenario and implementation platform. Section 6 discusses these issues and draws some conclusions.

Though GSMITH and the triangular systolic SLE solvers have been briefly sketched as building blocks of other larger designs in [2] and [6], respectively, the paper at hand adds substantial contributions to the aforementioned works: It elaborates the description and analysis of these architectures (especially of the systolic solvers). New implementation results for all architectures under various parameter choices on the same hardware platforms are presented. Furthermore, we also propose some novel designs, linear systolic array and lines solvers, which are expected to allow for better time-area tradeoffs in certain application scenarios (such as [6]) than GSMITH and the triangular architectures. This paper for the first time contrasts and compares all the various designs.

2 Background on Gauss-Jordan Elimination over $\mathbb{F}_{2^k}$

The hardware SLE solvers we are going to consider implement variants of Gaussian/Gauss-Jordan elimination which is described in the following. In

order to maintain simplicity and consistency throughout this paper, we will use the following basic notation for describing matrix algorithms:

- Matrices are denoted by capital letters, e.g., A.
- The ith row vector of A is denoted by $\vec{a}_i$.
- The jth column vector of A is denoted by a_j.
- An element in the ith row and jth column of A is denoted by $a_{i,j}$.
- Column vectors are denoted by lowercase letters, e.g., b.

Gaussian elimination transforms a given system of linear equations of the form $A \cdot x = b$ into the equivalent system $U \cdot x = b'$, where U is an upper triangular matrix, by applying elementary *row operations*. The system can then trivially be solved by using backward substitution. The Gauss-Jordan algorithm is a variant of Gaussian elimination where the backward-substitution steps are interleaved with the elimination steps. Thus, when the algorithm terminates the resulting matrix is not just an upper triangular matrix U, but the identity matrix. In this way one immediately obtains the solution vector without doing any post-processing. A more complete introduction to Gaussian and Gauss-Jordan elimination can be found in [26].

When describing algorithms for solving SLEs in the following, we do not explicitly take care of the right hand side b. One can think of it as an additional, *passive column* to the coefficient matrix A. That means, the algorithm is carried out as if this column is not present (i.e., its presence does not influence the execution of the algorithm in any way), but every executed row/column operation also affects the passive column.

Algorithm 1 presents Gauss-Jordan elimination for an $n \times n$ matrix over $\mathbb{F}_{2^k}$. We assume that the matrix A given as input is non-singular. Steps 2 to 6 take care of the diagonal element $a_{\ell,\ell}$ (called pivot element) being non-zero by identifying the first row $\vec{a}_s$, where $s \geq \ell$, with a non-zero element in the currently considered column ℓ and exchanging this row with $\vec{a}_\ell$. This process is called *pivoting*. Steps 7 to 10 perform a *normalization* of the pivot row, i.e., the pivot element $a_{\ell,\ell}$ is transformed to 1 by multiplying all elements of the row with the inverse of $a_{\ell,\ell}$. Since $a_{\ell,i} = 0$ for $i < \ell$, real multiplications only need to be performed for elements to the right of the pivot element. Note that for the special case of LSEs over $\mathbb{F}_2$ normalization is not needed. Steps 11 to 18 are responsible for the *elimination* of all non-zero elements in the currently considered column by adding (the normalized) row $\vec{a}_\ell$. The three basic operations, namely pivoting, normalization, and elimination are repeated for all columns until eventually the identity matrix is obtained. In the end, the passive column would then contain the desired result. The worst-case as well as the average-case runtime (i.e., for matrices with uniformly random entries) of the general Gauss-Jordan algorithm is cubic in n.

Algorithm 1 Gauss-Jordan Elimination over $\mathbb{F}_{2^k}$

Input: Non-singular matrix $A \in \mathbb{F}_{2^k}^{n \times n}$

1: **for** each column $\ell = 1 : n$ **do**
2: $s \leftarrow \ell$
3: **while** $a_{s,\ell} = 0$ **do**
4: $s \leftarrow s + 1$
5: **end while**
6: exchange $\vec{a}_\ell$ with $\vec{a}_s$
7: **for** each element $i = \ell + 1 : n$ **do**
8: $a_{\ell,i} \leftarrow a_{\ell,i} \cdot a_{\ell,\ell}^{-1}$
9: **end for**
10: $a_{\ell,\ell} \leftarrow 1$
11: **for** each row $i = 1 : n$ **do**
12: **if** $(i \neq \ell) \wedge (a_{i,\ell} \neq 0)$ **then**
13: **for** each element $j = \ell + 1 : n$ **do**
14: $a_{i,j} \leftarrow a_{i,j} \oplus a_{\ell,j} \cdot a_{i,\ell}$
15: **end for**
16: $a_{i,\ell} \leftarrow 0$
17: **end if**
18: **end for**
19: **end for**

3 GSMITH — A Hardware SLE Solver for $\mathbb{F}_{2^k}$

This section describes the GSMITH architecture for solving SLEs over the extension field $\mathbb{F}_{2^k}$. GSMITH stands for "Generalized Scalable Matrix Inversion on Time-Area optimized Hardware" and is a generalization of the SMITH architecture which was originally presented in [9]. The design of GSMITH was first briefly sketched in [2] as the main building block of a time-optimized core for the Rainbow signature scheme.

3.1 Adapting Gauss-Jordan Elimination over $\mathbb{F}_{2^k}$ to Hardware

The main idea in order to obtain an efficient hardware implementation of the Gauss-Jordan algorithm is to perform all steps required for both elimination and normalization in parallel, instead of executing them one-by-one. Moreover, we change the flow of the algorithm in order to simplify an implementation in hardware: having a fixed matrix and changing the row under examination is equivalent to always examining the first row and shifting the whole matrix accordingly. Obviously, this is also true for columns.

Gaussian elimination over $\mathbb{F}_{2^k}$ then transforms to Algorithm 2 as described in the following. We iterate over all columns of the matrix A. In the ℓth iteration we perform the following steps to obtain a pivot element: we do a cyclic shift-up

Algorithm 2 Hardware-based Gauss-Jordan Elimination over $\mathbb{F}_{2^k}$

Input: Non-singular matrix $A \in \mathbb{F}_{2^k}^{n \times n}$

1: **for** each column $\ell = 1 : n$ **do**
2: **while** $a_{1,1} = 0$ **do**
3: $A \leftarrow \mathsf{shiftup}(n - \ell + 1, A)$;
4: **end while**
5: $\vec{a}_1 \leftarrow \mathsf{normalize}(\vec{a}_1)$
6: $A \leftarrow \mathsf{eliminate}(A)$;
7: **end for**

of all rows not yet used for elimination (due to the shifting approach these are the first $n - \ell + 1$ rows) until the element at the fixed pivot position ($a_{1,1}$) is non-zero. More precisely, the mapping shiftup on $(n - \ell + 1, A)$ is computed [3] (in one step) until $a_{1,1}$ is a non-zero element, where shiftup is defined as:

$$\begin{gathered}\mathsf{shiftup} : \{1, \ldots, n\} \times \mathbb{F}_{2^k}^{n \times n} \to \mathbb{F}_{2^k}^{n \times n} \\ (i, (\vec{a}_1, \ldots, \vec{a}_n)^T) \mapsto (\vec{a}_2, \ldots, \vec{a}_i, \vec{a}_1, \vec{a}_{i+1}, \ldots, \vec{a}_n)^T\end{gathered}$$

After a pivot element has been found in the ℓth iteration (it is now located in the first row), we normalize the row by applying the mapping

$$\begin{gathered}\mathsf{normalize} : \mathbb{F}_{2^k}^{n} \to \mathbb{F}_{2^k}^{n} \\ (a_{i,1}, a_{i,2}, \ldots, a_{i,n}) \mapsto (1, a_{i,2} \cdot a_{1,1}^{-1}, \ldots, a_{i,n} \cdot a_{1,1}^{-1})\end{gathered}$$

Next, we add the first (normalized) row $\vec{a}_1$ to all other rows $\vec{a}_i$ where $i \neq 1$ and $a_{i,1} \neq 0$ to eliminate these elements. In addition, we do a cyclic shift-up of all rows and a cyclic shift-left of all columns. By doing the cyclic shift-up operations after an elimination, rows already used for elimination are "collected" at the bottom of the matrix, which ensures that these rows are not involved in the pivoting step anymore. The cyclic shift-left ensures that the elements which should be eliminated in the upcoming iteration are located in the first column of the matrix. More precisely, the following (partial) mapping is computed which combines the actual elimination, the cyclic shift-up and the cyclic shift-left:

$$\mathsf{eliminate} : \mathbb{F}_{2^k}^{n \times n} \to \mathbb{F}_{2^k}^{n \times n}$$

$$\begin{pmatrix} 1 & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{pmatrix} \mapsto \begin{pmatrix} a_{2,2} \oplus (a_{1,2} \cdot a_{2,1}) & \ldots & a_{2,n} \oplus (a_{1,n} \cdot a_{2,1}) & 0 \\ \vdots & & \vdots & \vdots \\ a_{n,2} \oplus (a_{1,2} \cdot a_{n,1}) & \ldots & a_{n,n} \oplus (a_{1,n} \cdot a_{n,1}) & 0 \\ a_{1,2} & \ldots & a_{1,n} & 1 \end{pmatrix}$$

[3] In the actual hardware implementation we keep track of the used rows by means of a `used`-flag instead of using a counter.

The hardware architecture GSMITH is able to compute the mapping shiftup in one clock. Also applying normalize immediately followed by eliminate will only require a single clock cycle in total.

Running Time. We restrict our consideration to the expected running time for matrices with uniformly chosen entries since this is the most common type appearing in cryptographic applications. Note that shiftup and the composition of normalize and eliminate are the two primitive operations of our hardware architecture. Clearly, we always have n applications of eliminate/normalize independently of the distribution of the entries of the matrix. To determine the required number of shiftup applications we observe the following: If the matrix entries are uniformly distributed, one has $\alpha := \Pr[a_{1,1} = 0] = \frac{1}{2^k}$ in the first iteration. In fact, this probability remains constant throughout all iterations. This is because the sum $x \oplus y$ of two uniformly chosen elements $x, y \in \mathbb{F}_{2^k}$ is again a uniformly distributed field element and the same holds for the product $x \cdot y$ of a uniform element $x \in \mathbb{F}_{2^k}$ and a uniform element $y \in \mathbb{F}_{2^k}^*$. Hence, the elements

$$a_{1,j} \leftarrow a_{1,j} \cdot a_{1,1}^{-1},$$

where $2 \leq j \leq n$, computed by normalize as well as the elements

$$a_{i-1,j-1} \leftarrow a_{i,j} \oplus a_{1,j} \cdot a_{i,1},$$

where $2 \leq i, j \leq n$, resulting from eliminate are uniform field elements. Therefore, the expected number of shiftup applications is at most $\frac{1}{1-\alpha} - 1$ in each iteration and thus

$$n\left(\frac{1}{1-\alpha} - 1\right)$$

in total. This leads to the following expected number of primitive operations executed by Algorithm 2:

Theorem 1 (Expected Running Time) *Let $A \in \mathbb{F}_{2^k}^{n \times n}$ be a matrix with uniformly random entries. Then the expected running time of Algorithm 2 on input A is $\frac{n}{1-\frac{1}{2^k}}$.*

Note that for the special case $k = 1$, we have an expected running time of $2n$. For sufficiently large field extensions (i.e., for sufficiently large values of k), the required number of shiftup operations can be neglected and so only about n clock cycles are needed to execute the algorithm. For instance, for random coefficient matrices over $\mathbb{F}_{2^8}$ of dimension 12×12 only about $12.05 \approx n$ clock cycles are required.

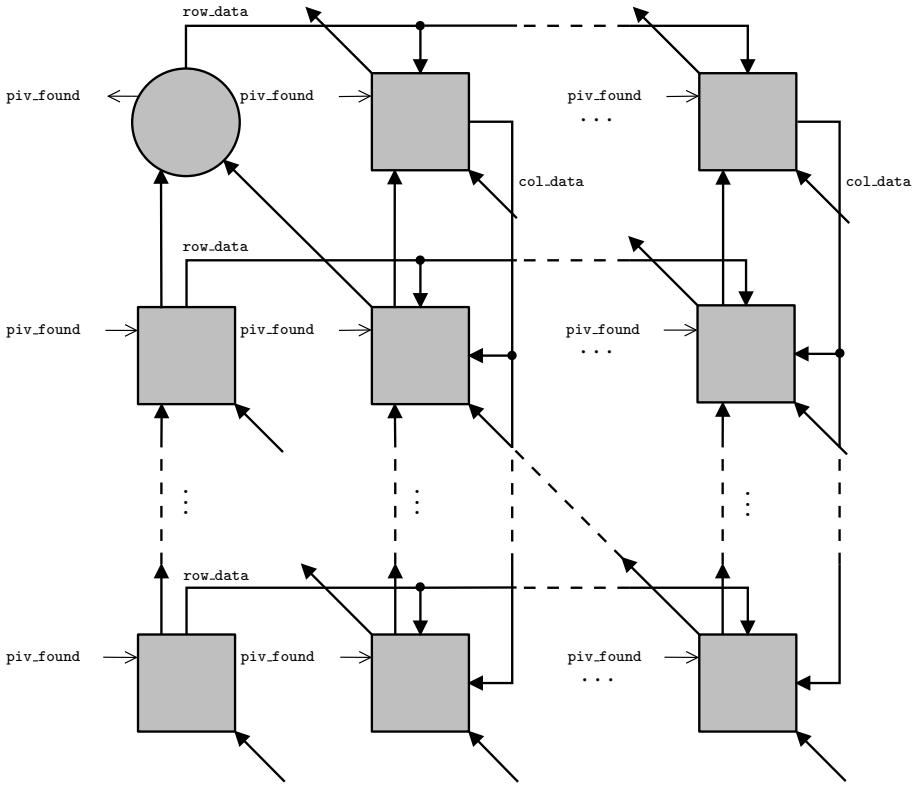

Fig. 1. Structure and signal flow of GSMITH.

3.2 The Proposed Hardware Architecture

The basic structure of the GSMITH architecture is a rectangular array of $n \times (n+1)$ cells as shown in Figure 1. In the following we describe these building blocks. The architecture consists of four different types of cells which realize the three basic operations involved in Algorithm 2. The cell in the upper left corner of the array is called the pivot cell, the cells in the first row and the first column other than the pivot cell are called the pivot row and pivot column cells, respectively. The remaining cells are called basic cells.

We start by describing the basic cells being the most complex. Figure 2 shows the design and interconnections of this cell type. Each cell has local connections to four of its neighbors and some global connections. Considering Figure 2, we additionally distinguish between data connections (depicted by thick lines), which are k-bit buses for exchanging elements from $\mathbb{F}_{2^k}$ between the cells, and 1-bit control signal connections (depicted by thin lines).

Let us first consider the local connections: A cell has a data and a control connection (`out1` and `lock_lower_row_out`) to its upper neighbor, a data and a control connection (`in1` and `lock_lower_row_in`) to its lower neighbor, and only data connections (`out2` and `in2`) to its upper left and lower right neighbor in the array. Note that a cell receives inputs only from its lower direct neigh-

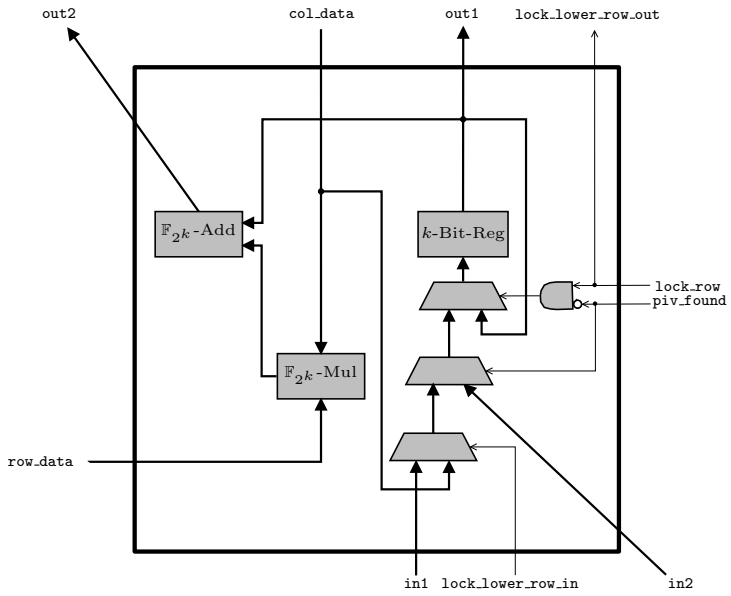

Fig. 2. GSMITH: Schematic of a basic cell

bors and sends data only to its upper direct neighbors. The array is wrapped around in the sense that the lower right neighbors of cells in the bottom row are the corresponding pivot row cells (cf. Figure 1). Furthermore, a cell is also connected to a global network providing a control signal (`pivot_found`) from the pivot cell, a data signal (`row_data` $= a_{i,1}$) from the first cell of the respective row (a pivot column cell) and a data signal (`col_data` $= a_{1,j}$) from the first cell of the respective column (a pivot row cell) the considered cell belongs to. Note that the `used`-flags for the corresponding row and the row below are provided by the signals `lock_row` and `lock_lower_row_in`, respectively. Additionally, every cell is connected to the global clock (`clk`) and reset (`rst`) network. For the sake of simplicity, the latter two control signals are omitted in the figure.

Each basic cell stores one element $a_{i,j} \in \mathbb{F}_{2^k}$ of the matrix in its k-bit register and contains one adder and one multiplier to implement the eliminate operation. Moreover, it also contains some multiplexers controlled by the signals `piv_found`, `lock_row`, and `lock_lower_row_in`. By means of these multiplexers, we decide which data is input to the register depending on the operation that is currently executed. The register content is kept if the current pivot element is zero (`piv_found` $= 0$) and the current row is locked (`lock_row` $= 1$). Otherwise, new data is stored. Assuming a non-zero pivot has been found, `in2` is input to the register. If `piv_found` is zero, `in1` or `col_data` is stored depending on whether the lower row is locked (`lock_lower_row_in` $= 1$).

Let us briefly consider the design of the other cell types in comparison to the basic cell. In the pivot cell, shown in Figure 3, we do not need any multiplier or adder but an inverter combined with a zero-tester. On the one hand, this

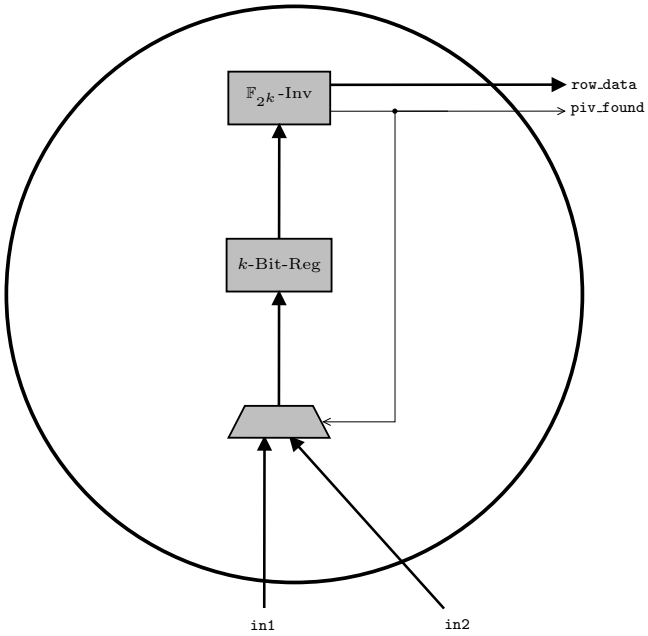

Fig. 3. GSMITH: Schematic of a pivot cell

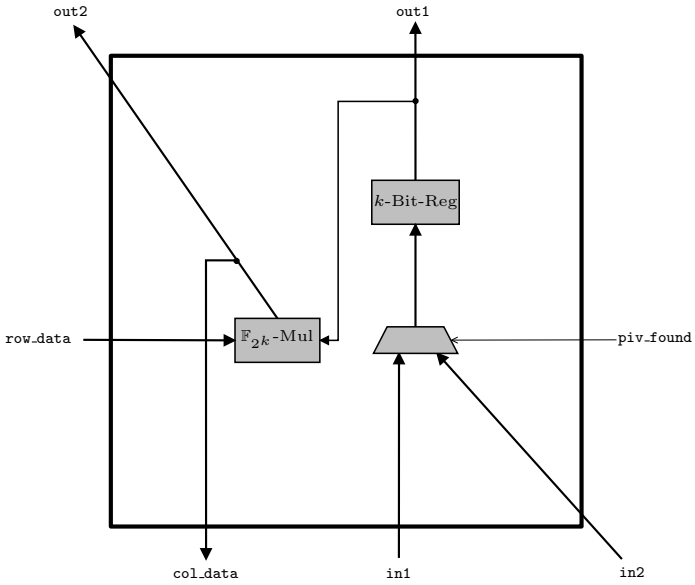

Fig. 4. GSMITH: Schematic of a pivot row cell

component checks whether the current content of the register is the zero element and sets the global control signal `piv_found` to zero in this case. On the other hand, if the register content is not equal to zero, the second output `row_data` of the component equals the inverse of this element which is needed to perform the normalize operation (cf. Section 3.1). The pivot cell does not need an `out1` nor an `out2` data connection, since these signals can be determined from the operation that is currently executed (which is indicated by the global signal `piv_found`).

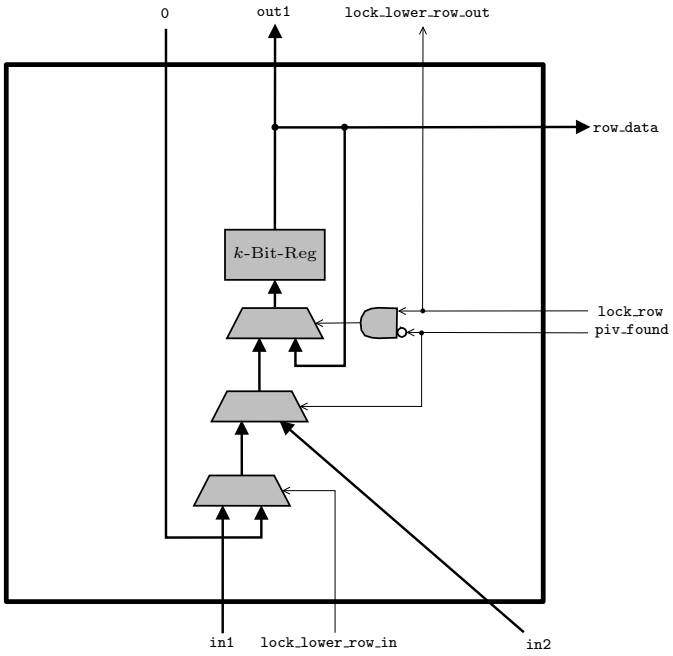

Fig. 5. GSMITH: Schematic of a pivot column cell

Figure 4 graphically presents the structure of a pivot row cell. In comparison to the basic cell type, these cells are equipped with a multiplier but no adder. Their `row_data` bus is connected to the output of the inverter of the pivot cell. The data received via this bus is input to the multiplier of a pivot row cell. The second input of the multiplier is the current content of the register. The output of the multiplier is connected to all basic cells of the respective column and provides the data signal `col_data`. This output is also the `out2` signal of a pivot row cell.

A pivot column cell, depicted in Figure 5, contains neither an adder nor a multiplier. Its register is connected to all basic cells of the respective row thereby providing the data signal `row_data` required to execute the eliminate operation. Note that no `out2` data connection is required for these cells since this signal would always be equal to zero if processed by other cells. Furthermore, the `col_data` signal input to these cells is always set to zero.

Finally, it is worth mentioning that for the special case $\mathbb{F}_2$ we obtain a significantly less complex hardware architecture: Since normalize does not need to be implemented for this case no inverter is needed. Furthermore, multipliers and adders boil down to simple AND and XOR gates, respectively. However, the probability of an element being 0 is much higher then, which leads to a slightly higher running time, since on average more shiftup operations have to be performed till the next nonzero element is found.

Realizing the shiftup *operation.* If the current content of the pivot cell is equal to zero, the architecture performs a simple cyclic shift-up operation: all cells

with the used-flag set to 0 (lock_row = 0) simply shift their values, being the content of their register, one row up. Values in the upmost row will be shifted to the lowest unused row, resulting in a cyclic shift of all unused rows. The registers of all cells with the used-flag set to 1 (lock_row = 1) are not affected by the shift and stay constant. For realizing the cyclic shift from the first row to the last unused row we make use of the col_data, lock_row, and lock_lower_row_in signals. More precisely, if for a cell holds that the lock_row signal is 0 and the lock_lower_row_in signal equals 1 then the value of col_data is stored to the register.

In the very beginning, values have to be loaded into the design. We can reuse the shiftup operation to fill the architecture row-wise: Since a reset of the design sets the content of all cells to 0, a shift-up is applied n times until the matrix is completely loaded, where the matrix is fed row-wise into the last row of cells.

Realizing the normalize *and* eliminate *operations.* The operations normalize and eliminate are invoked when the pivot element is not equal to zero (piv_found = 1). The normalize operation is realized by the pivot cell and pivot row cells whereas the eliminate operation is essentially performed by the basic cells. In the case that the pivot element is non-zero, the pivot cell computes the inverse of this value which is then multiplied by the pivot row cells with the content of their registers. This product is provided to the basic cells in each column by the corresponding pivot row cell using the col_data bus. A basic cell multiplies this data with the register content of the respective pivot column cell provided by the row_data bus, adds the result to the content of its own register and sends the sum to its upper left neighbor (out2). Also the output of the multiplier of a pivot row cell is provided to the upper left neighbor with respect to the wraparound. In the same clock cycle, each cell writes the data received from the in2 bus into its register.

End of computation. When all rows have actively been used for pivoting, the used-flag reaches the topmost row and the architecture stops computing. In our design, this signal is used for indicating the end of computation. The solution of the SLE can simply be extracted by reading out the registers of the first column (using the row_data-signals which are wired out explicitly). Depending on the outer control logic, the topmost used-flag can be used to indicate the finished state and eventually stop the computation until the result has been read.

3.2.1 Column-Wise Pipelining

In order to improve the performance when solving many SLEs in succession using the GSMITH architecture, one can exploit the fact that once a column

has undergone elimination it does not have to be kept inside the architecture anymore. So instead of keeping these columns, after each elimination we can load one column of a new matrix into the rightmost column of cells. Using this approach, solving the current matrix and loading a new matrix can be done in parallel. However, there are some technical difficulties one has to cope with: it has to be ensured that neither row permutations nor eliminate operations can corrupt the columns of the new matrix while processing the old one. This can be achieved with a flag similar to the already implemented `used`-flag protecting the rightmost columns. Moreover, when loading the columns of the new matrix one needs to take care of the fact that these columns are shifted diagonally through the architecture. Hence, the first matrix column experiences n cyclic shifts until it reaches the first column of cells, the second matrix column undergoes $n-1$ shifts and so on. So column elements need to be reordered correspondingly before loading, in order to avoid a permutation.

3.2.2 Handling Not Uniquely Solvable SLEs

For the sake of simplicity, we always assumed uniquely solvable SLEs until now. However, in some applications the solvability of an SLE might not be known prior to the computation. The design can easily be extended to also detect and handle not uniquely solvable SLEs: In this case, we will run into an infinite loop of shiftup operations at some point of the computation. To detect such a loop and avoid a *deadlock*, we have to initialize a counter after every elimination step to limit the number of the subsequent shiftup operations. This number must be less than the number of rows minus the number of locked rows.

4 Systolic SLE Solvers for $\mathbb{F}_{2^k}$

In this section we show that SLE-solving over $\mathbb{F}_{2^k}$ can be efficiently realized using systolic array processors in hardware. Our first approach is based on the work by Wang and Ling [25] presenting an SLE solver for the special case of $\mathbb{F}_2$. We extend this architecture by introducing three different triangular-shaped systolic architectures implementing Gauss-Jordan elimination over $\mathbb{F}_{2^k}$. Our second approach allows for a further reduction of the number of array processors, resulting in a linear array solver design. These one-dimensional SLE solvers trade a significant area reduction for a reduced throughput and are especially suitable when cheap memory is available.

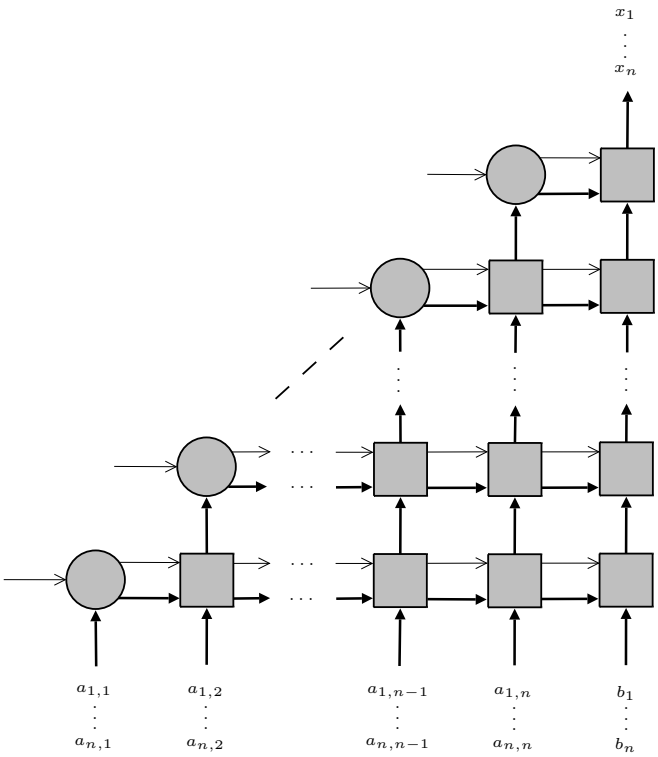

Fig. 6. Structure and signal flow of a triangular systolic network architecture.

4.1 Two-Dimensional SLE Solver Architectures over $\mathbb{F}_{2^k}$

Extending the approach by Wang and Ling [25] to Gauss-Jordan elimination over $\mathbb{F}_{2^k}$, we introduce three new triangular-shaped systolic architectures. These are referred to as *triangular systolic network (TSN)*, *triangular systolic lines (TSL)*, and *triangular systolic array (TSA)*, respectively. A brief description of the triangular systolic architectures presented in the following has been published in [6] where they are used as a building block for a multivariate signature generation core.

Systolic arrays feature a critical path that is independent of the size of the array. The price paid is a large initial delay. Yet, due to the structure of the systolic architectures, a new SLE can already be passed into the array while it is still processing the previous one. A full solution of an $n \times n$ SLE is generated after $4n$ cycles and every n cycles thereafter. The solution is computed in a serial fashion.

Systolic networks allow signals to propagate through the whole architecture in a single clock cycle. This allows the initial latency to be reduced to $2n$ clock cycles for the first result. Of course the critical path now depends on the size of the whole array, slowing the design down for huge systems of equations. Systolic arrays can be derived from systolic networks by putting delay elements

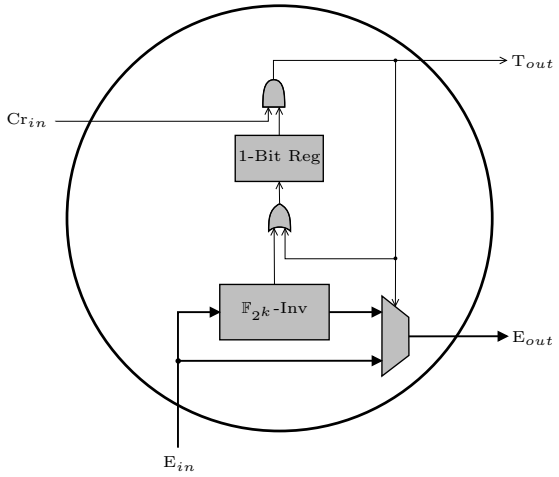

Fig. 7. Systolic Network: Schematic of a pivot cell.

(registers) into the signal paths between the cells. Systolic lines show one form of tradeoff between the first two approaches and have an initial delay of $3n$ clock cycles.

We first introduce the systolic network as a basic architecture. The two-dimensional systolic array and systolic lines architectures can be explained as variants of the network architecture.

4.1.1 Triangular Systolic Network

The triangular systolic network (TSN) architecture, depicted in Figure 6, is a mesh of simple processors connected in a triangular structure. The cells on the main diagonal are called pivot cells and the remaining cells are called basic cells. For solving an $n \times n$ SLE, we employ an architecture consisting of n pivot cells and $n(n+1)/2$ basic cells.

Unlike GSMITH, the systolic designs start processing the input SLE as it is clocked into the design. Each line of the design performs one iteration step of Gaussian elimination during every clock cycle. We first explain the design of the two different cell types before giving details on how the cells interact to solve the SLE.

The design of a pivot cell is shown in Figure 7. The most complex component of this cell is an $\mathbb{F}_{2^k}$-inverter combined with a zero-tester, which is needed for performing normalization. Furthermore, the cell contains a single 1-bit register to store whether a non-zero pivot element has been found. Initially, the register is set to zero. If the input E_{in}, which is an element of $\mathbb{F}_{2^k}$, is zero and the signal T_{out} is also zero, the output E_{out} is equal to 1. The inverter component is designed to output a 1 when receiving the zero element as input. If E_{in} is not equal to zero and T_{out} is zero, the output E_{out} is the $\mathbb{F}_{2^k}$-inverse of

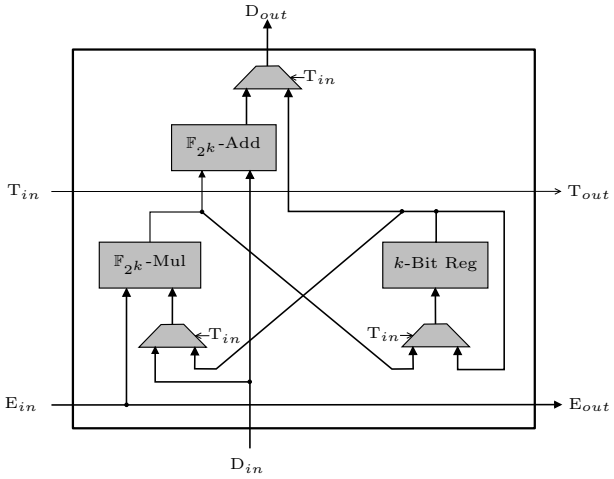

Fig. 8. Systolic Network: Schematic of a basic cell.

E_{in}. In the case that T_{out} is equal to 1, the data E_{in} is simply passed unchanged to E_{out}. This is required for computing multiples of the normalized pivot row in the course of an elimination operation. The signal Cr_{in} is used to control the mode of operation of the row of cells the corresponding pivot cell belongs to. We have two such modes, namely the *initialization* and *backward-substitution* mode ($\mathrm{Cr}_{in} = 0$) and the *elimination* mode ($\mathrm{Cr}_{in} = 1$). In initialization mode, data of a new SLE is loaded into the line of cells and old data, stored in the registers of the cells, is passed to the next line of cells for backward-substitution. The elimination mode is turned on until all rows of the new SLE passed the respective line of cells. Switching back to initialization mode allows for restarting the data loading process while the result for the previous data is still being produced.

A basic cell, shown in Figure 8, comprises a k-bit register, an $\mathbb{F}_{2^k}$-multiplier, and an $\mathbb{F}_{2^k}$-adder. Furthermore, the cell contains a few multiplexers. If no pivot element has been found yet ($\mathrm{T}_{in} = 0$), the entering data D_{in} is multiplied with E_{in} and stored in the register of the cell. Thus, in the case of the currently processed pivot element being non-zero, normalization is applied and the register content equals the inverse of the pivot multiplied by D_{in}. If the pivot is zero, the output of the multiplication is equal to the element D_{in} (since E_{in} has been set to 1 by the pivot cell) which is simply stored and passed to the next row in the next clock cycle. Note that in both cases, the old content of the register is passed as D_{out} to the next row of cells. If a pivot element has been found before ($\mathrm{T}_{in} = 1$), elimination is performed, *i.e.*, we compute

$$\mathrm{D}_{out} = \mathrm{D}_{in} + \mathrm{E}_{in} r \,,$$

where r denotes the current content of the register. The result is passed as D_{out} to the upper row in the same clock cycle.

Every clock cycle one equation of the SLE is fed into the architecture, in a

row-wise fashion. The kth line of cells (counted from the bottom) executes the kth iteration of the Gauss-Jordan algorithm. However, the iterations are not performed sequentially by the architecture, *i.e.*, every line, once filled, performs one step of one elimination round during each clock cycle in parallel.

Let us assume a new matrix row $\vec{a}_1$ is fed into a systolic network in initialization mode. It is then normalized and stored in the first line of the array if the pivot is not zero. Else it will be stored, but the line remains uninitialized and the matrix row will just be passed on, as the first element is considered eliminated. After that, a new row $\vec{a}_2$ enters the network. If the first line of cells was initialized, the first element of $\vec{a}_2$ is eliminated and the (modified) row is passed on to the second line. Else, $\vec{a}_2$ will be processed in the same manner as the first matrix row was processed before. Now this procedure is repeated for every entering row, one element being eliminated at each line until it hits an uninitialized line where it will be stored.

Once all equations have been loaded and the whole network is filled, one line after the next is forced into backward-substitution mode ($\mathrm{Cr}_{in} = 0$) again, beginning with the first line. In this way the matrix row stored in this line is being eliminated at each of the following lines until finally the first result x_1 is produced as output of the main cell of the upmost line of the network. The first component x_1 of the solution vector is ready after $n + 1$ clock cycles. By producing a new component in each cycle, the complete vector is ready after $2n$ cycles. The runtime of this architecture is independent of the distribution of the matrix entries, *i.e.*, fixed for all matrices. Since backward-substitution mode and initialization mode are the same, SLEs can be loaded in a serial fashion, loading the first equation of a new SLE directly after loading the last of the previous SLE.

Note that the architecture just described allows signals to propagate through the whole network in a single clock cycle. Hence the critical path certainly depends on the problem size n slowing the design down for large systems of equations.

4.1.2 Triangular Systolic Array

To make the critical path independent of the problem size, we can interpose a one-cycle delay element in each path between two cells to buffer intermediate results. Such a delay element can be a k-bit or a 1-bit register depending on whether the considered path is a data or control signal bus. Clearly, this modification increases the area consumption compared to the original design. Furthermore, to make the architecture work correctly also the input data needs to be appropriately rearranged. More precisely, the feeding of the SLE matrix A into the array must be delayed column-wise. Only $a_{1,1}$ enters the systolic

array at the first clock cycle, then $a_{2,1}$ enters the first and $a_{1,2}$ enters the second cell, and so on. We will refer to a matrix in this form as *skewed* from now on. In this way, the last piece of data being the element b_n enters the architecture in the $2n$th cycle. Backward-substitution starts for each column once the last element has entered. The control signal Cr_{in} must be delayed for two clock cycles between two lines. The first component of the solution vector appears in the $(3n + 1)$th and the last component in the $4n$th cycle. Hence, the initial runtime in terms of clock cycles has been doubled compared to the original architecture. However, note that assuming new SLEs can be fed in continuously as before, a new solution vector is generated every n cycles. We call this fully pipelined variant a triangular systolic array (TSA) SLE solver.

4.1.3 Triangular Systolic Lines

It is also possible to strike a balance between these two extreme approaches. In order to reduce latency it is almost as helpful to insert only delay elements into vertical paths. Since no delay elements are present within horizontal paths, the input data does not need to be skewed. Hence, as in the original design, a complete row can enter the architecture every clock cycle. Thus, the last matrix row is the input of the nth cycle and the backward-substitution phase starts one cycle later. This approach results in the first component of the solution vector leaving the architecture in the $(2n + 1)$th cycle. The full solution is ready after $3n$ clock cycles. We call this variant a triangular systolic lines (TSL) SLE solver.

Please note that systolic lines are a special pipelining strategy that only works in cases where the design is two-dimensional (or more). It is not straightforwardly applicable to one-dimensional systolic arrays. Also, other tradeoffs between the systolic array and the systolic network approach are possible. Registers could, for example be put only after every ith row and every jth column. The size of blocks of cells that are connected without delay buffers can then be chosen in a way to achieve a critical path length as close to a desired clock frequency as possible, while slightly lowering resource consumption and initial delay time.

4.2 One-Dimensional SLE Solver Architectures for $\mathbb{F}_{2^k}$

Although the architectures of Section 4.1 pose different tradeoffs between area and latency, the number of cells remains the same in all cases. A different way to trade size for speed is achieved by reducing the number of processing elements in the design.

We can achieve this by taking only the first line of the triangular systolic array

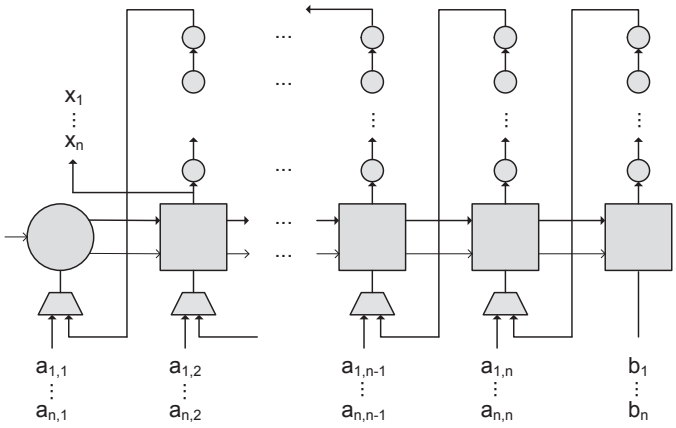

Fig. 9. Structure and signal flow of a linear systolic lines architecture.

or systolic line approach (the systolic network does not apply in this case). The arithmetic processing line consists of one pivot cell and n main cells. While the triangular approaches have n lines that, in the optimal case, are all processing one elimination of the Gauss-Jordan algorithm in parallel, the new design has only one line, and only one elimination can be performed per clock cycle. Hence the execution time is increased by a factor of n. Furthermore we need an array of shift registers to store intermediate results. For the one-dimensional SLE solver we propose two different architectures, the linear systolic array (LSA) approach and the linear systolic line (LSL) approach. For the systolic array solver, the SLE must be fed into the engine in skewed form. But contrary to the two-dimensional TSA, the LSA does not feature a higher delay than the LSL. The effective operation time of both architectures is always n^2. Contrary to the triangular architectures, the linear engines do not, however, benefit from an SLE pipelining.

We now describe the functionality of the linear systolic lines SLE solver in more detail. The systolic array architecture is again explained as a variation of the former.

4.2.1 Linear Systolic Line

The linear systolic line SLE solver is depicted in Figure 9. The arithmetic part consists of one pivot cell and n main cells, just as the initial line of the triangular SLE solver. The data is clocked into the LSL in the same manner as for the TSL solver in Section 4.1, that is, a full equation at each cycle. The output D_{out} of each main cell is stored in a shift register with $n-1$ consecutive stages. The output of the shift registers is connected to the input of the arithmetic cells via a multiplexer. Note that the output of the shift register of column i is fed into the input of column $i-1$. The last main cell n does not need a multiplexer, because it does not get a feedback from a shift register.

Let us assume that a matrix row $\vec{a}_1$ is given such that the first component of this vector is not equal to zero. Then this row is normalized and stored in the registers of the main cells. The following matrix rows are processed (elimination of the first column) and subsequently pushed into the n shift registers. After n clock cycles, the first column has been eliminated and all equations are stored in the shift registers and the main cells, respectively. Now the multiplexers are set to feedback mode and the control signal Cr_{in} of the pivot cell is forced to zero again. Every n clock cycles, the control signal Cr_{in} of the processing line has to be reset by the control logic, so the current register contents r are pushed into the shift registers and the next row is fed in for pivoting and normalization. Now again, all rows are reduced by eliminating the next column vector, including the row that has been used for elimination before. Hence, after n repetitions the whole matrix has been converted into a diagonal matrix and the solution vector x is generated as the output of the first main cell. The first component of this vector is generated after $n^2 - n + 1$ clock cycles, and the full vector is ready after n^2 clock cycles. Please note that every n clock cycles, one column becomes obsolete, leaving another shift register empty. We call this variant a linear systolic line (LSL) SLE solver.

Although one shift register is freed every n clock cycles, we cannot make use of pipelining as in Section 4.1. This is easily understandable, as the pivot cell is used in every single clock cycle.

4.2.2 Linear Systolic Array

As before, we can construct a systolic array variant of a systolic line solution by buffering all horizontal signals. The SLE equations have to be fed into the architecture in skewed form again. Because of the skewing, the depth of the shift registers can be decreased by one, resulting in a depth of only $(n - 2)$.

Although it takes $2n$ clock cycles until the whole SLE is loaded into the solver, the solver finishes after $n^2 + 1$ clock cycles, due to the reduced length of the shift register. By reducing the depth of the shift registers we counter the effect of skewing that usually implies that column i is ahead one time step of column $i + 1$. The one additional clock cycle delay in the total computation time of the SLE is due to the pipelining between the boundary cell and the first main cell, which computes the outputs x_i. We call this variant a linear systolic array (LSA) SLE solver.

4.3 Handling Not Uniquely Solvable SLEs

The SLEs considered in various application scenarios are not always uniquely solvable. Fortunately, the detection of unsolvable SLEs is simple for all de-

scribed architectures. For the triangular designs, the detection is performed by checking the state of the pivot cells after $3n$ clock cycles, for a systolic array, $2n$ clock cycles for a systolic lines architecture, and n clock cycles for a systolic network, respectively. If any of the 1-bit registers contained in these pivot cells has not been set to 1, the system is not uniquely solvable. For the linear SLE solvers, the 1-bit register of the pivot cell must be checked every n clock cycles, just before they are reset to 0 for loading a new row $\vec{a}_i$ for elimination. If the register contains a zero at any of the n time instances it is checked, the SLE is not uniquely solvable.

5 Performance

This section provides implementation results for the previously presented architectures such as running time and area consumption on a low-cost FPGA, the Xilinx Spartan-3 XC3S1500 (13,312 Slices, 26,624 4-bit-LUTs/FFs, 300 MHz), as well as a high-performance FPGA, the Xilinx Virtex-5 XC5VLX50 (7,200 Slices, 28,800 6-bit-LUTs/FFs, 550 MHz). We consider these measures for SLEs over $\mathbb{F}_2$ and $\mathbb{F}_{2^8}$ which are of great importance for cryptographic applications [5,6,17]. The architectures have been implemented in VHDL and are kept generic to allow for easy modification of the SLE dimension. For synthesis, map, and place and route, Xilinx ISE 9.2 was used. Note that we did not optimize the code for a certain FPGA. Thus, FPGA-specific optimizations might lead to improved area usage or speed.

When considering the following implementation figures one should keep in mind that in order to fully exploit the performance of the presented architectures, the speed of data in- and output is crucial. Hence, interfaces for loading LSEs into the solvers need to be capable of running at high-speed (e.g., by using PCI-express or proprietary systems such as those on a Cray-XD1). Alternatively, an integrated pre- and post-processing may be required in order to prevent solvers from running idle. For instance, this is the reason why the generation of SLEs as well as the verification of the solutions is performed on-chip in the case of the GSMITH-based A5/2 attack engine presented in [5].

5.1 Theoretical Performance

Table 1 shows rough estimates of the theoretical time and area complexities of the various architectures we introduced throughout the paper. Execution time and area complexity are given as a function of SLE size n and field size k. We decided to separate the area complexity of logic cells and memory, since memory can in some cases be realized very efficiently, while logic blocks like

inverters can not. While the choice of the field does not have any impact on the structure of the SLE solver architecture, it does impact the throughput and size of the components, namely of the basic and the pivot cells. We estimate the area complexity of $O(1)$-time adders and multipliers for $\mathbb{F}_{2^k}$ by k and k^2 [20], respectively. Unfortunately, general estimates for bit-parallel constant-time inverters over $\mathbb{F}_{2^k}$ do not seem to be available in the literature. For this reason, we restrict our estimates to the cryptographically interesting case $k = (2^4)^t$, which has been extensively studied in [20]. Based on the results in [20], the area complexity of such inverters is approximated by k^3. Clearly, the size of a k-bit register can be estimated by k.

The GSMITH architecture for SLEs over $\mathbb{F}_{2^k}$ of dimension $n \times n$ is basically comprised of the following building blocks: $n(n+1)$ k-bit registers, a single $\mathbb{F}_{2^k}$-inverter for computing $a_{1,1}^{-1}$, n $\mathbb{F}_{2^k}$-multipliers for normalizing the first row, and $n(n-1)$ $\mathbb{F}_{2^k}$-multipliers and adders for the elimination step. Hence, for sufficiently large values of n, the overall hardware complexity is dominated by the flip-flops and multipliers. The expected number of clock cycles needed by GSMITH for processing SLEs over $\mathbb{F}_{2^k}$ of this dimension (assuming uniformly distributed coefficients) is $\frac{n}{1-2^{-k}}$ in the pipelined and $n + \frac{n}{1-2^{-k}}$ in the non-pipelined case. This boils down to $2n$ and $3n$, respectively, over $\mathbb{F}_2$ as well as n and $2n$, respectively, for sufficiently large k (e.g., $k \geq 8$).

All triangular systolic architectures we have seen, essentially consist of the following building blocks: n $\mathbb{F}_{2^k}$-inverters and 1-bit-registers located in the pivot cells as well as $\frac{n(n+1)}{2}$ k-bit registers, $\mathbb{F}_{2^k}$-multipliers, and adders contained in the basic cells. For the TSL, we require an additional amount of $\frac{n(n+1)}{2}$ k-bit registers realizing the delay in vertical direction. The TSA again induces an overhead of $\frac{n(n+1)}{2}$ k-bit and 1-bit registers in comparison to the TSL. For TSA, TSL, and TSN we have a time complexity of $4n$, $3n$, and $2n$ clock cycles, respectively, in the non-pipelined case and n clock cycles in the other case.

Compared to the triangular architectures, the number of processing cells is reduced from n pivot cells and $n(n+1)/2$ basic cells to 1 pivot cell and n basic cells for the linear systolic solvers. This translates to 1 $\mathbb{F}_{2^k}$-inverter and n k-bit registers, $\mathbb{F}_{2^k}$-multipliers, and adders. Furthermore, n multiplexers and n shift registers are needed. Simple shift registers are very cheap on FPGAs [4]. In comparison to the LSL, the LSA additionally induces an overhead of n k-bit and 1-bit registers for signal buffering, but the shift registers need one stage less. LSA and LSN both have the same time complexity of n^2. Unlike the triangular designs, they do not profit from pipelining. Note that the architecture is less time-area optimal (by a factor of $2n/(n+1)$) than the two-dimensional approach. However, in cases where the SLE solver engines are not in continu-

[4] To store the 720 bits of a 10x10 SLE over $\mathbb{F}_{2^8}$, we only need 80 LUTs on a Xilinx Spartan-3, because one LUT can store up to 16 consecutive shift bits.

Table 1
Theoretical performance of the presented architectures to solve an $n \times n$ SLE with uniformly random entries over $\mathbb{F}_{2^k}$

Architecture	Time (non-pipelined)	Time (pipelined)	Logic Complexity	Memory Complexity
GSMITH	$n + \frac{n}{1-2^{-k}}$	$\frac{n}{1-2^{-k}}$	$k^3 + n^2(k^2+k)$	n^2k
TSA	$4n$	n	$nk^3 + \frac{n^2}{2}(k^2+k)$	$\frac{3}{2}n^2k$
TSL	$3n$	n	$nk^3 + \frac{n^2}{2}(k^2+k)$	n^2k
TSN	$2n$	n	$nk^3 + \frac{n^2}{2}(k^2+k)$	$\frac{n^2}{2}k$
LSA	n^2	n^2	$k^3 + n(k^2+k)$	n^2k^*
LSL	n^2	n^2	$k^3 + n(k^2+k)$	n^2k^*

* The memory of the linear systolic designs can almost completely be realized as bulk shifting memory, e.g., as cheap shift registers on FPGAs.

ous use [6], the linear systolic architectures provide low-footprint alternatives to the triangular ones.

Finally, it should be noted that in the special case $\mathbb{F}_2$ no inverters are needed and multipliers reduce to simple AND gates. Hence, in this case the area required by all designs is dominated by registers and multiplexers.

5.2 Implementation Results for $\mathbb{F}_2$

Tables 2 and 3 present our FPGA implementation figures for the case $\mathbb{F}_2$. They show the number of slices, lookup-tables (LUTs), and flip-flops (FFs) occupied by the designs as well as the number of gate equivalents (GEs) estimated by the tool. The maximal frequency at which an architecture can be operated is also given along with its runtime for solving $n \times n$ SLEs provided that input can ($\mathrm{RT}_{\mathsf{pipe}}$) or cannot (RT) be provided in a pipelined fashion. To allow easy comparison of the designs, the last column of the table contains the time-area product (TA) for the pipelined case defined by

$$\mathrm{RT}_{\mathsf{pipe}} \cdot \mathrm{Slices}\,. \tag{1}$$

We now consider some characteristics of the various architectures starting with GSMITH. As one can see from the tables, the number of slices occupied by GSMITH is equal to about $1.5 \cdot n^2$ on the Spartan-3 and $0.55 \cdot n^2$ on the Virtex-5 device. In this way, a Spartan-3 can host a GSMITH architecture up to a dimension of about 93×93 cells whereas the Virtex-5 can handle dimensions up to 114×114. Note that the operating frequency behaves more stable on the Virtex-5 than on the Spartan-3. For instance, for $n = 90$ we can operate GSMITH at 88% of the maximal frequency of the Virtex-5 whereas on the Spartan-3 only about 42% is reached.

For the triangular systolic architectures we can make the following observa-

Table 2
FPGA implementation results for $\mathbb{F}_2$ on Xilinx Spartan-3 XC3S1500 (13,312 Slices, 26,624 4-bit-LUTs/FFs, 300 MHz)

	n	Slices	LUTs	FFs	GEs	Freq. [MHz]	RT [μs]	RT_{pipe} [μs]	TA
GSMITH	20	606	1,176	434	10,531	178	0.34	0.22	133
	50	3,731	7,384	2,574	64,899	150	1.00	0.67	2,500
	90	12,239	24,161	8,239	210,884	127	2.13	1.42	17,379
TSA	20	820	251	817	9,509	262	0.31	0.08	66
	50	5,050	1,376	5,047	52,259	262	0.76	0.19	960
	80	12,880	3,481	12,877	129,209	262	1.22	0.31	3,993
TSL	20	370	272	437	6,475	234	0.27	0.09	33
	50	2,082	1,507	2,597	33,145	232	0.65	0.22	458
	90	6,537	4,693	8,277	100,341	224	1.21	0.40	2,615
TSN	20	159	253	227	3,521	34	1.18	0.59	94
	50	793	1,415	1,322	19,493	13	7.69	3.85	3,053
	90	2,340	4,370	4,182	60,423	7	25.71	12.86	30,092
LSA	20	100	81	99	3,609	300	1.33	1.33	133
	50	249	251	249	12,209	300	8.33	8.33	2,074
	90	630	721	449	39,249	250	32.40	32.40	20,412
LSL	20	44	85	60	3,321	300	1.34	1.34	59
	50	157	259	150	11,465	253	9.89	9.89	1,552
	90	369	735	270	37,901	221	36.66	36.66	13,528

tions: Except for very small SLEs, the systolic network architecture is not competitive here with regard to speed. Furthermore, it is worth mentioning that on the high-performance Virtex-5 FPGA we can clock the TSA as well as the TSN architecture at the maximal frequency even for relatively large dimensions. On the Spartan-3, we have an average cell area of about 0.7, 1.7, and 3.7 slices for the TSN, TSL, and TSA architecture, respectively, leading to a maximal SLE dimension of about 193×193, 123×123 and 83×83. The Virtex-5 can handle maximal dimensions of about 168×168, 142×142, and 101×101 where a cell occupies about 0.5, 0.7, and 1.4 slices on average.

The linear systolic architectures can also be clocked at maximum frequency on the Virtex-5. For larger designs, the clock frequency degrades for the Spartan-3. Due to the usage of shift registers, the overall area consumption grows more than linearly in n, but significantly slower than quadratically.

Conventional FPGAs exhibit many features not usable for the designs presented in this paper. (One exception are the shift registers used by the LSA and LSL engines allowing for a significant area-reduction.) Furthermore, the FPGA's inherent overhead to maintain its programmability additionally prevents from optimal time-area efficient designs. Hence, in order to obtain high-

Table 3
FPGA implementation results for $\mathbb{F}_2$ on Xilinx Virtex-5 XC5VLX50 (7,200 Slices, 28,800 6-bit-LUTs/FFs, 550 MHz)

	n	Slices	LUTs	FFs	GEs	Freq. [MHz]	RT [μs]	RT_{pipe} [μs]	TA
GSMITH	20	237	755	429	8,717	550	0.11	0.07	17
	50	1,413	4,867	2,557	54,525	539	0.28	0.19	268
	90	4,504	16,035	8,198	177,829	487	0.55	0.37	1,666
	110	6,263	23,995	12,218	265,709	391	0.84	0.56	3,507
TSA	20	363	479	840	12,493	550	0.15	0.04	15
	50	1,727	2,699	5,100	65,743	550	0.36	0.09	155
	90	5,804	8,459	16,380	201,143	550	0.65	0.16	929
TSL	20	161	460	460	9,320	550	0.11	0.04	6
	50	912	2,650	2,650	45,800	550	0.27	0.09	82
	90	3,082	8,370	8,370	136,440	550	0.49	0.16	493
TSN	20	160	252	250	3,764	102	0.39	0.20	32
	50	715	1,376	1,375	20,632	39	2.56	1.28	915
	90	2,045	4,281	4,275	64,167	21	8.57	4.26	8,712
LSA	20	55	81	101	3,764	550	0.73	0.73	40
	50	171	251	251	15,865	550	4.55	4.55	778
	90	291	541	451	40,065	550	14.73	14.73	4,286
LSL	20	33	82	61	3,482	550	0.73	0.73	24
	50	78	252	151	15,072	550	4.55	4.55	355
	90	116	542	272	38,640	550	14.73	14.73	1,709

performance SLE solvers, an ASIC realization should be considered which can be tweaked completely to fit the requirements.

5.3 Implementation Results for $\mathbb{F}_{2^8}$

We implemented all architectures for solving SLEs over $\mathbb{F}_{2^8}$ of different dimensions. Tables 4 and 5 present the results of these implementations in terms of area (Slices, FFs, LUTs), time (operating frequency, runtime in non-pipelined and pipelined mode), and time-area product (as defined by Equation (1)).

As one can see, on the Virtex-5 we can operate GSMITH at about 30% of the maximal frequency of the FPGA whereas on the Spartan-3 only a ratio of about 20% can be reached. Regarding area consumption, the Virtex-5 can host the GMSITH design up to an SLE dimension of 23×23 and the Spartan-3 can handle a maximal dimension of about 20×20. We have an average cell area of about 14 slices for the former and 32 slices for the latter FPGA. Note that in comparison to the $\mathbb{F}_2$-case, the average cell area has been increased by

a factor greater than 20.

As expected, the TSN exhibits very low clocking frequencies due to the long critical paths, even for the small 10×10 SLE. It should be considered a case study in this context. The operating frequencies of the TSA/LSA and TSL/LSL architectures reach approximately 41% and 24% of the top speed on the Spartan-3 and 60% and 38% for the Virtex-5. For the considered range of SLE dimensions the reached clock frequencies stay relatively constant. With regard to area consumption, the Spartan-3 can handle the TSA, TSL, and TSN designs up to a dimension of about 24×24, 25×25, and 26×26, respectively. Counting the delay elements as part of the systolic cells, we obtain an average cell area of approximately 38, 36, and 34, respectively. In the case of the Virtex-5, we have maximal dimensions of about 25×25, 26×26, and 27×27 and an average area consumption per cell of about 20.4, 18.2, and 17.5 for the TSA, TSL and TSN architectures. The best performance in terms of time (pipelined case) as well as time-area is achieved by the TSA, as expected. Of course, this comes at the price of a higher number of registers and slices.

For the linear systolic architectures, the LSA achieves the highest clock frequency and the best overall performance. The area consumption is very comparable in both cases. Hence, the only favorable point about the systolic lines approach in this case is that it takes full equations at a clock cycle, avoiding the skewing of matrices. The area consumption is almost linear in n, due to the efficient usage of memory and the much larger logic when compared to $\mathbb{F}_2$. The maximal achievable dimensions exceed 150×150 on both platforms. Compared to the triangular architectures, the linear approaches feature comparable clock frequencies at a much smaller resource count, but a much higher time-area product (for larger n).

6 Final Comparisons and Conclusions

Considering the presented designs we can make several general observations. The systolic approaches feature the advantage that their running times are independent of the characteristics of the coefficient matrix whereas for the GSMITH architecture only expected running times can be given. However, for matrices with uniformly distributed entries, frequently appearing in cryptographic applications, GSMITH (along with the TSN architecture) has the lowest asymptotic time complexity ($2n$) among all considered architectures when a continuous input stream cannot be provided. In contrast to that, the triangular and linear systolic arrays have the highest asymptotic running times in this case, but due to the fact that their critical path is independent of the problem size, they allow for higher frequencies compared to all other designs. This makes the design especially attractive for large values of n. The

Table 4
FPGA implementation results for $\mathbb{F}_{2^8}$ on Xilinx Spartan-3 XC3S1500 (13,312 Slices, 26,624 4-bit-LUTs/FFs, 300 MHz)

	n	Slices	LUTs	FFs	GEs	Freq. [MHz]	RT [μs]	RT_{pipe} [μs]	TA
GSMITH	10	3,525	6,809	895	48,185	61	0.33	0.16	564
	12	4,942	9,532	1,265	67,510	60	0.40	0.20	988
	20	13,310	24,821	3,386	176,017	55	0.73	0.36	4,792
TSA	10	2,431	4,262	1,386	37,624	125	0.32	0.08	193
	12	3,386	5,908	1,965	52,347	125	0.38	0.10	338
	20	8,903	15,132	5,281	135,199	125	0.64	0.16	1,424
TSL	10	2,296	4,405	904	34,986	69	0.43	0.14	321
	12	3,205	6,171	1,279	49,034	69	0.52	0.17	545
	20	8,454	16,151	3,433	128,528	68	0.88	0.29	2,452
TSN	10	2,251	4,379	461	30,272	8	2.50	1.25	2,814
	12	3,110	6,071	653	42,013	7	3.43	1.71	5,318
	20	7,384	14,390	1,723	100,604	4	10.00	5.00	36,920
LSA	10	502	889	299	12,441	126	0.80	0.80	402
	12	584	1,054	358	14,840	125	1.15	1.15	672
	20	1,050	1,870	598	34,696	125	3.20	3.20	3,360
LSL	10	538	984	241	13,180	73	1.36	1.36	732
	12	684	1,260	290	15,721	73	1.97	1.97	1,347
	20	1,082	2,092	482	35,289	70	5.71	5.71	6,178

triangular systolic network has the lowest asymptotic time complexity of all systolic approaches, but at the expense of pretty low operating frequencies making the design unattractive already for small values of n. The linear systolic architectures exhibit the lowest area consumption, but at the cost of a quadratic execution time. They should be considered when area is of higher importance than throughput. Among the high performance engines, the triangular systolic architectures are still smaller than GMSITH. The triangular systolic network performs best in this respect, followed by the systolic lines and array architectures. Note that the GSMITH design and the slower linear systolic designs require only a single $\mathbb{F}_{2^k}$-inverter (contained in the pivot cell) whereas the triangular systolic architectures comprise n such inverters. This can become an advantage when considering SLEs over large extension fields where inverters are very expensive in terms of area and latency. However, which architecture eventually provides the best performance highly depends on the concrete application scenario and implementation platform.

With regard to our concrete implementation results, we can draw the following conclusions: For SLEs of the considered sizes ($10 \leq n \leq 20$) over $\mathbb{F}_{2^8}$, the triangular systolic array exhibits the lowest absolute running time of all architectures. In general, the TSA shows the best tradeoff between area and

Table 5
FPGA implementation results for $\mathbb{F}_{2^8}$ on Xilinx Virtex-5 XC5VLX50 (7,200 Slices, 28,800 6-bit-LUTs/FFs, 550 MHz)

	n	Slices	LUTs	FFs	GEs	Freq. [MHz]	RT [μs]	$\mathrm{RT}_{\mathsf{pipe}}$ [μs]	TA
GSMITH	10	1,562	4,301	895	37,267	164	0.12	0.06	94
	12	2,238	6,269	1,265	54,003	164	0.15	0.07	157
	20	5,748	17,134	3,386	147,026	164	0.24	0.12	690
TSA	10	1,356	3,705	1,395	38,305	333	0.12	0.03	41
	12	1,832	5,154	1,974	53,322	333	0.14	0.03	55
	20	4,937	13,268	5,290	137,616	333	0.24	0.06	148
TSL	10	1,249	3,782	900	34,884	208	0.14	0.05	62
	12	1,641	5,175	1,272	47,853	208	0.17	0.06	98
	20	4,165	13,872	3,400	126,724	209	0.29	0.10	417
TSN	10	1,157	3,479	460	28,033	17	1.18	0.59	683
	12	1,596	4,885	648	39,379	13	1.85	0.92	1,468
	20	4,025	10,965	1,720	90,515	12	3.33	1.67	6,722
LSA	10	277	749	331	17,571	333	0.30	0.30	83
	12	275	889	397	21,015	333	0.43	0.43	118
	20	704	1,430	661	34,658	333	1.20	1.20	845
LSL	10	381	887	241	17,817	200	0.50	0.50	191
	12	339	1,051	289	21,285	200	0.72	0.72	244
	20	657	1,727	481	35,297	182	2.20	2.20	1,445

execution time. The linear systolic arrays are advantageous only if area is constrained and a continuous input of equations cannot be guaranteed. For the chosen parameters, the area consumption is still dominated by the logic, resulting in an almost linear area consumption on FPGAs. Consequently, linear systolic designs are a good choice if the SLEs become large. With regard to the product of occupied area, the running time in the non-pipelined mode as well as the running time in the pipelined mode, the TSA seems to be the SLE solver of choice for the considered values of n.

For SLEs of the considered dimensions over $\mathbb{F}_2$, the TSL architecture has the best time performance when input is provided in a non-pipelined fashion. However, for bigger values of n, the triangular systolic array will eventually perform better in that respect, due to the persistency of the operating frequency. The TSA architecture also exhibits the best running time in the pipelined-mode but already for values of n in the considered range. Interestingly, over $\mathbb{F}_2$ the GSMITH design exhibits a slightly lower area consumption than a TSA, and the TSL architecture is significantly smaller than both. Like before, the linear systolic designs are a good choice only for large SLEs or if area is constrained. The resource consumption shows a stronger increase for growing n than in the case of $\mathbb{F}_{2^8}$, since the memory contributes stronger to the overall resource

consumption. When weighting area consumption and both types of running times equally the TSL approach seems to be best suited for $20 \leq n \leq 90$.

Due to their high flexibility, the systolic architectures can be adjusted to meet the performance criteria of different applications in an optimal way. Consequently, they outperform the GSMITH architecture in most application scenarios. The variety of the presented engines along with their evaluation should help implementers to choose and adapt the right engine for their designs.

Acknowledgements. The authors would like to thank Christof Paar for making this work possible. During the work on this article, the authors were supported partially by the Chair for Embedded Security at the Ruhr-University of Bochum, Germany. Andrey Bogdanov was supported in part by a visiting postdoctoral fellow grant from the Fund for Scientific Research - Flanders (FWO) within the FWO research project "Linear codes and cryptography" G.0317.06, by the Research Fund K.U.Leuven grant (OT/08/027) "A mathematical theory for the design of symmetric primitives", and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

References

[1] M. Albrecht and C. Cid. Algebraic Techniques in Differential Cryptanalysis. In O. Dunkelman, editor, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2009.

[2] S. Balasubramanian, H. W. Carter, A. Bogdanov, A. Rupp, and J. Ding. Fast Multivariate Signature Generation in Hardware: The Case of Rainbow (Poster). In *ASAP*, pages 25–30. IEEE Computer Society, 2008.

[3] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 600–616. Springer, 2003.

[4] A. Bogdanov. Linear Slide Attacks on the KeeLoq Block Cipher. In D. Pei, M. Yung, D. Lin, and C. Wu, editors, *Inscrypt*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[5] A. Bogdanov, T. Eisenbarth, and A. Rupp. A Hardware-Assisted Realtime Attack on A5/2 Without Precomputations. In Paillier and Verbauwhede [21], pages 394–412.

[6] A. Bogdanov, T. Eisenbarth, A. Rupp, and C. Wolf. Time-Area Optimized Public-Key Engines: MQ-Cryptosystems as Replacement for Elliptic Curves? In E. Oswald and P. Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2008.

[7] A. Bogdanov, I. Kizhvatov, and A. Pyshkin. Algebraic Methods in Side-Channel Collision Attacks and Practical Collision Detection. In D. R. Chowdhury, V. Rijmen, and A. Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2008.

[8] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Paillier and Verbauwhede [21], pages 450–466.

[9] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *FCCM*, pages 237–248. IEEE Computer Society, 2006.

[10] N. Courtois and G. V. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In S. D. Galbraith, editor, *IMA Int. Conf.*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2007.

[11] N. Courtois, G. V. Bard, and D. Wagner. Algebraic and Slide Attacks on KeeLoq. In K. Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[12] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In B. Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.

[13] J. Ding and D. Schmidt. Rainbow, a New Multivariable Polynomial Signature Scheme. In J. Ioannidis, A. D. Keromytis, and M. Yung, editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.

[14] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra*, 139:61–88, 1999.

[15] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F5). In T. Mora, editor, *ISSAC*, pages 75–83. ACM, 2002.

[16] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *SC*, page 3. IEEE Computer Society, 2005.

[17] W. Geiselmann, K. Matheis, and R. Steinwandt. PET SNAKE: A Special Purpose Architecture to Implement an Algebraic Attack in Hardware. Springer Transactions on Computational Science, Special Issue on "Security in Computing" (to appear). Extended version available on eprint: `http://eprint.iacr.org/2009/222`.

[18] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[19] B. Hochet, P. Quinton, and Y. Robert. Systolic Gaussian Elimination over GF(p) with Partial Pivoting. *IEEE Trans. Computers*, 38(9):1321–1324, 1989.

[20] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute of Expermiental Mathematics, University of Essen, Germany, 1994.

[21] P. Paillier and I. Verbauwhede, editors. *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*. Springer, 2007.

[22] D. Parkinson and M. Wunderlich. A Compact Algorithm for Gaussian Elimination over GF(2) Implemented on Highly Parallel Computers. *Parallel Computing*, 1:65–73, 1984.

[23] H. Raddum and I. Semaev. Solving Multiple Right Hand Sides Linear Equations. *Des. Codes Cryptography*, 49(1-3):147–160, 2008.

[24] M. Renauld, F.-X. Standaert, and N. Veyrat-Charvillon. Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2009.

[25] C.-L. Wang and J.-L. Lin. A Systolic Architecture for Computing Inverses and Divisions in Finite Fields GF(2^m). *IEEE Trans. Computers*, 42(9):1141–1146, 1993.

[26] G. Williams. *Linear Algebra with Applications*. Jones and Bartlett, 2005.

[27] B.-Y. Yang, J.-M. Chen, and N. Courtois. On Asymptotic Security Estimates in XL and Gröbner Bases-Related Algebraic Cryptanalysis. In J. Lopez, S. Qing, and E. Okamoto, editors, *ICICS*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.