RunStream: A High-level Rapid Prototyping Framework for Stream Ciphers

Ayesha Khalid, Queen's University Belfast, UK
Goutam Paul (*Corresponding Author*), Indian Statistical Institute, Kolkata, India
Anupam Chattopadhyay, Nanyang Technological University, Singapore
Faezeh Abediostad and Syed Imad Ud Din, RWTH Aachen University, Germany
Muhammad Hassan, University of Bremen, Germany
Baishik Biswas, Indian Institute of Technology Kharagpur, India
Prasanna Ravi, Center for Development of Telematics, Bangalore, India

We present RunStream, a rapid prototyping framework for realizing stream cipher implementations based on algorithmic specifications and architectural customizations desired by the users. In the dynamic world of cryptography where newer recommendations are frequently proposed, the need of such tools is imperative. It carries out design validation and generates an optimized software implementation and a synthesizable Register Transfer Level Verilog description. Our framework enables speedy benchmarking against critical resources like area, throughput, power, latency and allows exploration of alternatives. Using RunStream, we successfully implemented various stream ciphers and benchmarked the quality of results to be on-par with published hand-optimized implementations.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids, Automatic synthesis

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Hardware generation, high-level synthesis, rapid prototyping, eSTREAM, stream cipher

ACM Reference Format:
Ayesha Khalid, Goutam Paul, Anupam Chattopadhyay, Faezeh Abediostad, Syed Imad Ud Din, Muhammad Hassan, Baishik Biswas and Prasanna Ravi, 2015. RunStream: A High-level Rapid Prototyping Framework for Stream Ciphers. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 00 (0000), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION AND MOTIVATION

The world of cryptography is highly dynamic, the change being constantly fueled by various factors, some of which we highlight in the following. First, *successful cryptanalysis*, not only renders the further use of broken ciphers vulnerable but also open doors for newer/modified subsequent proposals. Second, development of *Custom hardware* aids cryptanalytic attacks by enabling even the brute force attacks for small key sized proposals today. Third, *architectural updates* in GPPs influence cryptographic schemes as the block sizes of software oriented cryptographic proposals are aligned with word sizes of latest computing devices. Also, the imminent ubiquitous computing era has patronized

Author's addresses: A. Khalid, The Institute of Electronics, Communications and Information Technology (ECIT), Queens University Belfast, Belfast, UK; G. Paul, Cryptology and Security Research Unit (CSRU), R. C. Bose Centre for Cryptology and Security, Indian Statistical Institute, Kolkata 700 108, India; A. Chattopadhyay, School of Computer Engineering, Nanyang Technological University (NTU), Singapore; F. Abediostad and S. Imad, RWTH University, Aachen 52074, Germany; M. Hassan, Faculty of Mathematics and Computer Science, University of Bremen, Germany; B. Biswas, Indian Institute of Technology Kharagpur, India; P. Ravi, Center for Development of Telematics, Bangalore, India.

DOI: 0000001.0000001

lightweight cryptography for the Internet-of-Things. Consequently, lightweight cryptographic proposals aiming a thrifty area-power budget with reasonable security are frequently proposed.

An increased interest in subsequent cryptographic competitions, including AES [AES 1997], NESSIE [NESSIE 2000], CRYPTREC [CRYPTREC 2003], eSTREAM [eSTREAM 2008], SHA-3 [SHA-3 2012], CAESER [CAESAR 2012] is evident by their growing number of candidate proposals submitted compared to their successor. The initial phase of eSTREAM competition for stream ciphers attracted 34 proposals [eSTREAM 2008]. Most of these proposals support multiple modes and versions for variable key, IV and block sizes. Quantifying their performance as custom VLSI implementations requires benchmarking against diverse parameters like area, power, throughput, latency etc. The human-driven process of writing and validating HDL for stream ciphers is slow, error prone and requires expertise both in algorithm and hardware design domains to reach the options best suited for an application requirement. Moreover, due to the emergence of lightweight block ciphers in recent times, the traditional generalization that stream ciphers always outdo block ciphers in resource economization no longer holds true. Hence in principle, stream ciphers' resource utilization should also be benchmarked against comparable block ciphers and vice versa. This further compounds the workload for equitable benchmarking of a new block/stream cipher design.

We aim to solve these problems through *automation*. RunStream is a rapid prototyping framework to quickly and efficiently realize stream cipher implementations from user specified configurations. This approach significantly shortens the VLSI design cycle, boosting productivity without tiring the designer into the low level implementation trivialities. It allows a quick hardware resource estimation, early functional validation of desirable cipher properties and speedy exploration/selection among a wide space of high quality cipher proposals. With a similar motivation, we earlier presented RunFein (RAPID-FeinSPN) [Khalid et al. 2013], that caters to the rapid prototyping of block ciphers. RunStream completes the picture for symmetric key cryptography by catering to stream ciphers. The quality-of-results for area/timing rival the hand-crafted cipher implementations.

1.1. Previous Work

Various high-level synthesis (HLS) tools to quicken the VLSI design cycle have been proposed both academically and commercially. Noticeable examples include Vivado HLS by Xilinx [Vivado 2012], GAUT [GAUT 2007], Synphony C by Synopsys [Synphony 2009], C-to-Silicon compiler by Cadence [Cadence 2009], Hercules [Ajax 2009], Handel C by Mentor Graphics [Mentor Graphics 1996]. These modeling environment tools accept concise representations of design specifications to generate an HDL implementation. The design specifications are in a higher abstraction level and often require learning a new language. Additionally, as none of these tools focus on any specific application class, the optimizations undertaken remain *generic* and often suboptimal compared to the hand-optimized implementations. Two case studies taking up HDL code for modern cryptographic algorithms and generating HDL descriptions by a new generation HLS tool (Vivado HLS by Xilinx [Vivado 2012]) are worth mentioning. In [Gaj et al. 2010], all the five round-3, SHA-3 candidates were undertaken by the Vivado HLS tool and their performance was benchmarked against manual RTL. In-spite of various iterations of the source code modifications by pragmas (constraints) to economize hardware resources, the throughput efficiency for HLS remains between 62% - 85% lower, compared to manual RTL for various Altera devices. Similarly, noticeable performance penalty is caused by the HLS tool when various configurations of AES are generated and performance profiled on different families of FPGAs [Homsirikamol and Gaj 2014]. On a Virtex-7 FPGA, the degradation of HLS generated AES in terms of throughput efficiency lags behind 28% to 42%, compared to manual RTL.

Taking up an orthogonal approach to high-level synthesis, we present a *language independent* configurable design space catering to cryptography. This also eliminates the dependence of design quality on the coding style of the programmer which is the case with conventional HLS tools. After analyzing cryptographic workloads we identified the set of constructive components and structural customizations that are generic enough to define any stream cipher. The user specified design configuration comprises of a set of these *functionally complete* constructive elements. RunStream accepts a sophisticated design capture of high-level design configuration through a GUI. The design is then validated for completeness and correctness. These rule checks detect functional and system-level problems much earlier in the design cycle improving design reliability and shortening time to market. The tool infers the necessary interfaces and structures to implement optimized HDL along with verification environments and necessary scripts. It provides a seamless end to end verification from the configuration to RTL validation/verification environments.

1.2. Original Contribution

The noteworthy contributions of this work are listed.

— We surveyed a diverse and wide range of stream ciphers to systematically build up a *functionally complete* set of constructive elements/structures to define the configuration space of any stream cipher.
— The configuration model completeness and RunStream tool effectiveness is validated by implementing some bit/byte/word oriented prominent stream ciphers and benchmarking their performance to match their manual implementations.
— We integrated NIST test suite with RunStream for evaluation of statistical randomness of the pseudorandom bitstream.

Rest of the paper is organized as follows. Section 2 classifies stream ciphers and elaborates their constructive design elements. The overall RunStream toolflow is presented in Section 3, while the details of software and hardware generation engines is discussed in Section 4. Experimental evaluation of RunStream is discussed in Section 5. Section 6 concludes the paper and presents future outlook.

2. INGREDIENTS OF A STREAM CIPHER

In this section, we present background material on stream ciphers, including their classification, typical elements of construction along with their modes of operation. Since our goal is to *define* configuration space of stream ciphers for high level synthesis, we strictly focus on their architectural/operational constructs. Their complexity, statistical and cryptanalytic properties are therefore skipped but could be referred from [Menezes et al. 1996, Chapter 6].

2.1. Stream Ciphers: Definition and Classification

Stream ciphers encrypt a *stream* of individual characters (bits or words), while block ciphers operate on chunks of *blocks*. Unlike the *memoryless* nature of block ciphers, the encryption/decryption transformation of stream ciphers is dependent on their current state, consequently, they are also termed as *state ciphers*. Their judiciously-chosen lightweight Boolean operations make them suitable for environments where resources are restricted and sustaining a high throughput is critical.

Two classes of stream ciphers are Synchronous or Self-synchronizing stream ciphers. A formal definition from [Menezes et al. 1996, Chapter 6] and a block diagram of the two types is depicted in Fig. 1.

The initial state (σ_0) of stream ciphers is generated by *Initialization function (init)* based on key (k) and Initialization Vector (IV). For a synchronous stream cipher, the *State update function (f)* calculates next state (σ_{t+1}) depending upon the current state

only, while for self-synchronizing stream ciphers, the internal state comprises of a fixed number (l) of previous ciphertexts generated. Due to this dependence on ciphertext, the keystream for self-synchronizing stream ciphers cannot be precomputed. The *keystream function (g)* transforms the current state (and the key in case of self-synchronizing stream ciphers) to generate keystream. *Additive stream ciphers* have *Output filter function (h)* as an XOR function, the inverse function (h^{-1}) at receiver is also a XOR function. Additive stream ciphers generating bits of keystream are termed as *Binary additive stream ciphers*.

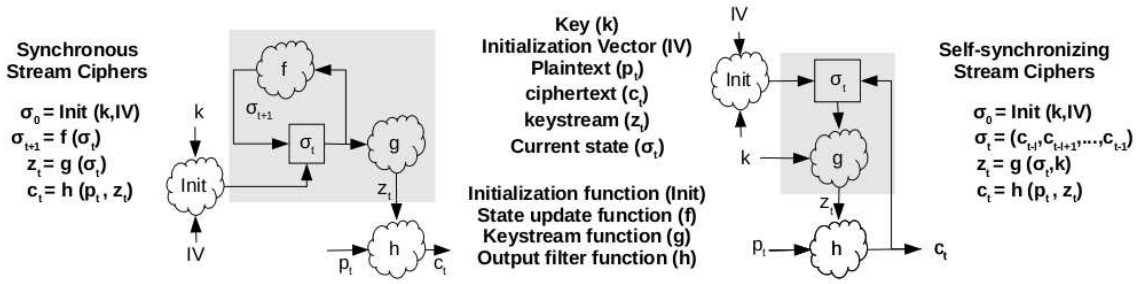

Fig. 1. Synchronous stream cipher (left) and self-synchronizing stream cipher (right)

2.2. Sequential Prototypes

A Stream cipher can be modeled as a regular clocked finite state machine (FSM). Like any FSM, it has sequential and combinational parts. The sequential components/ registers hold the internal state of stream ciphers. Typical structures used for their construction are given in the following.

2.2.1. FSR:. A Finite Shift Register (FSR) comprises of L delay elements, each of which is shifted to the next at each clock cycle, as shown in Fig. 2 (left). The content of stage 0 forms the output of the FSR, while the new content of stage $L-1$ is calculated based on a *feedback function*. Each c_i is a single bit number controlling the inclusion of i^{th} stage value in the calculation of the feedback function. The categorization of types of FSR is done based on the nature of the feedback function. We enlist here the prominent ones.

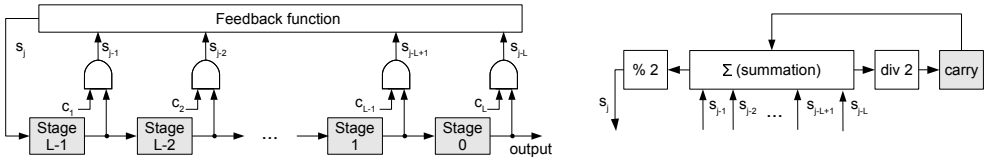

Fig. 2. A Feedback Shift Register, FSR (left) a carry Linear feedback shift register FCSR (right)

— *LFSR/NFSR:* Depending on the feedback function being linear or non-linear, an FSR is categorized as a linear FSR (LFSR) or nonlinear FSR (NFSR). Fig. 2 (left) shows an LFSR with feedback function comprising of bit-wise XOR of the contents of all stages for which $c_{L-i} = 1$.

$$s_j = (c_1.s_{j-1} \oplus c_2.s_{j-2} \oplus \ldots \oplus c_L.s_{j-L}) \; for \; j \geq L$$

— *FCSR:* Feedback with Carry Shift Register (FCSR) is an NFSR that has summation as feedback function, as shown in Fig. 2 (right). It keeps an extra memory bit $carry$ to retain the carry from one addition to be added up in the next cycle's addition.

$$sum \; = \; (c_1.s_{j-1} + c_2.s_{j-2} + \ldots + c_L.s_{j-L} + carry) \; for \; j \geq L$$

where s_j and $carry$ are the remainder and dividend of 2, respectively.

2.2.2. Jump Registers. Jump registers form a cascade of multiple delay elements, each of which implements an autonomous Linear FSM. Each of these delay elements have a corresponding controlling jump bit to make them $jump$ to a new value [Jansen 2004]. Two noticeable eSTREAM candidate ciphers that use jump registers are MICKEY [Babbage and Dodd 2006; 2008] and POMARANCH [Helleseth et al. 2006].

2.2.3. FSM Registers. Other than the cascade shift register constructions, sequential storages may also be used to hold the ciphers state. Stream ciphers using these FSM registers (other than FSRs) include SNOW 3G [3G 2006] and ZUC [ZUC 2011]. RC4 [Schneier 1996] and HC-128 [Wu 2008] require 2K and 32 Kbits of state information, respectively.

2.3. Nonlinear Combination Generators

2.3.1. For LFSR based Stream Ciphers. LFSRs are favorite primitives for stream cipher design due to their desirable statistical properties and hardware friendly nature. Their susceptibility to chosen plaintext attack must be overcome by breaking their linearity according to these recommended and analyzed constructions [Menezes et al. 1996, Chapter 6].

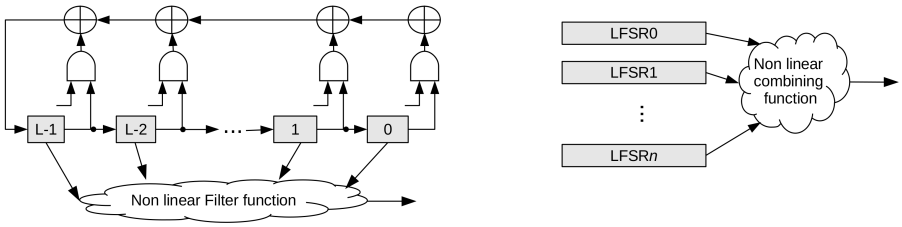

Fig. 3. Nonlinear filter generator (left) Nonlinear output from multiple LFSRs (right)

— *Nonlinear Filter Generator:* The output of an LFSR is generated through a nonlinear filter generator. At each clock it takes bits/words from fixed locations of LFSR to generate output as shown in Fig. 3(left).
— *Nonlinear Output from Multiple LFSRs:* Multiple LFSRs are used in parallel and the keystream is generated as a nonlinear transformation of the outputs of LFSRs as shown in Fig. 3(right). Stream ciphers using this construction are called nonlinear combination generators and the nonlinear function is called the *combining function*. *Geffe generator* and *summation generator* are examples of such combining functions.
— *Clock Controlled Generators:* Clock controlled generators introduce the nonlinearity in LFSRs by making the clocking irregular. The LFSRs are not clocked at every cycle and clocking of one LFSR is controlled by the Boolean transformation of another LFSR taps and vice versa. Some known techniques of clock controlled generators are *alternating step generator*, *shrinking generator* etc.

2.3.2. Other Designs. A nonlinear update function for a popular stream cipher RC4 [Schneier 1996] is the *Triangular function* (T-Function). RC4 has an internal state of 256 Byte array, denoted by $S[0 \ldots N-1]$ and accessed by indices i and j. In every iteration of Keystream generation phase, the T-function increments i, updates j by $j = j + S[i]$, swaps values of $S[i]$ and $S[j]$ and produces one byte of output as $S[S[i] + S[j]]$. Many RC4 variants, including RC4$^+$ [Maitra and Paul 2008], VMPC [Zoltak 2004] and the recently proposed Spritz [Rivest and Schuldt 2014] use RC4-like general design principle of T-function.

2.4. Combinational Primitives

The combinational primitives are required to configure and specify the *initialization (init)*, *state update function (f)*, *keystream generation function (g)* and *output filter function (h)* of a stream cipher as given in Fig. 1. The feedback function of FSRs, jump registers and FSM registers as well as the functionality of nonlinear combination generators (Section 2.3) is described and configured using these combinational primitives.

These primitives include Boolean, arithmetic and bitwise operations (including arithmetic/logical shifts, rotations). For non-binary stream ciphers, field operations, S-Box and P-Box are employed. Other glue logic elements like multiplexers are required to change the behavior of a cipher as its phase changes.

2.5. Operational Phases

Stream ciphers have the following three phases of operation.

— *Key/IV Setup:* This phase initializes the state of stream ciphers, based on the secret key and IV (or nonce, i.e., number used once) provided by the user. The key and IV values may require some preprocessing ($init$ in Fig. 1) before being fed into the stream cipher structure. For FSR based stream ciphers, the Key/IV Setup requires at least as many cycles as the length of the FSR to alter its initial value.
— *Randomization/Runup/Warmup:* The randomization phase sufficiently scrambles the state of the cipher based on Key and the IV, before the keystream is generated. Since this phase does not generate valid keystream, its duration is a trade-off between security and speed.
— *Keystream Generation Phase:* The keystream bits/words are produced as the randomization phase is over. A continuous keystream is generated that determines the throughput of the system.

Since no stream cipher output is generated during the first two phases, they are together named as keystream *Initialization*. For every new key, IV combination, Initialization must be carried out before valid keystream generation. Consequently, ciphers requiring frequent re-initialization have this initialization latency as a possible bottleneck.

2.6. Stream Cipher Sample Set

We pick up a diverse set of stream ciphers for results, discussion and implementation by RunStream. Table I gives their classification on the basis of their composing constructions and their salient features, while Table II summarizes the best known attacks against them till date. A/5-1 was part of the original encryption algorithm for GSM, but since it was export restricted, A/5-2 was developed. They were reverse engineered and verified and the design was revealed [Briceno et al. 1999]. Both of these are multiple LFSR based stream ciphers, using irregular clocking for nonlinearity. E0 [Bluetooth 2001] is also an LFSR based stream cipher with a summation combiner used for output generation. It was used in Bluetooth communication. Although the security of these proposals have been compromised [Barkan et al. 2003; Goldberg et al. 1999; Lu et al. 2005], we included them nevertheless for comparison against their known implementations.

eSTREAM was launched to restore the confidence of cryptographic community on stream ciphers as questions were raised on their usability [Shamir 2004]. Some popular stream ciphers, when found flawed, were replaced by block ciphers, e.g., KASUMI (or A/5-3) block cipher [3GPP 1999] replaced A/5-1 and A/5-2 stream ciphers after weaknesses were pointed out [Barkan et al. 2003; Goldberg et al. 1999], WPA/WPA2 use AES based security, unlike its predecessor WEP, that used RC4 stream cipher. Moreover, when all the 6 proposals of an earlier competition NESSIE succumbed to cryptanalysis, eSTREAM was launched. It was initiated by EU ECRYPT network and took four years of effort to finalize new stream cipher proposals [eSTREAM 2008].

Table I. Classification, construction and Salient Features of stream ciphers

	Classification/Construction											Salient Features		
	Synchronous	Additive	Binary	LFSR	FCSR	NFSR	Jump Registers	FSM Registers	Irregular Clocking	Multiple Generators	Nonlinear Filter	Granularity (bits)	Key (bits)	IV (bits)
A-5/1	✔	✔	✔	✔	✗	✗	✗	✗	✔	✔	✗	1	64	22
A-5/2	✔	✔	✔	✔	✗	✗	✗	✗	✔	✔	✔	1	64	22
E0	✔	✔	✔	✔	✗	✗	✗	✔	✗	✔	✔	1	8-128	-
Grain-v1	✔	✔	✔	✔	✗	✔	✗	✗	✗	✔	✔	1	80	64
Grain-128	✔	✔	✔	✔	✗	✔	✗	✗	✗	✔	✔	1	128	96
Grain-128a	✔	✔	✔	✔	✗	✔	✗	✗	✗	✔	✔	1	128	96
Trivium	✔	✔	✔	✔	✗	✗	✗	✗	✗	✔	✗	1	80	80
MICKEY 2.0	✔	✔	✔	✗	✗	✔	✔	✗	✗	✗	✗	1	80	80
MICKEY-128	✔	✔	✔	✗	✗	✔	✔	✗	✗	✗	✗	1	128	128
RC4	✔	✔	✗	✗	✗	✗	✗	✔	✗	✗	✔	8	40-2K	-
ZUC	✔	✔	✗	✗	✗	✔	✗	✔	✗	✗	✔	32	128	128
SNOW 3G	✔	✔	✗	✗	✗	✔	✗	✔	✗	✗	✔	32	128	128

Table II. Best known state recovery and key recovery attacks against stream ciphers

	State Recovery Attack	Comp. Complexity	Key Recovery Attack	Comp. Complexity
A-5/1	[Barkan et al. 2003]	< 1 sec on a PC	-	-
A-5/2	[Goldberg et al. 1999]	2^{16} dot products	-	-
E0	-	–	[Lu et al. 2005]	2^{40}
Grain-v1	[rstad 2008]	2^{71}	[Berbain et al. 2006]	2^{43}
Grain-128	[Mihaljevic et al. 2012]	2^{98}	[Dinur et al. 2011]	2^{90}
Grain-128a	-	-	-	-
Trivium	[Maximov and Biryukov 2007]	$192 \times 2^{83.5}$	[Dinur and Shamir 2009]	2^{45}
MICKEY 2.0	-	-	-	-
MICKEY-128	-	-	-	-
RC4	[Maximov and Khovratovich 2008]	2^{241}	[Basu et al. 2009]	2^{53}
ZUC	-	-	-	-
SNOW 3G	-	-	-	-

The initial phase of eSTREAM attracted 34 proposals. Out of these only two proposals (SSS and Moustique) were self-synchronizing stream ciphers while the rest were all synchronous. After various phases of thorough scrutiny, judging the security, performance, simplicity and flexibility of these proposals, 7 were included in the portfolio, 4 in software profile and 3 in hardware profile. We take up these three stream ciphers (along with their modified versions) in the hardware profile of eSTREAM portfolio including Grain-v1 [Hell et al. 2007], Grain-128 [Hell et al. 2006], Grain-128a [Ägren et al. 2011], MICKEY 2.0 [Babbage and Dodd 2006], MICKEY-128 [Babbage and Dodd 2008] and Trivium [De Canniere and Preneel 2005]. Their design diversity makes them good candidates for HLS, i.e., MICKEY is based on jump registers, Trivium employs 3 LFSRs, Grain ciphers have one LFSR and one NFSR. No successful cryptanalytic effort against them has yet been reported.

For non-binary stream ciphers, we take up RC4 [Schneier 1996], most commonly used to protect Internet traffic using the SSL (Secure Sockets Layer) protocol, Transport Layer Security (TLS), WEP (Wired Equivalent Privacy), Wi-Fi Protected Access (WPA), along with several application layer softwares. We also take up ZUC [ZUC 2011] and SNOW 3G [3G 2006], both of whom have been included in the security portfolio of 3GPP LTE-Advanced, the potential candidate for 4G mobile broadband communication standard. They generate 32-bit words of keystream and have internally 16 tap NFSRs.

The RunStream methodology provides a rich configuration design space, comprehensive enough to model a diverse range of stream cipher classifications. It covers the L/N FSR based structures that are favorite primitives for many hardware-oriented

stream ciphers. Similarly, the software based designs, commonly employing S/P-Boxes, T-functions, large states and other arithmetic/logical operations can be prototyped. Hybrid designs, employing a combination of these primitives are also realizable using RunStream. Additionally, block ciphers based on stream ciphers (e.g., KATAN, KATANTAN [De Canniere et al. 2009]) can also be realized. RunStream supports modular composition of cryptographic building blocks, which can be conveniently extended to support newer proposals if/when the need arises.

3. RUNSTREAM TOOLFLOW

RunStream is developed around the concepts of *modularity* and *extensibility*. For stream ciphers, cryptographers are free to choose from a large set of cryptographic primitives/structures since unlike block ciphers, stream ciphers do not have a standard model/structure for their construction. A key challenge addressed in this work is to identify a *complete* set of sub-structures that fulfill this diverse range of ciphers. The constructive composition of the design, consequently translates to rich primitive libraries for software and hardware realizations. Performance benchmarking/design validation and support for statistical analysis require additional checks and function definitions. The biggest technical challenge is to develop a tool capable of seamlessly integrating these sub-structures and functions into a working model, without sacrificing the performance of the implementation, both of software and hardware platforms. RunStream validates the design capture and successfully abstracts away the diversity of the design space by translating the configuration into a generic stream cipher template. The design optimality is carefully preserved in HDL realization and is experimentally benchmarked to be in par with hand crafted realizations.

The toolflow of RunStream is shown in Fig. 4. The user populates the configuration space of a stream cipher, along with test vectors and gets customized implementations. The configuration for a cipher could be added, parameter by parameter, or could be saved and loaded later for easier manipulation. A list of known cipher configurations (given in Table I) is provided along. The configurations undergo a set of design rule checks before generating the software and hardware implementations.

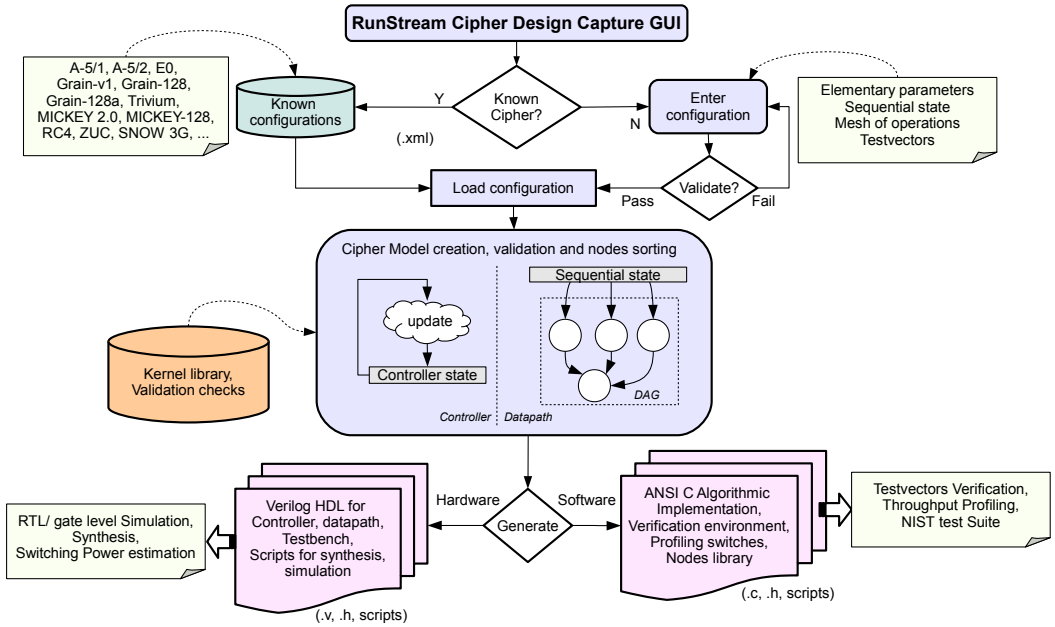

Fig. 4. RunStream Toolflow

3.1. Cipher Configuration Space

This section discusses the *configuration space* parameters, generic enough for any stream cipher. The configuration parameter set is categorized into *elementary parameters*, *sequential state* information and a *mesh of operations*. (All parameterizable attributes that a user must populate are highlighted in the proceeding discussion).

— *Elementary parameters:* The granularity/wordsize of a stream cipher is represented as $\mathbf{S_w}$. Elementary parameters including the sizes of key and IV, represented as $\mathbf{S_K}$ and $\mathbf{S_{IV}}$, respectively, are described in terms of number of words. The set of test vectors includes **Key** and **IV** values along with the expected **Keystream** for verification, specified in terms of a known **endian-ness**. For simplicity, the three phases of a stream cipher (as given in Section 2.5) are referred as *phase0*, *phase1* and *phase2*. The number of operational cycles for the three phases of stream ciphers is defined as $\mathbf{Pi\_cnt}$ by the user, where i is the phase number.
— *Sequential state information:* Each of the various types of sequential elements (as discussed in Section 2.2) is defined as a *register* array. The user specifies the total **number of register arrays**, along with the **type** of each (FSR, jump register, FSM register). The **granularity** of each element of these arrays is also specified, since it may or may not match the wordsize of the cipher. The **depth** of the register array is the number of delay elements in the FSR/jump register array; for individual FSM registers, the array depth is configured as 1. The **clocking** of these register arrays can be specified to be regular or conditional.
— *Mesh of operations:* The combinational primitives of the stream cipher (discussed in Section 2.4) are specified as a *mesh* of operations. This mesh comprises of *nodes* of operations that interact with each other through *interconnects*. This mesh is a Graph (G), defined by vertices (nodes) and edges (interconnects). Each interconnect representing the directed edge between nodes is specified by an *ordered pair* of *initial node* and *terminal node*. The width of interconnects entering and leaving a node may not always be equal (for nodes with ciphers operations like concatenation or bit splitting), requiring explicit specification. Hence the user specifies the total **number of nodes** along with the **width** of each and an **ordered pair** specifying initial and terminal node numbers. For each node, the user must also specify an atomic **operation**, comprising of either of the following operational classes.
  — **Basic Boolean operations:** AND, OR, NOT, XOR
  — **Arithmetic operations:** Add, subtract, multiply, divide (by power of two)
  — **Bit-wise operations:** Arithmetic shifts, logical shifts, rotation, rotation with carry
  — **Cryptographic primitives:** S-Box, P-Box
  — **Galois Field Operations:** GF-multiplication, GF-division
  — **Bit Manipulation:** bit masking, concatenation / bit-reorganization
  — **Glue logic:** Multiplexers, constants, no-operation

This parameterization covers the configuration space of today's stream ciphers (including the ones in Table I) and can be enhanced to accommodate any newer, non-typical future proposals. The framework is aided by a user-friendly GUI for a sophisticated configuration capture, convenient default values are provided by the tool in the GUI wherever necessary. The S-Box/P-Box values can be efficiently added using text files. It is worth highlighting that a cipher may have multiple configurations that are *algorithmically equivalent.* For example, a lookup table node (S-Box) could be replaced by a series of several combinational logic nodes to trade for better area efficiency (but higher critical time), without changing the cipher. Noteworthy is the fact that RunStream can be configured to implement an S-Box as a LUT or any other combinational logic with its nodes. Hence various design options could be quickly explored for reaching an optimal solution.
Trivium cipher configuration - A walk-through: We take up a typical synchronous stream cipher, Trivium [De Canniere and Preneel 2005] from eSTREAM and try to work-

out its configuration according to the discussed configuration space. Trivium's 288-bit internal state consists of 3 regularly clocked LFSRs of different lengths. It has an 80 bit Key and an 80 bit IV, with *phase0* and *phase1* defined to be **288** and $(\mathbf{288} \times \mathbf{4})$ cycles, respectively. The architectural details of Trivium are depicted in Fig. 5, the 3-to-1 MUXes control the input to the LFSRs under three different phases of operation. The user must first identify all atomic operations (AND, XOR, MUX) as a separate operational *node* and then enumerate them (not in any specific order). A total of 13 nodes describe the complete functionality of the cipher (The node number, register number is highlighted and placed at the bottom right of each node). Fig. 12 shows a GUI snapshot of RunStream with Trivium configuration.

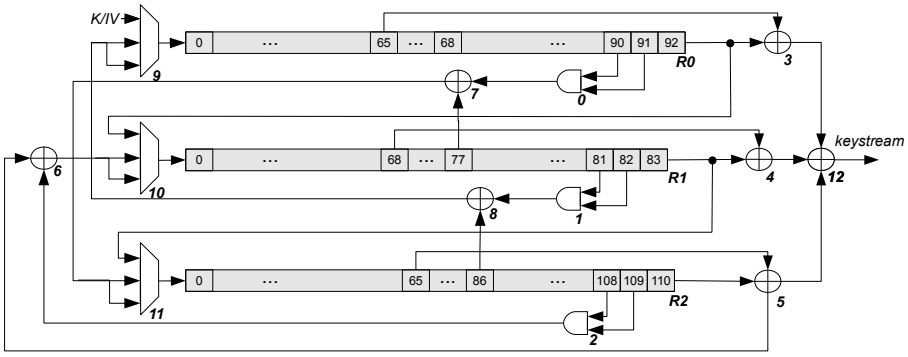

Fig. 5. Architectural structure of Trivium implementation

The three LFSRs are right shifted once per clock cycle and named *R0*, *R1* and *R2*; *granularity* being 1 bit, *unconditionally clocked* and the depth is specified to be 93, 84 and 111, respectively. All configuration parameters for Trivium are fed to the tool's GUI and are stored as an *xml* configuration file, a snapshot of which is shown in Fig. 6. The information for each of the FSR is kept as a separate token in the file, for *R0* the size, type, granularity and clocking can be seen. For each of the 13 nodes, the user specifies an operation (*R0*), output width (*wordsizeout*) and the node input sources (discussed in detail in the next section). The node generating the final keystream is specified as separate token (*keystream*). The feedback for each of the three registers is also specified by a node number. Other elementary parameters like the number of phases, cycles for each phase and the test vectors (not shown in Fig. 6) are also specified and saved as part of the cipher configuration.

3.2. Cipher Model Creation and Validation

The configuration file is parsed by RunStream and stream cipher model is created comprising of a *controller* and *datapath*. The controller keeps track of the phase changes by aid of a *counter*, according to the cycle count of phases specified (further discussion in following section). The datapath of the cipher is constructed by a *Directed Acyclic Graph* (DAG) comprising of nodes and interconnects.

Before that, each node specification by the user is subjected to some elementary rule checks, such as the following.

— The number of inputs match the number of inputs for the operation specified, hence two for binary operation etc.
— The S-Box values are $\in [0..2^{S_W}]$.
— P-Box, rotation/shifting, XOR operations have arguments $\in [0..S_B]$.
— The polynomial coefficients for GF-mul are not $\emptyset$.

```
<Defines>
 <No_Of_Registers Size="3"/>
 <Registers>
  <R_0 Size="93"/>
  <R_0 type="LFSR"/>
  <R_0 granularity="1"/>
  <R_0 clocking="unconditional"/>
  <R_0 feedback="node9_out"/>
 ...
 </Registers>

</Defines>

<Nodes>
 <No_of_Nodes Size="13"/>

 <Node_0>
  <Node_0/>
  <Node_0_Op value="AND"/>
  <Node_0_wordsizeout value="1"/>
  <Node_0_In_1 value="LFSR_0[90]"/>
  <Node_0_In_2 value="LFSR_0[91]"/>
 </Node_0>

 <Node_1>
  <Node_1/>
  <Node_1_Op value="AND"/>
  <Node_1_wordsizeout value="1"/>
  <Node_1_In_1 value="LFSR_1[81]"/>
  <Node_1_In_2 value="LFSR_1[82]"/>
 </Node_1>
   ...
<Node_3>
  <Node_3_Operation value="XOR"/>
  <Node_3_wordsizeout value="1"/>
  <Node_3_Input_1 value="LFSR_0[65]"/>
  <Node_3_Input_2 value="LFSR_0[92]"/>
 </Node_3>
   ...
```

```
   ...
 <Node_11>
  <Node_11/>
  <Node_11_Op value="MUX"/>
  <Node_11_wordsizeout value="1"/>
  <Node_11_In_1 value="PHASE"/>
  <Node_11_In_2 value="LFSR_1[83]"/>
  <Node_11_In_3 value="node7_out"/>
  <Node_11_In_4 value="node7_out"/>
 </Node_11>

 <Node_12>
  <Node_12/>
  <Node_12_Op value="XOR"/>
  <Node_12_wordsizeout value="1"/>
  <Node_12_In_1 value="node3_out"/>
  <Node_12_In_2 value="node4_out"/>
  <Node_12_In_3 value="node5_out"/>
 </Node_12>

</Nodes>

<keystream>
  <keystream value="node12_out"/>
<keystream>

<Phases>
 <No_of_Phases Size="3"/>
 <Phase_0_no_of_cycles size="288"/>
 <Phase_1_no_of_cycles size="1152"/>
 <Phase_2_no_of_cycles size="100"/>
</Phases>

</StreamCipher>
```

Fig. 6. The configuration file snapshot input to RunStream for Trivium

As these checks pass, a DAG is created, Fig. 7 shows the DAG to generate the feedback bits for the LFSRs and keystream output As a convention, the count for all nodes, phases, interconnects and registers starts from 0. A separation of sequential and combinational elements for Trivium can be seen.

The first layer of nodes is the *source nodes* as these nodes do not take inputs from any other nodes. Since they require no operation and are taps from the three FSRs, the node numbers are not assigned to them. The rest of the nodes are *enumerated*. *Node0* is an AND operation node with two incoming interconnects *(R0[90],0)* and *(R0[91],0)* along with one outgoing interconnect, i.e., *(0,7)*. *Node7* takes input from the *source nodes* as well as the enumerated nodes, the three inputs being *Node0*, *Node3* and *R1[77]*. *Sink nodes* are the ones whose output is not forwarded to any other node. These include *Node9*, *Node10* and *Node11* which are MUX by operation and control the input to the LFSRs during the 3 phases of operation.

The interconnects between the user defined nodes are stored as a finite graph G on n vertices where n is the sum of both the enumerated operational nodes and the source nodes from register taps. Referring back to Trivium (Fig. 7), there are 13 enumerated nodes and 18 source nodes. An $n \times n$ entries graph adjacency matrix holds the interconnects where each entry a_{ij} represents if an edge exists from i^{th} node to j^{th} node or not.

Next the graph(G) undergoes a second phase of model validation by a list of defined rule checks. The user is prompted in case of a violation and the cipher implementation does not proceed unless a valid configuration is specified.

— G should not be an empty graph, should have no unreachable dangling nodes. It should be a simple graph (a strict graph).
— G should have no duplicate edges (interconnects with same initial and head node), hence the corresponding graph adjacency matrix holds binary elements.

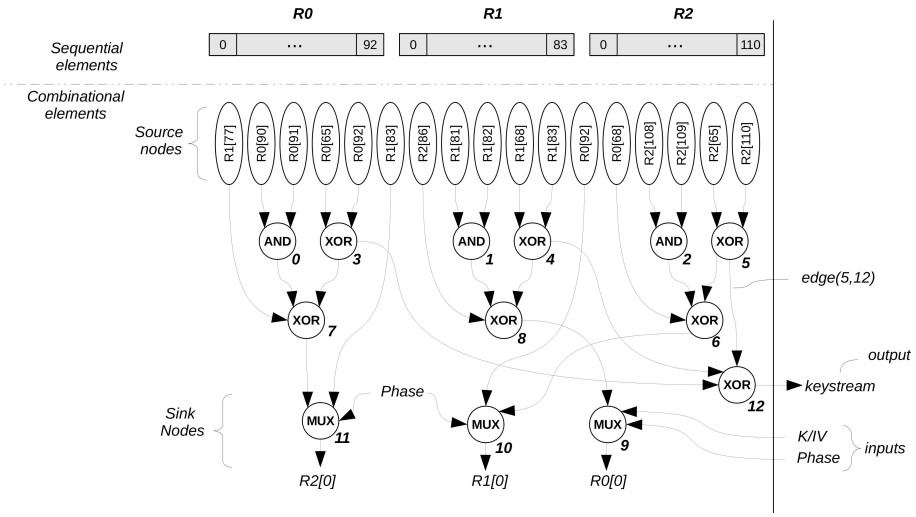

Fig. 7. The Directed Acyclic Graph for Trivium nodes implementation (datapath of cipher model)

— There should be no circular dependencies (interconnects with same node as initial and terminal nodes). Hence all diagonal entries of the adjacency matrix should be zeros.

For simulation, the graph(G) must have an *execution order* for a valid sequential evaluation of cipher. Although HDLs mimic a parallel execution model, for software implementations that undertake a sequential execution of code chunks. Hence an execution order for evaluation of nodes is required for a valid simulation of the cipher model. A directed graph with no loops makes the configuration a directed acyclic graph (DAG), whose execution order may be found by *topological sorting*. The node numbers are updated after sorting. Fig. 5 and Fig. 9 shows DAGs with topologically sorted nodes.

4. GENERATION ENGINES AND ALGORITHM MAPPING

This section discusses the mapping of cipher model into software and hardware implementations. Modeling of stream ciphers given in Table I using RunStream is discussed next.

4.1. Software Generation Engine

The software generation engine compiles the cipher model to generate a high performance, fixed-point ANSI-C description. The code is enhanced by a simulation environment with user controllable switches for verification, throughput profiling, data dumping etc. The generated code is not specifically optimized for a particular General Purpose Processor (GPP), however, it has a regular structure and good code readability. All the configuration parameters of the cipher (as specified in xml file listing in Fig. 6) are *#defined* in a *header* file. This includes all elementary cipher parameters along with the information about sequential resources and the sorted nodes. Data types of registers, nodes interfaces and other controller related variables are *typedef*-ed in accordance with the *granularity* specified.

Supplementary *functions* are kept in a separate file, that is *included* in the main file during simulation. These functions include datatype conversion functions (e.g., conversion of hexadecimal to binary arrays), data dumping and verbose simulations. For each operational node, a separate function is defined with interface and functionality, as per the user specified.

The main body of code, having the *controller* and the *datapath* of the cipher model, is a separate file that *includes* all supplementary and header files. For elaboration of code

simulation environment, we refer to Trivium. Algorithm 1 gives a code chunk for $phase0$ of the cipher. The controller part of the cipher comprises of two local variables $phase$ and loop variable i, keeping count of the phase and cycle under execution, respectively. The loop in line 2 iterates for the number of cycles defined for $phase0$ ($P0\_cnt$).

In every cycle iteration, first the combinational mesh of operational nodes are executed and then the sequential resources are updated.

— For updating the combinational nodes, the function calls to all 13 nodes are executed (line 3-7). The inputs to the nodes are taken from the defines file while the outputs are local variables (for *node0*, $Node0\_in0$ and $Node0\_in1$ are defined to be R0[90] and R0[91]). The output of one node could also be input to another, as per the user specification.
— Next the sequential elements are updated (line 8-16). *R0*, *R1* and *R2* are defined as local arrays with user specified size and type. *R0* being an LFSR, the left shifting of its contents is carried out by a loop over variable j (line 8). Similarly, *R1* and *R2* are shifted by loops in line 10 and 12, respectively. The input tap of these LFSRs are updated by their corresponding nodes as shown in line 14, 15 and 16, respectively. The code for value update of FSM registers follows the LFSRs update, however Trivium has no FSM registers.

A valid $keystream$ bit is generated from a node ($node12$ for Trivium) no earlier than the last phase of cipher implementation. For the other two phases, the code listing is similar to the one given in Algorithm 1, except the line 1 that holds different phase number.

ALGORITHM 1: RunStream generated pseudo code chunk for phase0 of Trivium stream cipher

```
1  phase = 2;
2  for i=0 till ≤ P0_cnt step 1 do
3      node0_out = node0(Node0_in0, Node0_in1);
4      node1_out = node1(Node1_in0, Node1_in1);
5      ...
6      node11_out = node11(Node11_in0, Node11_in1, Node11_in2, Node11_in3);
7      node12_out = node12(Node12_in0, Node12_in1, Node12_in2);
8      for j=(R0_size − 1) till ≥ (0) step 1 do
9          R0[j] = R0[j − 1];
       end
10     for j=(R1_size − 1) till ≥ (0) step 1 do
11         R1[j] = R1[j − 1];
       end
12     for j=(R2_size − 1) till ≥ (0) step 1 do
13         R2[j] = R2[j − 1];
       end
14     R0[0] = node9_out;
15     R1[0] = node10_out;
16     R2[0] = node11_out;
17     if (phase == 2)
       then
18         keystream[i] = node12_out;
       end
   end
```

The software generation engine of RunStream generates a single-threaded, untimed, sequential C model of the stream cipher with necessary libraries and scripts. Some of its additional features are highlighted.

— **NIST Test Suite**: RunStream has the NIST test suite [NIST 2001] integrated with it to characterize the statistical qualities of PRNGs. It serves as a first step in determining the suitability of a PRNG used for cryptographic purposes. Fig. 13 gives a GUI snapshot of RunStream for the selection and parameterization of various statistical tests available for execution as per the user wishes.
— **Verification**: RunStream generates a verification environment for validating the generated model according to the user specified test vectors. For known/published stream

ciphers, the test vectors come along with the proposal, while for new proposals, the test vectors are generated once a working ANSI C model is up and running. Therefore, test vectors are an optional input to RunStream and should be given if known before hand. Without defined test vectors, the verification switches may be turned off by the user.

— **Performance Profiling**: The user may enable a performance profiling environment in the generated software implementation to evaluate encryption speed (in seconds, cycles/byte) of the cipher design. Provision of encrypting bulk data from random plaintext for monitoring data randomness is provided.

4.2. Hardware Generation Engine

The hardware generation engine generates a complete working model of the stream cipher in synthesizable *Verilog HDL* along with a testbench. Fig. 8 shows the architecture of the cipher. The toplevel module is $Testbench$ having the stream cipher or Design under Test (DUT) instantiated in it. The input to the DUT is the KIV vector (after any pre-initialization manipulation, *init* in Fig. 1), other than clk and $reset$ signals. The output is the $keystream$ along with a single bit high-asserted signal for its validation ($valid$). The $granularity$ of the cipher determines the width of KIV and the $keystream$ signals.

The $controller$ and $datapath$ are defined as separate modules, interacting with each other through the $phase$ signal. All definitions are kept in a separate *header* file.

— **Controller**: The controller keeps track of current phase using two registers; *counter* and *phase*. The size of counter is taken up as $ceil(\log_2(Pi\_cnt))$ bits, where Pi_cnt is the cycle count of the longest of the phases. For Trivium $P1\_cnt = 288 \times 4 = 1152$ determines counter size to be 11 bits. During *phase0* (key setup), the FSR registers are initialized using a 288 bit vector comprising of user specified key and IV values ($P0\_cnt = 1152$). As soon as the counter hits the phase count for the current phase, the phase register is incremented and counter is cleared (as shown in Fig. 8). For all stream ciphers, a valid output keystream is generated during the *last* phase of operation. Hence for Trivium, the *valid* signal is asserted when *phase* register is incremented to 2.

— **Datapath**: The datapath of stream cipher comprises of sequential elements and a mesh of nodes, for Trivium the architectural details are shown in Fig. 5. In case of multiple nodes, all operational nodes are generated as separate $modules$ in HDL with interface and functionality, as per the user specified. The *datapath* module has $instantiations$ of these node modules. The sequential elements are local registers in *datapath* module that are declared, reset, clocked and updated as specified by their type, granularity and size. For Trivium, the randomization phase or *phase1* and outputs of *node6*, *node7*, *node8* multiplexers are passed as inputs to *R0*, *R1* and *R2*, respectively. During the last phase, i.e., *phase2*, one of the terminal node, i.e., *Node12* generates valid keystream that is output of the cipher module.

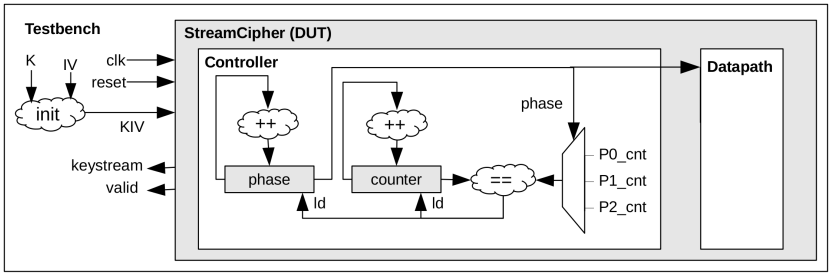

Fig. 8. HDL simulation environment for a RunStream generated stream cipher implementation

Table III. Sequential information configuration for stream ciphers taken up by RunStream

	Register Arrays (bits)	FSM Registers (bits)	Cipher State (bits)	Initialization (cycles) Key setup+Randomization
A-5/1	3 LFSRs (19, 22, 23)	-	64	86 + 100 =186
A-5/2	4 LFSRs (19, 22, 23, 17)	-	81	86 + 100 =186
E0	4 LFSRs (25, 31, 33, 39)	2 (4)	132	200 + 128 =328
Grain-v1	1 LFSR (80), 1 NFSR (80)	-	160	160 + 160 =320
Grain-128	1 LFSR (128), 1 NFSR (128)	-	256	256 + 256 =512
Grain-128a	1 LFSR (128), 1 NFSR (128)	-	256	256 + 256 =512
Trivium	3 LFSR (93, 84, 111)	-	288	288 + 1152 =1440
MICKEY 2.0	2 Jump regs (2 x 100)	-	160	160 + 100 =260
MICKEY-128	2 Jump regs (2 x 160)	-	320	256 + 160 =416
RC4	1 reg array (8 x 256)	2 (16)	32K	256 + 256 =512
ZUC	1 NFSR (16 x 31)	2 (64)	560	16 + 32 =48
SNOW 3G	1 NFSR (16 x 32)	3 (96)	608	16 + 32 =48

Table IV. Operational nodes information for RunStream configuration of stream ciphers

	Nodes															
	AND	OR	XOR	NOT	Majority	Multiplexer	Shift	Rotate	S-Box	Gmix	GAdd	bit Mask	Bit Reorder	Add	Compare	Total
A-5/1	-	-	7	-	1	3	-	-	-	-	-	-	-	-	-	11
A-5/2	-	-	9	3	4	4	-	-	-	-	-	-	-	-	-	20
E0	1	-	6	-	-	4	1	-	1	-	-	-	-	1	-	14
Grain-v1	19	-	6	-	-	2	-	-	-	-	-	-	-	-	-	27
Grain-128	12	-	6	-	-	2	-	-	-	-	-	-	-	-	-	20
Grain-128a	15	-	6	-	-	2	-	-	-	-	-	-	-	-	-	23
Trivium	3	-	7	-	-	3	-	-	-	-	-	-	-	-	-	13
MICKEY 2.0	2	-	10	-	-	6	3	-	-	-	-	-	-	1	-	22
MICKEY-128	2	-	10	-	-	6	3	-	-	-	-	-	-	1	-	22
RC4	-	-	-	-	-	9	-	-	-	-	-	-	-	7	6	22
ZUC	-	-	5	-	-	3	1	8	2	-	2	-	6	2	-	29
SNOW 3G	-	-	7	-	-	4	2	-	4	2	-	-	-	2	-	21

The hardware generation engine of RunStream generates a synthesizable, hierarchical stream cipher HDL and testbench with necessary scripts that can be further used to carry out

— Simulations for design verification, gate-level simulation (post-synthesis) using verification tools.
— Logic synthesis of the design for profiling critical parameters like the maximum clock frequency, chip area.
— Post-synthesis power consumption estimation by using back-annotation.

4.3. Algorithms Mapping

Using RunStream, we successfully mapped the diverse set of algorithms given in Table I. Table IV illustrates the configuration details of various stream ciphers undertaken for implementation by RunStream. The sequential information for these ciphers is given in Table III. Unlike the datapath of a cipher, the controller information of the stream ciphers changes only slightly from algorithm to algorithm. In the following discussion, we highlight only the distinguishing features of these ciphers as they are being undertaken by RunStream implementation methodology. Since RC4, ZUC and SNOW 3G are the only non-binary stream ciphers undertaken, we elaborate the mapping of one of them (SNOW 3G) in detail.

— **SNOW 3G**: The structural architecture and the mesh configuration of SNOW 3G [3G 2006] is shown in Fig. 9. The *R0* is a *16* element NFSR array with granularity of 32, while *R1*, *R2* and *R3* are 32-bit FSM registers, all clocked unconditionally.
The MUL_α and DIV_α operations [3G 2006] are carried out by *node0-2* and *node3-5*, respectively. The S-Boxes of *node0* and *node3* are 8×32 with a lookup table of 2^8 elements of 32 bits each. The S-Boxes in *node11* and *node13* are 8×8. The input word is divided into 4 bytes, each of which is transformed by an S-Box of 2^8 elements of 8 bits each. The result is concatenated and passed on to the next node (*node12* and *node14*) which is GF-multiplication. A valid output is generated from *node10* in the keystream generation phase.

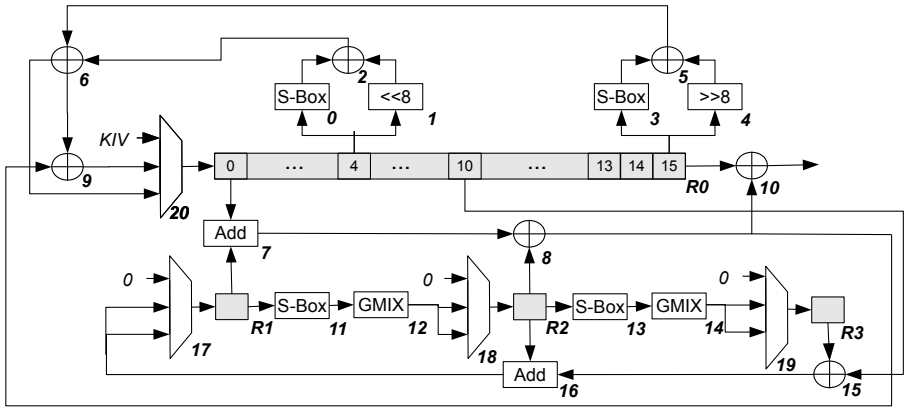

Fig. 9. SNOW 3G Architecture

— **A-5/1/2**: A-5/1 comprises of 3 LFSR registers, sum of which equates to a 64 bit state as given. The three LFSRs are irregularly clocked based on a $majority$ operation node, the 3 sink nodes for LFSR feedback are multiplexers while the linear feedback nature of the LFSRs is carried out by 7 XOR nodes, totally 11 nodes in all. A-5/2 follows a similar structure as A-5/1, the irregular clocking is carried out by 4 majority nodes.
— **E0**: E0 stream cipher has 2 FSM registers of 2 bits each, other than FSRs, hence the cipher state evaluates to a total of 132 bits.
— **Grain**: The Grain family of binary additive stream cipher have two FSRs, one with linear feedback and the second with nonlinear feedback. The FSRs are updated and the keystream is generated by means of some lightweight Boolean functions implemented by XOR and AND gates only. Both Grain-128 and Grain-128a have 256 bits state, Grain-128a has additional logic to calculate message authentication code in addition to generating a keystream.
— **MICKEY**: This feature of nodes operating on word sizes independent of the granularity of the FSR elements is particularly useful for MICKEY family of stream ciphers. MICKEY 2.0 has two jump registers of 100 bits each, whose each bit may be updated per clock cycle. The variable wordsize of nodes emancipates the user by allowing him to define operations to be performed on the whole register instead of each element. MICKEY 2.0 requires 22 nodes for its definition, 7 of which are operations performed on 1-bit inputs while the rest operate on 100 bits. For MICKEY-128, the node count remains the same, the 15 nodes operate on 160 bits operations instead of 100 bits.
— **RC4**: The T-function of RC4 during the $phase0$ and $phase1$ requires i update, followed by j update (taking updated value of i) along with a swap of two values of register array ($R0$) taking updated values of i and j. To enable a single cycle execution of T-function of these phases, the i and j updates are pre-calculated for the next cycle. All the corner

conditions are evaluated and handled using the 6 compare nodes and multiplexers. A total of 22 operational nodes are required for the operation.

— **ZUC**: For ZUC stream cipher, the FSR elements have 31 bits granularity, while the two FSM registers are 32 bits each. This mismatch is incorporated by explicitly defining the granularity of each node edge, that may not match the cipher granularity.

Using RunStream a very wide range of stream ciphers may be modeled. The first prototype of our tool models the sequential states (FSRs, FSM registers) as D-flip flops. Some stream ciphers with large states (e.g., RC4 [Schneier 1996] requiring 2K bits of state information), external SRAMs may be employed. For an efficient implementation of such ciphers, pipelining and optimized SRAM ports utilization is exploited. These optimizations are currently not offered by RunStream, however an extension to offer SRAMs access optimizations is on our roadmap.

5. RESULTS AND ANALYSIS

This section summarizes the ASIC performance results for various HDL stream cipher implementations, generated by RunStream. For all the stream ciphers undertaken, HDL generation, synthesis and benchmarking have been carried out using the same design flow. RunStream was configured in accordance with the parameters of each of the stream ciphers. The generated C based software implementation was tested for correctness against the available test vectors. RTL verification was carried out using Mentor Graphics ModelSim (version 10.2c).

Table V. Design results for highest operating frequencies @65nm CMOS

Cipher	Max. Freq. (MHz)	Interface bits	Area (GE)			Power (μW)		
			Comb.	Sequential	Total	Dynamic	Leakage	Total
A-5/1	1000	1	237.0	370.0	607.0	589.0	3.5	592.5
A-5/2	1000	1	263.0	455.0	718.0	1327.0	4.5	1331.5
E0	1000	1	100.5	690.0	790.5	390.8	5.0	395.8
Grain-v1	1000	1	154.5	850.0	1004.5	2272.0	7.1	2279.1
Grain-128	1000	1	134.2	1330.0	1464.3	2277.0	9.6	2286.6
Grain-128a	1000	1	144.7	1335.0	1479.8	2276.0	9.8	2285.8
Trivium	1000	1	108.0	1505.0	1613.0	3813.0	11.5	3824.5
MICKEY 2.0	1000	1	1027.2	1065.0	2092.3	4211.0	12.3	4223.3
MICKEY-128	1000	1	1591.5	1665.5	3257.0	5833.0	15.2	5848.2
RC4	1000	8	33920.6	13596.2	47516.8	1490.3	243.6	1733.9
ZUC	500	32	8404.0	3536.7	11940.7	3360.0	78.5	3438.5
SNOW 3G	1000	32	8136.0	3723.7	11859.7	17110.0	90.3	17200.3

We used *Synopsys Design Compiler* (version G-2012.06) with the Faraday standard cell libraries in topographical mode to carry out synthesis of HDL. The foundry typical values of 1.2 Volt for the core voltage and 25°C for the temperature for UMC 65 nm CMOS logic SP/RVT Low-K process were used. Each of the stream cipher accelerator was synthesized and profiled for area, power consumption and maximum frequency. To profile for the highest operating frequency, each design was repeatedly synthesized using *compile_ultra* option, in an incremental fashion with increasing clock frequency as long as no timing violation was reported. The synthesis was driven by throughput maximization with the *max area* constraint set to 0. As the design failed due to timing violation, we refined the frequency in jumps of 50 MHz to look for the frequency above which no valid design could be synthesized. All synthesis runs were carried out to optimize area. The area figures were converted to equivalent NAND gates. The power consumption is estimated by *Synopsys Primetime* (version 2009.12) based on gate-level netlist switching activity by back annotation. All the power estimations include the initialization and runup phases of stream ciphers along with generation of 1024 bits of keystream.

Table V shows the area and power consumption estimates for different stream cipher for the highest operating frequency, i.e., no valid design could be synthesized at frequency above 0.5 GHz for ZUC stream cipher. The natural choice of interface bits width equal to

the wordsize of each stream cipher is taken up. Area estimates of sequential and combinational logic, contributing to the core area are given. Understandably, RC4 has the highest area, owing to its 32 Kbits state (sequential logic).

The power consumption of a core on a feature size design is a function of the complexity of the design and the clock frequency. Static/leakage power is proportional to the area, the dynamic power contributes majorly to the total power consumption of the design. The maximum power dissipation occurs when the circuit is operated at its maximum frequency. Also, the area for any design is largest for the highest possible operating frequency of the design. Consequently, the design results are re-calculated for 10 MHz (10Mbps throughput for wireless LAN applications) and 100 KHz (for typical RFID applications), as given in Table VI and Table VII, respectively.

Table VI. Design results for 10 MHz @65nm CMOS

Cipher	Interface bits	Throughput (Mbps)	Area (GE)			Power (μW)		
			Comb.	Sequential	Total	Dynamic	Leakage	Total
A-5/1	1	10	237.0	370.0	607.0	11.7	3.7	15.4
A-5/2	1	10	263.0	455.0	718.0	13.3	4.5	17.8
E0	1	10	100.5	690.0	790.5	15.6	5.5	21.1
Grain-v1	1	10	152.5	850.0	1002.5	8.8	6.5	15.3
Grain-128	1	10	134.2	1330.0	1464.3	12.8	9.7	22.4
Grain-128a	1	10	144.7	1335.0	1479.8	12.8	9.8	22.5
Trivium	1	10	107.2	1505.0	1612.3	13.4	10.6	24.0
MICKEY 2.0	1	10	1023.2	1065.0	2088.3	12.7	11.2	23.9
MICKEY-128	1	10	1573.5	1665.0	3238.5	14.0	14.1	28.1
RC4	8	80	21543.7	10610.0	32153.8	8.8	143.3	152.1
ZUC	32	320	7023.0	2865.0	9888.0	156.0	56.7	212.7
SNOW 3G	32	320	6977.7	3105.0	10082.7	146.4	53.5	199.9

Table VII. Design results for 100 MHz @65nm CMOS

Cipher	Interface bits	Throughput Kbps	Area (GE)			Power (μW)		
			Comb.	Sequential	Total	Dynamic	Leakage	Total
A-5/1	1	100	237.0	370.0	607.0	0.12	3.68	3.80
A-5/2	1	100	263.0	455.0	718.0	0.13	4.53	4.66
E0	1	100	100.5	690.0	790.5	0.15	5.47	5.62
Grain-v1	1	100	152.5	850.0	1002.5	0.09	6.47	6.56
Grain-128	1	100	134.2	1330.0	1464.3	0.13	9.65	9.78
Grain128a	1	100	144.7	1335.0	1479.8	0.13	9.77	9.90
Trivium	1	100	107.2	1505.0	1612.3	0.13	10.60	10.73
MICKEY 2.0	1	100	1023.2	1065.0	2088.3	0.04	9.50	9.54
MICKEY-128	1	100	1573.5	1665.0	3238.5	0.04	10.10	10.14
RC4	8	800	21543.5	10610.0	32153.5	0.09	143.30	143.39
ZUC	32	3200	7023.0	2865.0	9888.0	1.56	56.70	58.26
SNOW 3G	32	3200	6977.7	3105.0	10082.7	1.46	53.50	54.96

As the operating frequency is lowered from the highest operating frequency to 10Mhz the area estimates decrease. For simple bit-oriented ciphers the area estimates do not decrease, for RC4, SNOW 3G and ZUC however, the reduction is drastic. A lower power consumption is also accompanied with the decrease in operating frequency.

From a set of *basic* metrics of stream ciphers implementations (area, power, operating frequency, interface, initialization cycles), a set of *derived* metrics are calculated for performance comparison as given.

— *Throughput* is the sustainable rate of data in keystream generation phase. It is the product of the operating frequency and the interface bits produced per cycle.
— *Energy / bit* is the ratio of the total power consumption and the throughput. A low number implies an economical energy consumption and vice versa.

— *Area-time* is the product of the time taken to generate each keystream bit and the area of the cipher. The reciprocal metric is *Throughput Per Area Ratio (TPAR)*. TPAR is specifically critical for high performance applications and is maximum at the highest operating frequency.

Some additional derived metrics, critical for *lightweight* encryption or RFID applications are given.

— *Power-area-time* is a triple product metric that gives a quantitative comparison of all three critical resources of VLSI design, i.e., compactness, throughput and power consumption.
— *latency* is the response time of a cipher (initialization phase latency) which is more critical for RFID applications. Every time a new key or IV is introduced, the system must bear the latency of initialization before valid keystream generation can start.
— *Power-latency* is the product of power consumption and latency time.

Table VIII. Derived metrics for highest operating frequencies @65nm CMOS

	Throughput (Gbps)	Energy/bit (pJ/bit)	Area-Time (GE-μs)	Tput/Area (Kbps/GE)	Power-Area-Time (GE-nJ)
A-5/1	1	0.59	0.61	1647.45	0.36
A-5/2	1	1.33	0.72	1392.76	0.96
E0	1	0.40	0.79	1265.02	0.31
Grain-v1	1	2.28	1.00	995.52	2.29
Grain-128	1	2.29	1.46	682.94	3.35
Grain-128a	1	2.29	1.48	675.79	3.38
Trivium	1	3.82	1.61	619.96	6.17
MICKEY 2.0	1	4.22	2.09	477.95	8.84
MICKEY-128	1	5.85	3.26	307.03	19.05
RC4	8	0.22	5.94	168.36	10.30
ZUC	16	0.21	0.75	1339.95	2.57
SNOW 3G	32	0.54	0.37	2698.20	6.37
Better is	*Higher*	*Lower*	*Lower*	*Higher*	*Lower*

Table IX. Derived metrics for 100 MHz operating frequency @65nm CMOS

	Energy/bit (nJ/bit)	Power-Area -Time (GE-μJ)	Init. cycles	Latency μs	Power-Area Latency (μJ-GE)	Power-Latency (nJ)
A-5/1	0.04	0.02	186	1860	0.00	7.06
A-5/2	0.05	0.03	186	1860	0.01	8.67
E0	0.06	0.04	328	3280	0.01	18.44
Grain-v1	0.07	0.07	320	3200	0.02	20.99
Grain-128	0.10	0.14	513	5130	0.07	50.16
Grain-128a	0.10	0.15	513	5130	0.08	50.77
Trivium	0.11	0.17	1440	14400	0.25	154.57
MICKEY 2.0	0.10	0.20	260	2600	0.05	24.81
MICKEY-128	0.10	0.33	416	4160	0.14	42.20
RC4	0.18	5.76	512	5120	23.61	734.15
ZUC	0.02	0.18	48	480	0.28	27.96
SNOW 3G	0.02	0.17	48	480	0.27	26.38
better is	*Lower*	*Lower*	*Lower*	*Lower*	*Lower*	*Lower*

For highest operating frequency these derived matrices are given in the Table VIII. Considering a performance comparison between SNOW 3G and ZUC, ZUC outperforms by having a lower Power-Area-Time ratio. For eSTREAM finalists, Grain-v1 outperforms the other proposals. For 100 MHz operating frequency, the derived metrics are given in the Table IX. The latency is compounded by lower operating frequency as it translates to higher latency delay (shown in latency column). Here too, Grain-v1 is clearly a winner.

The derived metrics for highest operating frequencies are graphically shown in Fig. 10. For a fair comparison we choose area-efficiency and energy per bit as our figure of merits. For RFID applications, Fig. 11 shows core areas plotted against power-latency of the designs, both lower being better (binary stream ciphers plotted only).

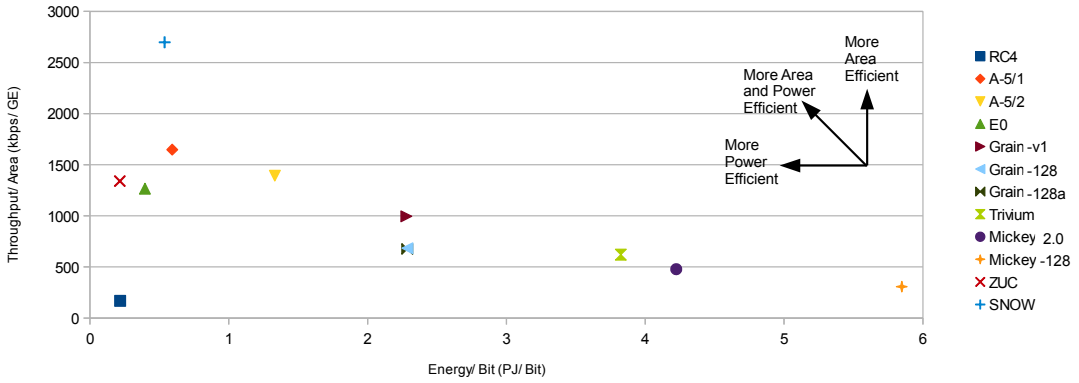

Fig. 10. Design performance metrics for highest operating frequencies @65nm CMOS

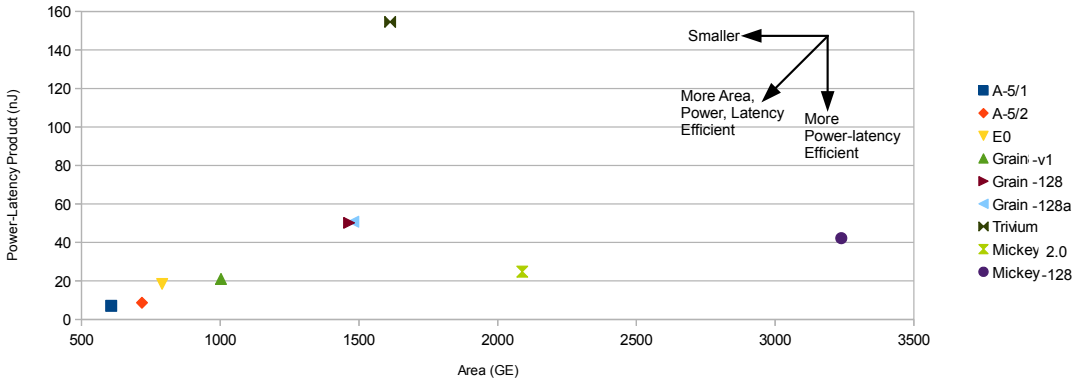

Fig. 11. Performance for RFID applications for 100 KHz @65nm CMOS

5.1. Comparison with Manual Implementations

Since the hand-crafted HDL implementations for any algorithm are more optimized compared to the tool-generated ones, for comparison of RunStream generated HDL we take up the manually written reported implementations of the stream ciphers and compare their efficiency. For RC4, we take up the open cores implementation [OpenCores-RC4 2013] that defines its internal state as D-flipflops. The two implementations are synthesized under *same* CMOS technology, synthesis tool and constraints. The results are given in Table X. The RunStream generated RTL has about 5% more area overhead in comparison. This is due to a single cycle per keystream expansion iteration ($phase1$) in RunStream RC4 implementation. The open cores RC4 implementation [OpenCores-RC4 2013] instead takes 2 cycles per iteration, though it has a relaxed critical path but increased initialization latency. The area overhead for RunStream RC4 is the price paid for this lower initialization latency.

When HDL implementation of a VLSI design is not available, a fair post-synthesis comparison of two designs has some inherent difficulties. The results of different CMOS synthesis technologies do not match, even with the same synthesis technology, different vendor cell libraries generate different implementations. Additionally, different settings, constraints or versions of synthesis tools generate differences in performance of the implementations. Moreover, the best case, typical case, worst case choice is often not specified for published results. For a fair comparison of RunStream generated RTL, we strive to have an environment as close to the one for hand-crafted reported implementations as possible. We re-synthesize our RunStream RTL using the same synthesis technol-

Table X. Resource comparison of RunStream implementations with others

Algorithm name	keystream bits/cycle	Initialization cycles	Op. Freq. (MHz)	Throughput (Mbps)	Area (GE) μm^2	Area (GE) (GE)
RunStream generated RTL, 65nm CMOS						
RC4	1	512	10	80	41156.80	32153.8
[OpenCores-RC4 2013], 65nm CMOS						
RC4	1	768	10	80	38931.20	30415.00
RunStream generated RTL, 90nm CMOS						
A-5/1	1	186	685	685	1730.29	551.75
[Gaj et al. 2007], 90nm CMOS						
A-5/1	1	186	685	685	1985	-
RunStream generated RTL, 90nm CMOS						
ZUC	32	48	18.75	600	32046.00	10218.75
[Traboulsi et al. 2012], 90nm CMOS						
ZUC	32	48	18.75	600	-	14000
RunStream generated RTL, 65nm CMOS						
ZUC	32	48	500	16000	15284.16	11940.7
[CLP-410 2011], 65nm CMOS						
ZUC	32	48	500	16000	-	10-13K
RunStream generated RTL, 65nm CMOS						
SNOW 3G	32	48	943	30176	13900.48	10859.7
[SNOW3G1 2011], 65nm CMOS						
SNOW 3G	32	48	943	30176	-	8.9K
RunStream generated RTL, 130nm CMOS						
Graini-v1	1	321	724.6	724.6	5288.78	1021
Grain-128	1	513	925.9	925.9	7676.76	1482
Trivium	1	1314	327.9	327.9	9272.20	1790
MICKEY 2.0	1	261	454.5	454.5	11494.42	2219
MICKEY-128	1	417	413.2	413.2	17829.56	3442
[Good and Benaissa 2008a], 130nm CMOS						
Grain-v1	1	321	724.6	724.6	6702.92	1294
Grain-128	1	513	925.9	925.9	9712.50	1875
Trivium	1	1314	327.9	327.9	13364.40	2580
MICKEY 2.0	1	261	454.5	454.5	16513.84	3188
MICKEY-128	1	417	413.2	413.2	26102.02	5039

ogy node (with possibly different vendor libraries), same synthesis tool (with different version) and same operating frequency. The power however, cannot be reliably scaled between different processes and libraries and is not discussed.

For A/5-1, the only reported VLSI implementation has area estimates reported in μm^2 [Gaj et al. 2007]. RunStream results in a lower area budget of $1790 \mu m^2$ against their area figure of $1985 \mu m^2$ (12% higher). For E0 and A/5-2, no CMOS implementation results have been reported. The fastest FPGA implementation for E0 [Galanis et al. 2004] is synthesized, placed and routed, using Xilinx FPGA device Virtex 2-2V250FG256 with a throughput of 189 Mbps.

For ZUC we take up two VLSI implementations, one from academia at 90nm [Traboulsi et al. 2012] and the other from Elliptic Technologies at 65nm [CLP-410 2011]. RunStream generated ZUC implementation matches closely with the ZUC core from Elliptic Technologies in area estimates. For SNOW 3G too, we took up the IP Cores Inc. implementation at 65nm and compared the RunStream generated RTL against the same operating frequency and technology library. RunStream SNOW 3G results in around 22% more area. Its hard to identify the reason of this overhead since the internal working of these commercial cores for ZUC and SNOW 3G is not known. However, the synthesis for our implementations show that the major area contribution is due to the S-Boxes. Moreover, the implementation of S-Boxes is not unique and may vary widely in an area-throughput spectrum from a simplistic read-only LUT (high performance) to an equivalent combinational implementation (area efficient).

A detailed study of HDL manual implementations of eSTREAM ciphers at various phases of selection has been carried out by T. Good et. el. [Good et al. 2006; Good and Benaissa 2007; 2008b; 2008a]. Their goal was to evaluate the suitability of the proposals for maximum throughput, power consumption and area compactness for RFID and LAN applications. They targeted a 130nm process using a standard cell library produced by

Faraday. Table X compares their implementation results against the RunStream generated implementations, synthesized using 130nm standard CMOS. For the same operating frequencies (and the same throughput), the area estimates for RunStream remain $20 - 30\%$ lower. This apparent improvement of RunStream results could be attributed to differences in the synthesis environments. RunStream strives to facilitate fast and reliable prototyping of stream ciphers and endeavors to come *close* to hand written implementations.

Worth mentioning is the fact that the performance based enumeration of all the 5 eSTREAM ciphers remains *exactly* the same as [Good and Benaissa 2008a]. Grain-v1 outperforms both MICKEY and Trivium in terms of TPAR and energy per bit, when compared for highest operating frequency (Fig. 10). At 10 Mbps data rate too, a similar enumeration trend for the eSTREAM ciphers is seen as in [Good and Benaissa 2008a]. For RFID applications too, our performance based ordering for Grain-v1, Grain-128, Grain-128a, Trivium, MICKEY 2.0 and MICKEY-128 in Fig. 11 conforms *completely* to [Good and Benaissa 2008a].

The impact of choosing different building blocks, on the hardware performance of stream ciphers can be seen from Fig. 10 and Fig. 11. The simpler primitives (LFSRs and NFSRs) render a superior performance of Grain versions and Trivium, compared to the jump registers based MICKEY versions. Grain-v1 outperforms Trivium in energy consumption too, possibly due to a much smaller state size (Fig. 10). The MICKEY versions are clearly the least energy efficient (Fig. 10) and the least area efficient (Fig. 11). Due to the large initialization latency that trivium requires for setup, it's the least power-latency efficient cipher of eSTREAM for RFID applications (Fig. 11), while Grain-v1 outperforms the other four ciphers in terms of area compactness while MICKEY-128 has the highest area overhead.

6. CONCLUSION AND FUTURE WORKS

We present RunStream, a novel rapid-prototyping framework for stream ciphers. We identify the configuration space of stream ciphers, that is taken up by the RunStream for design capture. A design validation is carried out, after whose successful completion, an optimized software and HDL implementations are generated. We took up all the hardware portfolio stream ciphers of eSTREAM, SNOW 3G, ZUG and other noticeable stream ciphers for generation using RunStream. Equitable comparisons for area-throughput-power were carried out. Our results rival the respective best available handwritten IP cores.

Indeed, the long-term goal of the project is to have a tool that can generate the optimized implementation of a stream cipher over a variety of platforms, including microcontrollers, FPGAs, ASICs, GPPs, GPUs etc. This requires inculcating some device-specific optimizations for a range of devices catering to each of these platform categories. This draft focused and benchmarked the ASIC implementations only. Additionally, We are enthusiastic to extend it in various directions.

— Inclusion of cryptanalytic tools for stream ciphers.
— Enhancing RunStream to offer architectural optimizations like parallel implementations (generating higher throughput), SRAM specific memory access optimizations [Khalid et al. 2014], etc.

REFERENCES

SNOW 3G. 2006. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 and UIA2. Document 2: SNOW 3G 3G Specification. ETSI/SAGE Specification, Version: 1.1. (2006).

3GPP. 1999. General Report on the Design, Speification and Evaluation of 3GPP Standard Confidentiality and Integrity Algorithms. (1999). http://www.3gpp.org/ftp/tsg_sa/WG3_Security/_Specs/33908-300.pdf 3G TR 33.908 version 3.0.0 Release.

AES. 1997. Announcing development of a federal information processing standard for advanced encryption standard. National Institute of Standards and Technology, Docket No. 960924272-6272-01, RIN 0693-ZA13, January 2, 1997.. (1997). http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt

Martin Ägren, Martin Hell, Thomas Johansson, and Willi Meier. 2011. Grain-128a: A new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing* 5, 1 (2011), 48–59.

Ajax 2009. The HercuLeS high-level synthesis tool. (2009). http://www.nkavvadias.com/hercules.

Steve Babbage and Matthew Dodd. 2006. The stream cipher MICKEY 2.0. *ECRYPT Stream Cipher, available at http://www. ecrypt. eu. org/stream/p3ciphers/mickey/mickey_p3. pdf* (2006).

Steve Babbage and Matthew Dodd. 2008. The MICKEY stream ciphers. In *New Stream Cipher Designs*. Springer, 191–209.

Elad Barkan, Eli Biham, and Nathan Keller. 2003. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In *Advances in Cryptology-CRYPTO 2003*. Springer, 600–616.

Riddhipratim Basu, Subhamoy Maitra, Goutam Paul, and Tanmoy Talukdar. 2009. On Some Sequences of the Secret Pseudo-random Index j in RC4 Key Scheduling. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 18th International Symposium, AAECC-18 2009, Tarragona, Catalonia, Spain, June 8-12, 2009. Proceedings (Lecture Notes in Computer Science)*, Maria Bras-Amorós and Tom Høholdt (Eds.), Vol. 5527. Springer, 137–148. DOI:http://dx.doi.org/10.1007/978-3-642-02181-7_15

Côme Berbain, Henri Gilbert, and Alexander Maximov. 2006. Cryptanalysis of Grain. In *Fast Software Encryption*. Springer, 15–29.

S. I. G. Bluetooth. 2001. Specification of the Bluetooth System.Version 1.1. (2001). Retrieved February 22, 2001 from www.inf.ethz.ch/personal/hvogt/proj/btmp3/Datasheets/Bluetooth_11_Specifications_Book.pdf

Marc Briceno, Ian Goldberg, and David Wagner. 1999. A pedagogical implementation of the GSM A5/1 and A5/2 voice privacy encryption algorithms. (1999). Retrieved October 29, 1999 from http://cryptome.org/gsm-a512.htm

Cadence 2009. Cadence C-to-Silicon Compiler: Next-generation high-level synthesis for design and verification. (2009). http://www.cadence.com/products/sd/silicon_compiler.

CAESAR. 2012. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. (2012). http://competitions.cr.yp.to/caesar.html

CLP-410. 2011. Elliptic Technologies Inc. CLP-410: ZUC Key Stream Generator. (2011). Retrieved retrieved on August 5, 2011 from http://elliptictech.com/products-clp-410.php

CRYPTREC. 2003. CRYPTREC: Cryptography Research and Evaluation Committees. (2003). http://competitions.cr.yp.to/cryptrec.html

Christophe De Canniere, Orr Dunkelman, and Miroslav Knežević. 2009. KATAN and KTANTANa family of small and efficient hardware-oriented block ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 272–288.

Christophe De Canniere and Bart Preneel. 2005. Trivium specifications. eSTREAM. *ECRYPT Stream Cipher Project, Report* 30 (2005), 2005.

Itai Dinur, Tim Güneysu, Christof Paar, Adi Shamir, and Ralf Zimmermann. 2011. An Experimentally Verified Attack on Full Grain-128 Using Dedicated Reconfigurable Hardware. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings (Lecture Notes in Computer Science)*, Dong Hoon Lee and Xiaoyun Wang (Eds.), Vol. 7073. Springer, 327–343. DOI:http://dx.doi.org/10.1007/978-3-642-25385-0_18

Itai Dinur and Adi Shamir. 2009. Cube Attacks on Tweakable Black Box Polynomials. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings (Lecture Notes in Computer Science)*, Antoine Joux (Ed.), Vol. 5479. Springer, 278–299. DOI:http://dx.doi.org/10.1007/978-3-642-01001-9_16

eSTREAM. 2008. eSTREAM: The ECRYPT Stream Cipher Project. (2008). Retrieved March 7, 2012 from http://www.ecrypt.eu.org/stream

Kris Gaj, J Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y Brewster. 2010. Athena-automated tool for hardware evaluation: Toward fair and comprehensive benchmarking of cryptographic hardware using fpgas. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 414–421.

Kris Gaj, Gabriel Southern, and Ramakrishna Bachimanchi. 2007. Comparison of hardware performance of selected Phase II eSTREAM candidates. In *State of the Art of Stream Ciphers Workshop (SASC 2007), eSTREAM, ECRYPT Stream Cipher Project, Report*, Vol. 26. 2007.

Michalis D Galanis, Paris Kitsos, Giorgos Kostopoulos, Nicolas Sklavos, O Koufopavlou, and Costas E Goutis. 2004. Comparison of the hardware architectures and FPGA implementations of stream ciphers. In *The 11th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2004*. IEEE, 571–574.

GAUT 2007. GAUT - High-Level Synthesis tool From C to RTL. (2007). http://hls-labsticc.univ-ubs.fr.

Ian Goldberg, David Wagner, and Lucky Green. 1999. The real-time cryptanalysis of A5/2. *Rump session of Crypto* 99 (1999), 239–255.

Tim Good and Mohammed Benaissa. 2007. Hardware results for selected stream cipher candidates. *State of the Art of Stream Ciphers* (2007), 191–204.

Tim Good and Mohammed Benaissa. 2008a. ASIC hardware performance. In *New Stream Cipher Designs*. Springer, 267–293.

Tim Good and Mohammed Benaissa. 2008b. Hardware performance of eStream phase-III stream cipher candidates. In *Proc. of Workshop on the State of the Art of Stream Ciphers (SACS08)*.

Tim Good, William Chelton, and Mohammed Benaissa. 2006. Review of stream cipher candidates from a low resource hardware perspective. *SASC 2006 Stream Ciphers Revisited* (2006), 125.

Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. 2006. A stream cipher proposal: Grain-128. In *IEEE International Symposium on Information Theory (ISIT 2006)*. Citeseer.

Martin Hell, Thomas Johansson, and Willi Meier. 2007. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing* 2, 1 (2007), 86–93.

Tor Helleseth, Cees JA Jansen, and Alexander Kholosha. 2006. Pomaranch-design and analysis of a family of stream ciphers. *SASC 2006 Stream Ciphers Revisited* (2006), 13.

Ekawat Homsirikamol and Kris Gaj. 2014. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 1–8.

Cees J. A. Jansen. 2004. Streamcipher design: Make your LFSRs jump. In *Proceedings of the The State of the Art of Stream Ciphers (ECRYPT Network of Excellence in Cryptology)*. 94–108.

Ayesha Khalid, Muhammad Hassan, Anupam Chattopadhyay, and Goutam Paul. 2013. RAPID-FeinSPN: A Rapid Prototyping Framework for Feistel and SPN-Based Block Ciphers. In *Proceedings of the Information Systems Security*. Springer Berlin Heidelberg, 169–190. An extended version of this has been accepted for publication in the *Journal of Cryptographic Engineering* (Springer), 2016.

Ayesha Khalid, Prasanna Ravi, Anupam Chattopadhyay, and Goutam Paul. 2014. One Word/Cycle HC-128 Accelerator via State-Splitting Optimization. In *Progress in Cryptology - INDOCRYPT 2014*. 283–303.

Yi Lu, Willi Meier, and Serge Vaudenay. 2005. The conditional correlation attack: A practical attack on bluetooth encryption. In *Advances in cryptology–CRYPTO 2005*. Springer, 97–117.

Subhamoy Maitra and Goutam Paul. 2008. Analysis of RC4 and proposal of additional layers for better security margin. In *Progress in Cryptology-INDOCRYPT 2008*. Springer, 27–39.

Alexander Maximov and Alex Biryukov. 2007. Two Trivial Attacks on Trivium. Cryptology ePrint Archive, Report 2007/021, (2007). http://eprint.iacr.org/.

Alexander Maximov and Dmitry Khovratovich. 2008. New State Recovery Attack on RC4. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings (Lecture Notes in Computer Science)*, David Wagner (Ed.), Vol. 5157. Springer, 297–316. DOI:http://dx.doi.org/10.1007/978-3-540-85174-5_17

Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. 1996. *Handbook of applied cryptography*. CRC press.

Mentor Graphics 1996. Handel-C Synthesis Methodology. (1996). http://www.mentor.com/products/fpga/handel-c.

Miodrag J. Mihaljevic, Sugata Gangopadhyay, Goutam Paul, and Hideki Imai. 2012. Generic cryptographic weakness of k-normal Boolean functions in certain stream ciphers and cryptanalysis of grain-128. *Periodica Mathematica Hungarica* 65, 2 (2012), 205–227. DOI:http://dx.doi.org/10.1007/s10998-012-4631-8

NESSIE. 2000. NESSIE: New European Schemes for Signatures, Integrity, and Encryption. (2000). https://www.cosic.esat.kuleuven.be/nessie/

NIST. 2001. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications,. (2001). Retrieved May 15, 2001 from www.csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf

OpenCores-RC4. 2013. OpenCores: RC4 Pseudo-random stream generator. (2013). Retrieved retrieved on May 25, 2015 from http://opencores.org/project,rc4-prbs

Ronald L Rivest and Jacob CN Schuldt. 2014. Spritz-a spongy RC4-like stream cipher and hash function. CRYPTO 2014 Rump Session. (2014). http://people.csail.mit.edu/rivest/pubs/RS14.pdf

T. Bjørstad. 2008. Cryptanalysis of Grain using Time / Memory / Data Tradeoffs. (2008). http://www.ii.uib.no/~tor/pdf/grain.pdf

Bruce Schneier. 1996. *Applied Cryptography*. John Wiley and Sons, Chapter 17, 397–398.

SHA-3. 2012. SHA-3 Cryptographic Secure Hash Algorithm Competition. (2012). http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

Adi Shamir. 2004. Stream ciphers: Dead or alive?. In *ASIACRYPT*. 78.

SNOW3G1. 2011. IP Cores Inc. SNOW 3G Encryption Core. (2011). Retrieved retrieved on August 5, 2011 from http://ipcores.com/Snow3G.htm

Synphony 2009. High-Level Synthesis with Synphony Model Compiler by Synopsys. (2009). http://www.synopsys.com/Tools/Implementation/FPGAImplementation/Pages/synphony-model-compiler.aspx.

Shadi Traboulsi, Nils Pohl, Josef Hausner, Attila Bilgic, and Valerio Frascolla. 2012. Power analysis and optimization of the ZUC stream cipher for LTE-advanced mobile terminals. In *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*. IEEE, 1–4.

Vivado 2012. Vivado Design Suite. (2012). www.xilinx.com/products/design-tools/vivado.html.

Hongjun Wu. 2008. The stream cipher HC-128. In *New Stream Cipher Designs*. Springer, 39–47.

Bartosz Zoltak. 2004. VMPC one-way function and stream cipher. In *Fast Software Encryption*. Springer, 210–225.

ZUC. 2011. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 and 128-EIA3. Document 2: ZUC Specification. ETSI/SAGE Specification, Version: 1.5. (2011).

Appendix

We present here some GUI snapshots of various tabs of RunStream tool. CRYKET (CRYptographic Kernels Toolkit) caters to rapid prototyping of various cryptographic functions while RunStream is an instance of it dealing with stream ciphers.

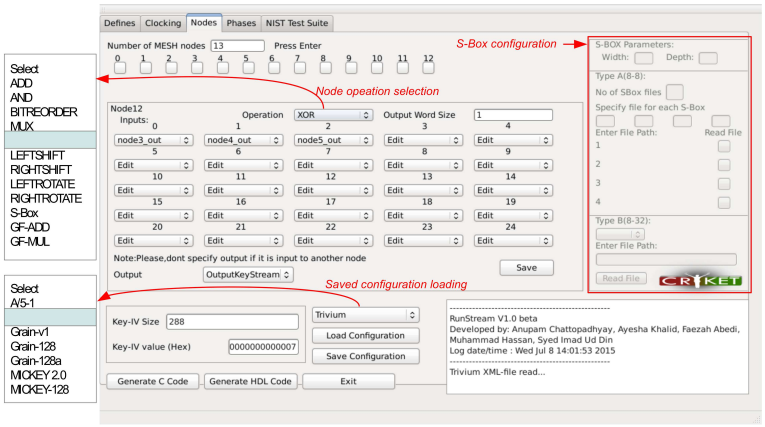

Fig. 12. Mesh nodes tab for RunStream showing input/outputs for Trivium's node12. Various Stream ciphers can be quickly loaded. Each mesh node can be configured to have an operation.

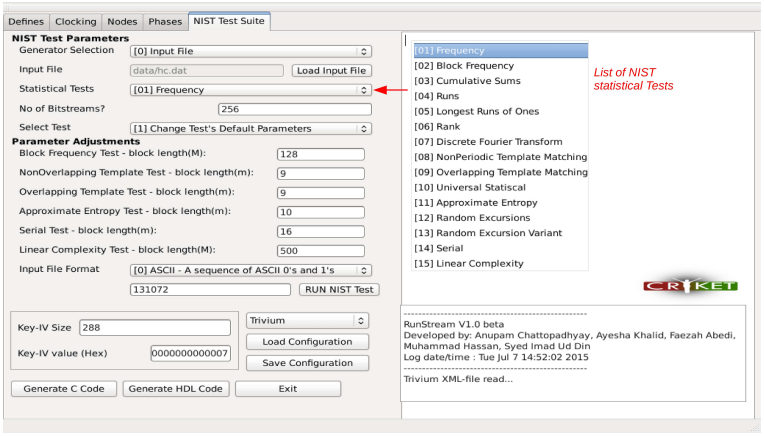

Fig. 13. NIST Test Suite tab in RunStream, various statistical tests can be chosen as per the user desires