

Hardware Evaluation of eSTREAM Candidates:

Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt

Frank K. Gürkaynak¹, Peter Luethi¹, Nico Bernold², René Blattmann²,
Victoria Goode², Marcel Marghitola², Hubert Kaeslin³,
Norbert Felber¹ and Wolfgang Fichtner¹

⁽¹⁾ Integrated Systems Laboratory, ETH Zurich, CH-8092 Zurich

⁽²⁾ Dept. of Information Technology and Electrical Engineering, ETH Zurich, CH-8092 Zurich

⁽³⁾ Microelectronics Design Center, ETH Zurich, CH-8092 Zurich

Abstract. One important requirement imposed on all eSTREAM stream cipher candidates was to show the potential to be superior to the AES in at least one significant aspect. We present hardware implementation results of eight different eSTREAM Profile-II candidates, all integrated in 0.25 μm 5-Metal CMOS technology. The goal of this work was to provide a fair base for comparison of different hardware crypto algorithms. Additionally, an AES core optimized for stream cipher output has been implemented and is listed as comparative reference.

1 Introduction

The European Network of Excellence for Cryptography (ECRYPT) has started a multi-year effort called eSTREAM to identify new stream ciphers that might become suitable for widespread adoption. A total of 34 algorithms have been submitted to eSTREAM. Nine candidates (ABC, CryptMT, DICING, Dragon, Frogbit, HC-256, Mir-1, Py, and SOSEMANUK) have been specified as pure software implementations, and a further 13 (F-FCSR, Hermes8, LEX, MAG, NLS, Phelix, Polar Bear, POMARANCH, Rabbit, Salsa20, SSS, TRBDK3 YAEA, and Yamb) have been specified to be suited for both software and hardware implementations. The remaining 12 algorithms (Achterbahn, DECIM, Edon80, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, TSC-3, VEST, WG, and ZK-Crypt) were designed with primarily hardware implementations in mind.

The *Integrated Systems Laboratory* (IIS), together with the *Microelectronics Design Center*, provides a series of lectures on VLSI design at the *Department of Information Technology and Electrical Engineering* (D-ITET) of the *ETH Zurich*. As part of this lecture, students are encouraged to work on projects where they design their own ASICs. Successful implementations are then sent to fabrication, and the manufactured chips are finally tested during a later semester. Since a lot of cryptographic algorithms are developed with hardware realizations in mind, they are very well suited for such semester theses. As a consequence, a number of successful projects were realized at our institute over the years [1,2,3].

For the winter semester 2005/2006, four students showed interest in a project targeting the implementation of cryptographic hardware. It was decided to design a subset of eSTREAM candidates and thereby to provide a fair comparison (of at least the implemented set) of candidate algorithms. Since the entire IC design had to be completed within one semester (14 weeks), not all 34 candidate algorithms could be realized with reasonable effort. According to the advice of Elisabeth Oswald, Thomas Johansson and Matt Robshaw [4], the following guidelines were adopted in order to reduce the number of algorithms suitable for integration within this project. Consider only:

1. Algorithms that were specifically intended for hardware realization (eSTREAM Profile-II candidates).
2. Algorithms that were not known to have any negative cryptological or technical issues.
3. Algorithms for which future development is more likely to be expected.

At the start of the project in October 2005, the decision for a subset of stream cipher algorithms had to be made, and eventually seven eSTREAM candidate algorithms were sorted out: Grain[5], MICKEY[6], MOSQUITO[7], SFINKS[8], Trivium[9], VEST[10], ZK-Crypt[11]. By the time when all of these seven algorithms were successfully implemented in hardware, little time was still left. At this stage, it was decided to briefly revise the remaining five Profile-II algorithms (Achterbahn, DECIM, Edon80, TSC-3, WG), and considered to implement additional algorithms if this could be achieved with reasonable effort. As a result of this procedure, Achterbahn[12] was added to the list of implemented algorithms.

To provide a comparative reference for the results, the well-known *Advanced Encryption Standard* (AES) [13] block-cipher was implemented in Output-Feedback (OFB) mode. In this configuration mode, the block-cipher is able to generate a continuous output stream that can be used as a stream-cipher. Since we have significant experience in implementing the AES algorithm at the IIS, we were able to efficiently customize an AES block for stream-cipher implementation. The customized AES core and the eight stream-cipher designs were then integrated in 0.25 μm CMOS technology.

The organization of the paper is as follows: Section 2 describes the methodology used in designing all circuits. The algorithms are briefly described in section 3. A brief discussion about *hardware efficiency* can be found in section 4 and the implementation results are presented in section 5. Finally, the conclusions are drawn in section 6.

2 Methodology

The comparison of hardware implementations of different algorithms is a difficult and challenging task. Most eSTREAM candidate submissions contain information regarding the hardware implementation of the algorithm. While the presented information is certainly valuable, it is difficult to use the data directly to compare different algorithms with each other. The reasons are as follows:

- The implementations may use different design styles, heavily depending on the type of target hardware: FPGA or ASIC. For ASIC design flows, we can usually take advantage of quite a fine-grained logic optimization enabled by dedicated synthesis tools. As a consequence, this allows for deep logic structures, or in other words, more logic functionality can be executed during a single clock cycle. On the opposite, FPGAs contain dedicated macro structures (e.g. logic slices, multipliers), which allow only for coarse-grained optimization. Most often, the interconnect delay significantly adds up to the overall timing of the final placed and routed FPGA design. Moreover, memory resources are in general quite costly on ASICs, while on FPGAs, large memories are rather common. In order to meet high throughput constraints, this leads naturally to different design styles: one relying on fine-grained logic optimization, the other on shallow logic depth and increased memory usage.
- For ASIC implementations, different manufacturing technologies may have been used, while for FPGAs different types of programmable devices may have been chosen. For instance, ASIC designs may differ in process technology or macro cell library (e.g. subset of fixed-size memories vs. RAM compiler), whilst FPGA architectures may employ specialized blocks such as multipliers or DSP slices. Therefore, it may not be obvious how algorithms realized on different hardware technologies can be compared and how they would fare on identical technology.
- The experience of the designer and the project schedule may play an important factor on how well an algorithm is mapped to hardware.

2.1 Design Flow

This project aims at providing a fair comparison between different algorithms, all of them implemented using a standard cell based ASIC design flow. The target technology for the chip integration is UMC 0.25 μm 5-Metal CMOS technology. Four seventh semester master students (Nico Bernold, Rene Blattmann, Victoria Goode, Marcel Marghitola) worked in two groups to implement the algorithms in VHDL. The students were supervised by two research assistants (Frank K. Gürkaynak, Peter Lüthi) with experience in the entire ASIC design flow. The VHDL source code was functionally verified using the *Mentor Graphics ModelSim* simulation environment. The C-code from the eSTREAM

submission package has been used as a golden model for verification. The circuit was synthesized using *Synopsys Design Vision* tools and the resulting netlist was placed and routed using *Cadence Design Systems SoC Encounter* software. The students had 14 weeks to complete the entire design flow in order to meet a strict tape-out deadline. The chips are due back from manufacturing mid-may 2006.

2.2 Interface

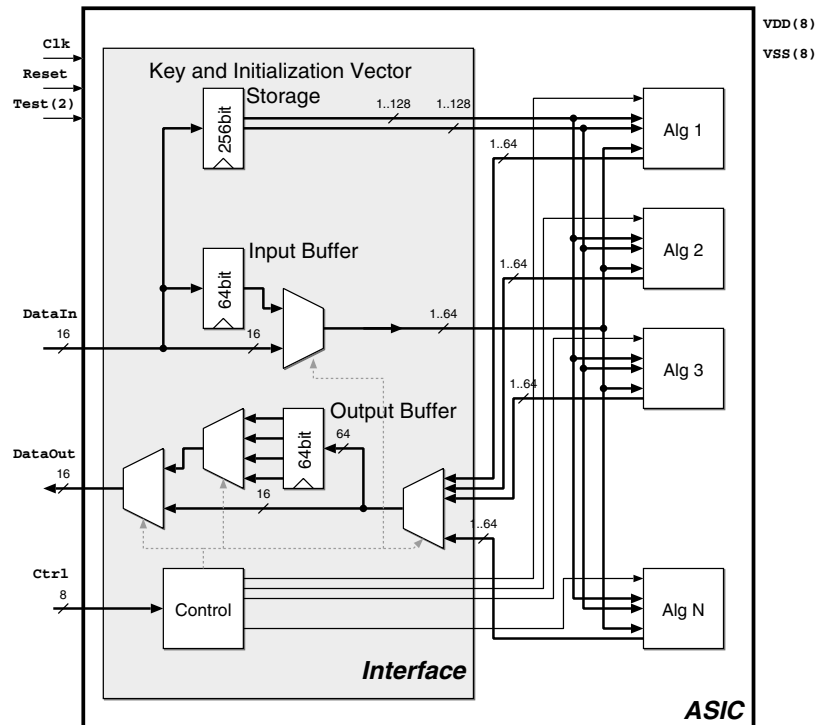


Fig. 1. Simplified block diagram showing the common interface used to access all algorithms.

The available resources for the physical implementation was limited by several constraints. Each group was assigned a die area of 5.92 mm^2 . For the technology used, this area is sufficient for an ASIC with 84 pins and a core area of 3.56 mm^2 . Since multiple algorithms had to be implemented on a single ASIC, a common interface as shown in figure 1 was developed. Due to a limited amount of I/O pads, the ASIC uses 16-bit input and output buses for data exchange, although some algorithms require more than 16 data bits per clock cycle. To satisfy the requirement for delivering more data, 64-bit buffers for both input and output have been added. Algorithms that require more than 16 bits of I/O data per clock cycle can be run using one of two options.

1. In the slow mode, the algorithm is halted until the input buffer has collected sufficient data. After accumulation of all data, the algorithm is run and the output is again collected at the output buffer. The algorithm is halted until the data is read out of the buffer. In this mode, all input data is used for en-/decryption.
2. In the fast mode, the algorithm is not paused. Instead, the missing input data is obtained by replication, and only a portion of the output is observed. This mode may be applied for speed testing.

The cipherkey and the initialization vector (IV) are stored in a common 256-bit register. This register is made available to all algorithms in parallel.

To provide an equal basis for comparison, the guidelines listed below were followed:

- Some submissions did not provide an associated authentication method. All algorithms were implemented without any authentication method add-on.
- All synthesized algorithms include scan-test structures for full-scan testing.
- No ROM macros were used for the look-up tables and/or complex functions.
- All algorithms were designed to accept plaintext and deliver ciphertext. Algorithms which only generate key streams were enhanced by adding XOR gates.

2.3 Cryptographic Security

We believe that our expertise resides mainly in the design of digital circuits. The discussion of security aspects of the implemented algorithms is therefore left to experts in cryptography. All algorithms have been assumed to be equally secure for performance comparison.

Once an otherwise secure cryptographic algorithm is implemented in hardware or software, it will acquire physical properties that can be observed. If it is possible to guess parts of the cipherkey by observing these physical properties, the hardware implementation is said to be vulnerable against side-channel attacks. The specific implementation of an algorithm may have a strong influence on how effective a given side-channel attack will be. However, there is no algorithm- and attack-independent methodology to rate the side-channel vulnerability of an implementation. Therefore, no remarks will be made on how vulnerable the implementations are against side-channel attacks. During the design phase, no special countermeasures against side-channel attacks have been considered and implemented. The side-channel security of the implemented algorithms will be determined by measurements on fabricated ASICs in a follow-up project.

2.4 Measuring Performance

The following performance metrics will be used in this paper:

– Circuit area (A)

A represents the total area that is required for the implementation, expressed in μm^2 . For reference, in the technology used for this implementation: a 2-input NAND gate occupies an area of $23.76 \mu\text{m}^2$, a 2-input XOR gate occupies $55.44 \mu\text{m}^2$ and a scannable flip-flop with reset occupies $205.92 \mu\text{m}^2$. The circuit area is obtained from synthesis results, and does not include buffers for clock distribution and additional overhead for placement and routing.

– Maximum clock rate (f)

The maximum clock rate, given in MHz, is determined by the critical path of the circuit. The number is once again taken from post-synthesis timing analysis. In the technology applied here, the fanout-of-four (FO4) delay [14] of a simple inverter is approximately 0.1 ns.

– Processed bits per clock cycle (*radix*)

Most of the submitted stream-cipher candidates have been specified with single bit output. For some algorithms, it is possible to modify the architecture in such a way that multiple output bits are calculated concurrently. Moreover, some algorithms like VEST have variations of the architecture for different output bit lengths. The number of bits simultaneously generated by the algorithm is referred to as the *radix* of the implementation. Some algorithms will have multiple implementations with different radices.

– Total throughput (T)

One of the fundamental parameters of a cryptographic algorithm is the amount of data it can process within a given period. The total throughput of the algorithm is expressed as Gbits/s and can be calculated from the previous parameters as:

$$T = f \times \text{Radix}$$

– Throughput per unit area (TpA)

Judging the performance purely by the throughput is not representative as this provides no indication about the area required for the implementation. For this purpose, the *throughput per unit area* measure will be used:

$$TpA = \frac{f \times Radix}{A}$$

3 Algorithms

In this section, specific comments about the eight implemented algorithms are given. These comments target mainly the *process of implementation* of the eight eSTREAM candidates, in other words, how straight-forward the actual implementation process was based on the provided documentation. Note that, for a hardware designer, the reference C-code is just as important as the written documentation. The implementation needs always to be verified against the reference C-code, and when in doubt, always the implementation in the C-code is assumed to be correct.

3.1 Achterbahn

As mentioned earlier, Achterbahn was not amongst the initial candidates for implementation. Once all intended algorithms were implemented, it was decided to briefly revise the remaining 5 algorithms to determine whether or not more could be implemented. Achterbahn was selected primarily because it is very well documented and has an excellent reference C-code, exactly the desired prerequisites for hardware designers.

Achterbahn can be configured to use initialization vectors (IV) of different bit-lengths. This flexibility comes at the expense of a more complex initialization sequence which also requires more hardware. Our implementation is therefore limited to support only a 64-bit IV.

While it is possible to implement higher *radix* versions of Achterbahn, doing so increases the critical path, hence reducing the efficiency of this approach. In principle the algorithm could be implemented employing any *radix* without major difficulties. Due to the initialization sequence, practical *radices* are limited to even dividers of 176.

3.2 Grain

Grain is an algorithm that is rather simple and straightforward to implement for *radices* up to 16. A *radix*-32 implementation is also possible, but would result in a longer critical path. At the start of the project (October 2005), there were two versions of Grain available. From a hardware performance point of view, there is no difference between the two versions. The submission package of Grain included good documentation and good reference C-code.

3.3 MICKEY

MICKEY is another compact algorithm that is very easy to implement. The documentation is written in a '*hardware designer friendly*' way and the reference C-code is also easy to follow. The only technical issue of this algorithm is the difficulty to increase the *radix*.

3.4 MOSQUITO

MOSQUITO is the only algorithm implemented that has separate encryption and decryption modes. There were several problems with the reference C-code. The initial submission was corrected in July 2005. However, this code still had some errors, which were finally corrected in December 2005. The accompanying documentation lacks precision for implementation.

MOSQUITO has a pipelined structure with very few gates in between registers. It is therefore difficult to modify the algorithm for higher *radix* implementations. Without the initialization sequence, a *radix*-9 implementation would theoretically be possible. However, the initialization sequence that uses 104 bits renders this impractical. We have implemented a *radix*-3 version of MOSQUITO.

The fifth-stage register is specified to be 53 bits wide. During synthesis of the algorithm, it was noticed that only 48 bits were used, the content of the 5 most significant bits was discarded.

3.5 SFINKS

SFINKS is mainly dominated by a multiplicative inverse function in $GF(2^{16})$. This is a relatively complex block that can be implemented by iteratively decomposing the function into operations in $GF(2^2)$. While the documentation contains an appendix explaining this process, especially for hardware designers that are not well versed with Galois field implementations, the required transformation is not trivial. In essence, the inverse function is a 16-bit input, 16-bit output function. However, during normal operation, only 1-bit is used to calculate the key stream (a further bit is used for the calculation of MAC, which was not implemented in this project). The full 16-bit output is only required during initialization. As explained in section 4.1, from a hardware design perspective, the system can be modified in a way where the initial states of the registers are directly loaded. If the algorithm is implemented in this way (we called this implementation SFINKS+), the *throughput per area* can be increased by more than 75%. Additionally, it is also less costly to increase the *radix* of SFINKS+.

The high logic complexity of the inverse function results in a relatively low *maximum clock frequency*. It can be increased by adding pipeline registers into the inverse function. Our implementation uses a single pipeline stage.

The documentation of SFINKS had some errors at the beginning, these were corrected later. Apart from the description of the inverse function, which is difficult to understand, the documentation is easy to follow.

3.6 Trivium

Trivium has a very simple structure that is well-suited for higher-*radix* implementations up to *radix*-64 without noticeable hardware penalties. In fact, from a *hardware efficiency* point of view, it is wasteful to implement Trivium with a *radix* less than 64. *Radix*-64 Trivium is just 54% larger, and has a *maximum clock frequency* that is only 10% lower than a *radix*-1 implementation. Consequently, the *throughput per area* of the *radix*-64 version is roughly 40 times higher compared to the *radix*-1 alternative.

The main problem with Trivium is the reference C-code, which does not have any comments. This made it extremely difficult to integrate it into the verification flow.

3.7 VEST

The initial documentation set of VEST was not very easy to follow, and was not very clear regarding the input permutations to the non-linear functions in the accumulator. It was later discovered that the documentation had been updated in the meantime, and we believe that the problems were addressed in this revision. The reference C-code is not well suited for understanding the algorithm, as it is cluttered with pre-processor commands.

VEST has been described in separate families of functions for 4, 16, and 32 bit output per clock cycle, called VEST4, VEST16 and VEST32 respectively. The new documentation also includes the 8-bit version VEST8.

The algorithm is fairly complex and has an equally complex initialization sequence. The majority of the functions are described as look-up tables. Describing the algorithm in VHDL is not a very trivial task. We have modified the reference code so that it generated output that we could include in the VHDL description itself. The large number of look-up tables might be suitable for FPGA implementations, since FPGAs realize functions within small look-up tables. Nevertheless, for a custom ASIC solution the approach of using look-up tables is cumbersome. In fact, using our standard design flow it was only possible to synthesize VEST4 and VEST16. We will have to re-write the code for VEST32 so that we can pass it through the synthesis stage.

3.8 ZK-Crypt

ZK-Crypt has by far the worst documentation of all eSTREAM candidates that we have implemented. First, there is no overview which makes it extremely difficult to follow. A multitude of drawings has been provided, but some drawings are marked as '*conceptual*' and seem to be inconsistent with the documentation. The reference C-code fares better, but is also far from easy to understand. At least in one case there is an inconsistency between the reference C-code and one of the drawings, regarding how key bits 26 and 27 are handled. We have implemented the algorithm in such a way that is consistent with the reference C-code (which simply ignores the content of these two bits). From a hardware designer's point of view, this peculiarity might originate from an incomplete C-code, what would also result in non-exhaustive functional pattern generation for hardware verification once the functionality of these two bits is implemented. We decided to stick to the original C-code and to ignore any ambiguous information for the hardware design.

The algorithm is very difficult to implement, due to its irregular structure and many details, especially in the control state machines. We made no attempt to try different *radix* implementations. On the positive side, the algorithm does not have an initialization sequence and uses no IV.

3.9 Reference AES implementation

To serve as a reference, we have implemented an AES core that is configured to run in *output feedback mode* (OFB). The core accepts 128-bit keys, and uses an on-the-fly roundkey generator. It has 4 parallel look-up tables for the *SubBytes* function, and requires 41 clock cycles to compute a 128 bit output that is used as the key stream (resulting in a calculated *radix* of 3.12). For an independent implementation, the AES core would also have to store the cipherkey so that it can restart generating the roundkeys for each encryption. In this implementation, the cipherkey is stored in the interface which results in a slightly more compact realization (about 10% less circuit area).

4 Efficiency in Hardware

4.1 Initialization

Several eSTREAM candidates require an initialization phase. During the initialization phase, the internal registers are preset to a certain value, and the cipherkey and the initialization vector are loaded into specified registers. Some eSTREAM candidates require a number of operational cycles that will initialize all internal registers prior to generating the key stream.

From a cryptographic point of view, it may be important to differentiate between cipherkey and the initialization vector. However, from a VLSI designers point of view, only the internal registers responsible for key stream generation during the en-/decryption process need to be set to an initial state. This initial state of the registers can be derived mathematically from the cipherkey and the initialization vector and may then be loaded directly into the registers, saving precious setup time before being able to process any data. Moreover, the streamlining of the initialization procedure might also reveal some benefits in terms of hardware complexity: Basically, the control overhead and data switching through multiplexers is reduced, what leads to minor improvements in clock speed and area. But in rare cases, the optimization of the initialization procedure may result in a significantly improved *hardware efficiency*:

As an example, in SFINKS, the initialization routine requires a 16-bit multiplicative inverse which must be delayed by 6 clock cycles. The 16-bit output of the delay buffer implemented as 96 flip-flops (FF) is then fed back to the LFSR. This function is only required for the initialization procedure, during normal operation only the LSB output of the multiplicative inverse is used. If the algorithm is modified so that the initial state is calculated externally and loaded directly onto the hardware as seen in figure 2, the inverse function can be simplified and the delay elements can be substantially reduced as well. In this way, the *throughput per area* of SFINKS can be increased by more than 70%. This major improvement in *hardware efficiency* originates from both, reduction in circuit area and increase in clock speed.

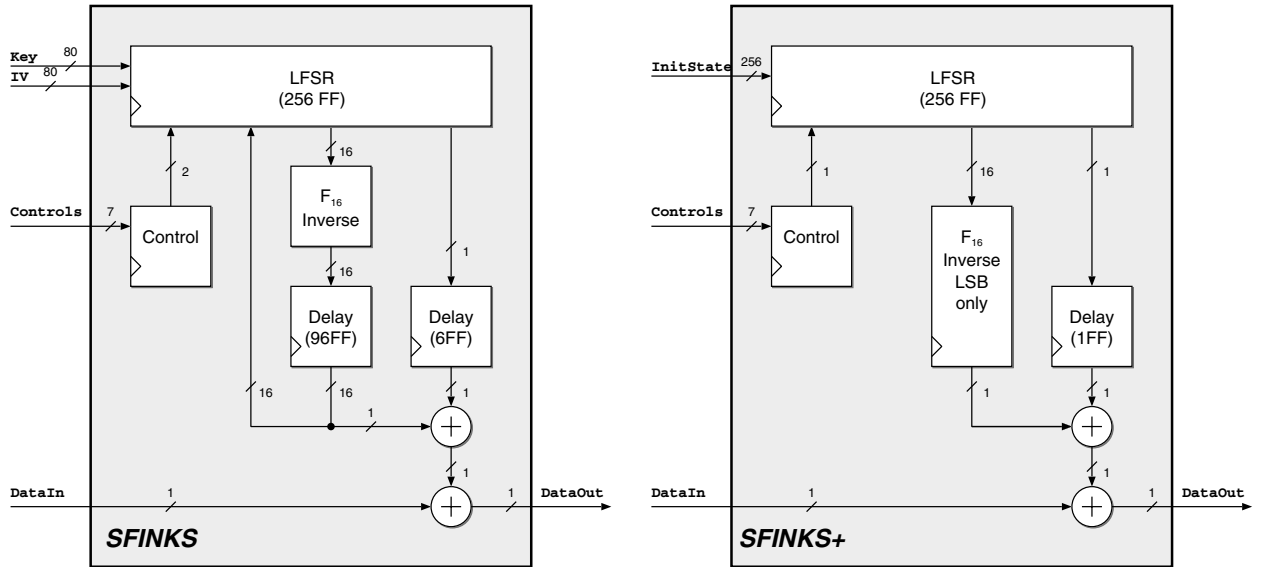


Fig. 2. Native implementation of SFINKS (left), as suggested by the original eSTREAM candidate submission, has significant overhead for initialization. SFINKS+ (right) does not have this overhead and is even more efficient in terms of circuit area, data throughput and initialization latency.

4.2 Stage delay

Synchronous digital circuits for ASICs are in general built from standard cell libraries. The elements in standard cell libraries are classified in logic and sequential cells. Sequential cells, such as flip-flops and latches, serve for storage of data, while logic cells are necessary to reflect the mathematic functions in hardware.

The *maximum clock frequency* of the circuit is determined by the longest path induced by numerous logic cells between two sequential elements in the circuit. Each logic cell (or *gate*) in the critical path will contribute some delay to the signal propagating through. For a simplified analysis, one can assume that all gates have a technology-dependent unit delay. For such analyses the FO4 delay is frequently used [14]. In this simplified analysis, the clock frequency can be expressed in terms of FO4 gate delays. This allows for simple extrapolation of circuit performance in other technologies.

State-of-the-art high performance digital circuits can be designed with as little as 10-20 FO4 delays. However, such designs require utmost precision in the back-end design phase (the *physical* design process: cell placement, routing and clock distribution) and are most often hand crafted. The back-end overhead for 20-50 FO4 delay circuits is still significant. It is a very challenging task to implement these circuits using standard cells. Circuits with roughly 50-100 FO4 delays are fast designs that are manageable with standard cell design methodologies, and implementing circuits with 100-200 FO4 delays is a straightforward procedure. Finally, circuits with more than 200 FO4 delays hardly pose timing related challenges.

For the UMC 0.25 μm technology, the FO4 delay is approximately 0.1 ns. Consequently, designs with up to 200 MHz can be realized without excessive overhead. Circuits that can be clocked faster can still be implemented, but they pose significant challenges to the back-end design process and are rarely practical.

4.3 Bits per clock cycle

There are two fundamental options that can be employed to improve the throughput of a cryptographic circuit. Either the *maximum clock frequency* can be raised or the *radix* of the circuit is increased. As already explained before, the

increase of the frequency is only viable to a certain extent. Nevertheless, a higher *radix* remains as second option and in general turns out to be a very powerful method to boost the overall throughput.

Not all algorithms are equally suited to produce multiple output bits per clock cycle. Some algorithms require replication of major operational blocks in order to increase the *radix*, and thus lead to an enlarged circuit area. Basically, if the n -fold increase in the *radix* requires n -fold increase in the area, both implementations would have a similar *throughput per area*, and thus would be equally efficient. Moreover, such changes increase often the critical path and decrease the maximum operating frequency as well. However, as can be seen in figure 3, several eSTREAM candidates have been designed with support for multiple-bit computation in mind. The performance of these algorithms can be improved considerably by increasing the *radix*.

Grain is an illustrative example for the typical VLSI challenge of trading in *throughput* against *circuit area*: If more throughput is required, the *radix* might be doubled, but the resulting gain is not two-fold since the *circuit area* slightly grows and/or the *clock frequency* drops. Trivium is an extraordinary example where doubling the *radix* has almost no impact on the *area*, and the *clock frequency* can be sustained. Therefore, the achieved gain is nearly doubled and almost at the ideal curve. On the other hand, Achterbahn represents an implementation, which is not appropriate for architectural changes in *radix* in order to achieve a higher throughput. The best *hardware efficiency* is obtained with *radix*-2, increasing the *radix* further even degrades the efficiency of Achterbahn. This is because both values are nearly equally affected, the *circuit area* grows and the *clock frequency* degrades. From an efficiency point of view, it is not advisable to implement any other version than *radix*-2. For higher throughput rates rather than using higher-*radix* implementations, replicating several *radix*-2 versions of Achterbahn would be more efficient.

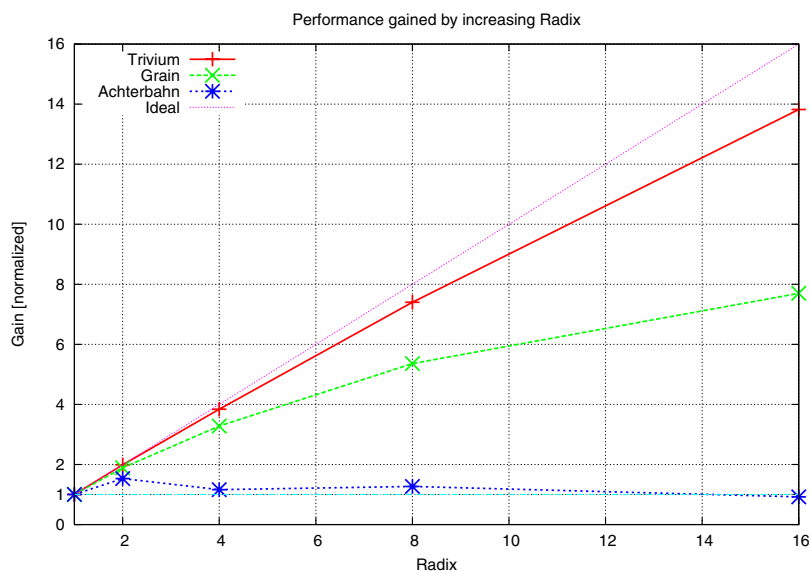


Fig. 3. Performance gained by increasing the area. Performance is expressed in terms of throughput per area, and is normalized to the *radix*-1 implementation of each algorithm.

5 Results

The numbers listed here are synthesis results. Post-layout results, including power figures, will be presented at the SASC 2006 workshop.

As a first step, all algorithms have been implemented to match their description. Apart from VEST and ZK-Crypt, this results in *radix*-1 implementations which have *throughputs* at around 0.3 Gbit/s. When compared to the reference

AES implementation, *radix*-1 algorithms with smaller *area* (Grain, MICKEY, Trivium) achieve a higher *throughput per area* ratio, while algorithms that require more *area* (Achterbahn, MOSQUITO and SFINKS) can not match the performance of AES. Both VEST and ZK-Crypt, which have a higher *radix* by definition, are able to outperform AES implementation noticeably.

As a second step, we tried to optimize all algorithms in order to increase their performance. In most cases, significant performance gains can be obtained by increasing the *radix*. Especially Trivium, which has been designed with parallelization in mind, reaches an exceedingly high *throughput*. Table 1 compares the main performance figures for all algorithms. For algorithms that have multiple implementations, only the one with the highest *throughput per area* is listed. A graphical comparison of the results are given in figure 4 as well.

Table 1. Summary of results for eSTREAM candidates. For each algorithm, the most efficient implementation (high *throughput per area*) has been listed.

Algorithm	A (μm^2)	f (MHz)	radix (bits)	T (Gbit/s)	TpA (Gbit/s. mm^2)	TpA (norm)
Achterbahn	227,763	250	2	0.466	2.044	1.08
Grain	119,821	300	16	4.475	37.346	19.79
MICKEY	82,328	308	1	0.287	3.481	1.85
MOSQUITO	306,907	265	3	0.739	2.408	1.27
SFINKS+	361,643	167	8	1.242	3.434	1.82
Trivium	144,128	312	64	18.568	128.833	68.30
VEST	393,000	286	16	4.257	10.833	5.74
ZK-Crypt	142,007	203	32	6.057	42.656	22.61
AES (OFB)	280.098	182	3.12	0.528	1.886	1.00

Several algorithms (Achterbahn, MOSQUITO, SFINKS, VEST), even in their non-optimized forms, require an *area* comparable to AES. For higher-*radix* implementations, only few (Grain, MICKEY, Trivium, ZK-Crypt) are noticeably smaller than AES. To achieve the stated performance, most algorithms require a *clock frequency* that is above the comfort zone for a standard-cell-based design (roughly 50 FO4 delays, 200 MHz for UMC 0.25 μm technology). Implementations with faster clock rates are possible, but have considerably more overhead during physical design.

Some algorithms are able to achieve significantly higher *throughput* (Grain, Trivium, VEST, ZK-Crypt) than the reference AES implementation. But the real efficiency comparison is the achieved *throughput per area*. Three algorithms (Grain, Trivium and ZK-Crypt) are at least 20 times more efficient than AES. Out of the remaining algorithms, only VEST is able to clearly distance itself from AES, while the others (Achterbahn, MICKEY, MOSQUITO and SFINKS) are only slightly better.

6 Conclusions

The expectations from an efficient cryptographic algorithm will differ depending on the specific application. Sometimes, small area will be of utmost importance, at other times, a certain data throughput will have to be maintained. It is therefore not practical to expect that a single implementation will satisfy all requirements. Our opinion is that the most important aspect for a hardware-efficient cryptographic algorithm is flexibility. It must be possible to trade-off total throughput with area over a wide range.

From the eight implemented eSTREAM candidates, there are several algorithms that can achieve significantly higher throughput per area ratings, and several others which are noticeably smaller in area than the reference AES imple-

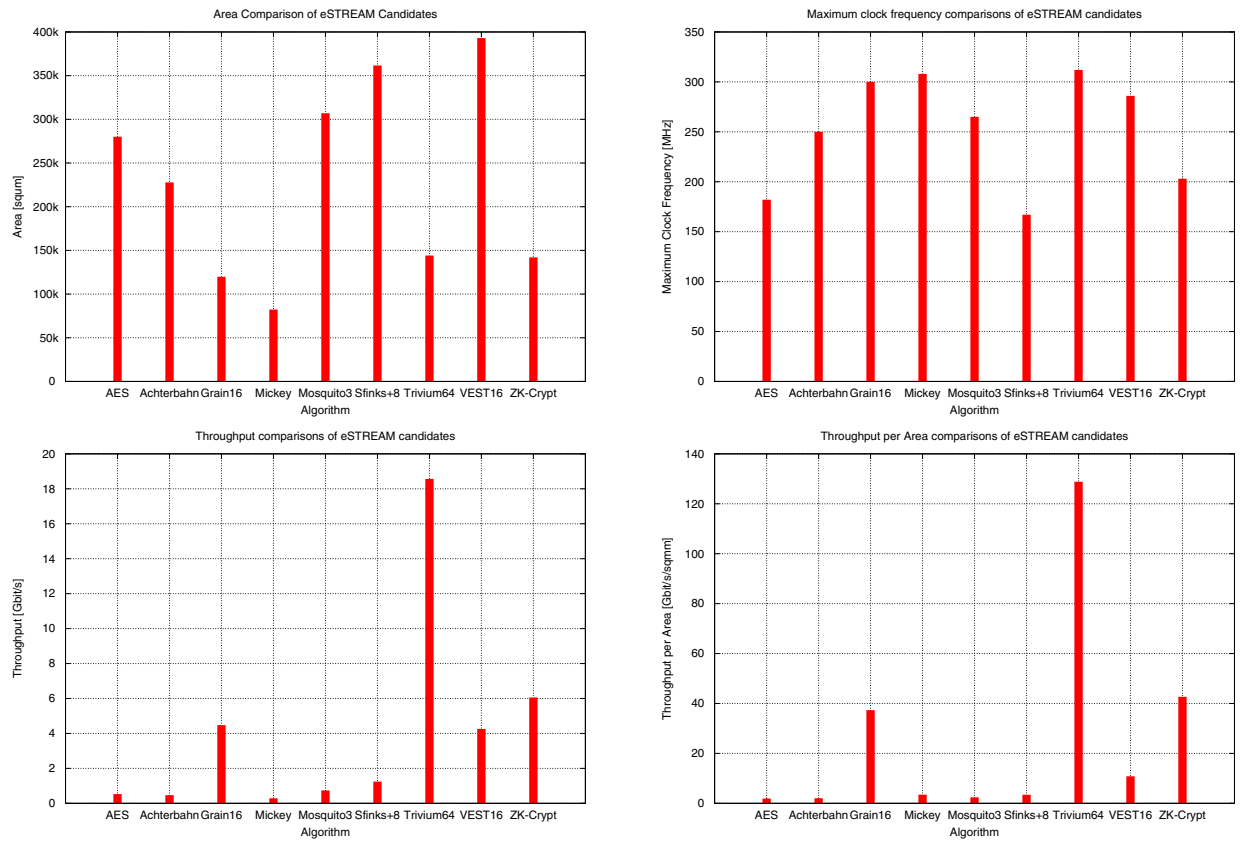


Fig. 4. Synthesis results for eSTREAM candidate algorithms, compared to an efficient AES implementation.

mentation. However, we believe that it is not possible to rate the presented algorithms without knowing their relative cryptographic qualities.

References

1. T. Villiger, J. Muttersbach, H. Kaeslin, N. Felber, and W. Fichtner, "A Globally-Asynchronous Locally-Synchronous VLSI Circuit for the SAFER Cryptoalgorithm," in *Handouts of the First ACiD-WG Workshop of the European Commission's Fifth Framework Programme, Neuchatel, Switzerland*, Feb. 2001, pp. 249–256.
2. A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner, "2 Gb/s Hardware Realizations of RIJNDAEL and SERPENT: A Comparative Analysis," in *Proc. Cryptographic Hardware and Embedded Systems - CHES 2002, LNCS 2523*, Aug. 2002, pp. 144–158, Springer-Verlag.
3. F. K. Gürkaynak, A. Burg, D. Gasser, F. Hug, N. Felber, H. Kaeslin, and W. Fichtner, "A 2Gb/s Balanced AES Crypto-Chip Implementation," in *Proc. of the Great Lakes Symposium on VLSI*, Apr. 2004, pp. 39–44, ACM Press.
4. E. Oswald, T. Johansson, and M. Robshaw, "Criteria to select eSTREAM candidates for implementation as iis student projects." personal communication, 2005.
5. M. Hell, T. Johansson, and W. Meier, "Grain - a stream cipher for constrained environments." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/010, 2005. <http://www.ecrypt.eu.org/stream>.
6. S. Babbage and M. Dodd, "The stream cipher MICKEY." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/015, 2005. <http://www.ecrypt.eu.org/stream>.
7. J. Daemen and P. Kitsos, "The self-synchronizing stream cipher mosquito." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/018, 2005. <http://www.ecrypt.eu.org/stream>.
8. A. Braeken, J. Lano, N. Mentens, B. Preneel, and I. Verbauwhede, "SFINKS : A synchronous stream cipher for restricted hardware environments." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/026, 2005. <http://www.ecrypt.eu.org/stream>.
9. C. D. Cannière and B. Preneel, "Trivium - a stream cipher construction inspired by block cipher design principles." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. <http://www.ecrypt.eu.org/stream>.
10. S. O'Neil, B. Gittins, and H. Landman, "VEST - hardware-dedicated stream ciphers." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/032, 2005. <http://www.ecrypt.eu.org/stream>.
11. C. Gressel, R. Granot, and G. Vago, "ZK-crypt." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/035, 2005. <http://www.ecrypt.eu.org/stream>.
12. B. Gammel, R. Göttfert, and O. Kniffler, "The achterbahn stream cipher." eSTREAM, ECRYPT Stream Cipher Project, Report 2005/002, 2005. <http://www.ecrypt.eu.org/stream>.
13. National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)," *FIPS Publication*, vol. 197, 2001.
14. R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.