Rapid hardware design for cryptographic modules with filtering structures over small finite fields*

Nusa Zidaric[0000−0001−9308−7392], Mark Aagaard[0000−0002−7331−3177], and Guang Gong[0000−0003−2684−9259]

University of Waterloo, Waterloo, ON, Canada N2L 3G1
{nzidaric,maagaard,ggong}@uwaterloo.ca

Abstract. This paper presents a design automation toolkit for hardware implementations of linear and non-linear feedback shift registers (FSRs). The toolkit is implemented in the GAP computer algebra system and generates both executable GAP code and VHDL for synthesizable hardware. To design an FSR, the user needs only to provide a template and instantiate a few parameters. The primary objects are LFSRs; NLFSRs; and arbitrary combinational functions, which are modelled as FILFUNs, for "filtering functions". Conventional feedback functions are modelled as univariate polynomials. More complex functions can be modelled as FILFUNs. The paper demonstrates the capabilities of the toolkit using the WG-7 and WG-8 keystream generators and the Grain v1 stream cipher. Less than 30 lines of GAP code are required to generate a complete datapath in VHDL.

Keywords: Feedback shift registers · filtering generators · rapid hardware design · stream ciphers.

1 Introduction

Feedback shift registers (FSR) play an important role in stream cipher design. A milestone in stream cipher design is the eSTREAM project [1], launched in 2004. All 3 hardware portfolio ciphers, Grain, MICKEY and Trivium, as well as the software portfolio cipher Sosemanuk, use FSRs. The stream cipher ACORN [2], a remaining round 3 CAESAR candidate [3], is based on 6 LFSRs. Last but not least, the two stream ciphers used for encryption and integrity of communications in mobile networks, Snow3G and ZUC[4,5], both use LFSRs. Another application area for LFSRs are the cyclic redundancy codes (CRC) used in many communication and data storage devices for error-detection. Less noticeable is the use of LFSRs in algorithms for finite field arithmetic: e.g. a serial circuit implementing multiplication by x followed by reduction modulo the field defining polynomial $f(x)$ can be implemented as a LFSR with the feedback $f(x)$.

* The authors would like to thank Dr. Alexander Konovalov from University of St. Andrews for his advice during FSR package implementation

This work presents a toolkit for rapid hardware design for modules with a filtering structure composed of feedback shift registers and filtering functions. The toolkit consists of two packages written in GAP [6]. The first package is called FSR, and allows creation, initialization and running of both LFSRs and NLFSRs. In addition, it implements filtering functions FILFUNs (multivariate polynomials). The generality of the FSR package enables the FSRs to be used directly as building blocks for many cryptographic modules. The second package is called FSRtoVHDL, and as its name suggests, it creates hardware modules using the VHDL language. By design of the two packages, each FSR object corresponds to a hardware entity, and the FSR objects themselves contain all the information needed for both their execution in GAP and their generation in hardware. Hardware is generated from a template, and for a simple cipher, such as WG-7, the entire datapath VHDL was created from just 30 lines of GAP code. The FSR package can be used to implement arbitrary primitives in GAP, which can operate e.g. as random number generators.

Two critical points in the design of this toolkit were modular thinking, inherent to hardware designs; and recognition and exploitation of structural similarities between LFSRs, NLFSRs and filters, from both mathematical and hardware perspective. A modular approach to design and implementation can open new perspectives and improve the initial design. Cryptographic primitives are always carefully selected to meet certain security requirements and an appropriate trade-off between security and hardware efficiency is desirable. It is thus imperative to be able to estimate the hardware cost of the design early on, and having the ability to quickly generate and synthesize hardware modules is very beneficial. Theoretical estimates of hardware cost, such as Hamming weights, often do not reflect the actual area and delay in hardware. Modern synthesis tools are powerful and quite successful at optimizing combinational logic, especially for small designs. Furthermore, they are aware of hardware resources available for the chosen target technology and can be instructed to optimize for a specific performance goal, e.g. high speed or small area. These optimizations can cause discrepancies between theoretical estimates and actual hardware, and the results can be quite surprising.

The toolkit is able to define (N)LFSRs and filters described by multivariate polynomials. It can execute these (N)LFSRs in GAP, generate traces, and generate VHDL code. Furthermore, it can define, execute, and generate VHDL datapaths for many hierarchical modules constructed from a set of (N)LFSRs and FILFUN filters.

2 Background and Related Work

2.1 Basic Terminology

To keep this section short, many details are omitted and the reader can refer to a number of sources such as [12,13,11].

Multivariate Polynomials Let $\mathcal{F} \equiv \mathbb{F}_q$ where q is a prime or a prime power. A multivariate function in t variables $x_0, x_1, \ldots, x_{t-1}$ is defined as follows:

$$f : \mathcal{F}^t \to \mathcal{F}$$
$$f(x_0, x_1, \ldots, x_{t-1}) = \sum_{\forall (i_0, i_1, \ldots, i_{t-1}) \in Z_q^t} c_{i_0, i_1, \ldots, i_{t-1}} x_0^{i_0} x_1^{i_1} \ldots x_{t-1}^{i_{t-1}} \tag{1}$$

with coefficients $c_{i_0,i_1,\ldots,i_{t-1}} \in \mathcal{F} \equiv \mathbb{F}_q$ and where $i_j \in Z_q$ for $0 \le j < t$. The sum in expression (1) runs over all possible monomials $x_0^{i_0} x_1^{i_1} \ldots x_{t-1}^{i_{t-1}}$, where $\forall x \in \mathcal{F} : x^q = x$. The expression on the r.h.s of equation (1) describes an univariate polynomial when $t = 1$, and a multivariate polynomial when $t > 1$. The degree of a monomial is defined as the sum of all its exponents (2) and the degree of the polynomial as the maximum degree of all its monomials (3). For readability, notation $m_{i_0,i_1,\ldots,i_{t-1}}$ is introduced for monomials:

$$m_{i_0,i_1,\ldots,i_{t-1}} = m(x_0, x_1, \ldots, x_{t-1}) = x_0^{i_0} x_1^{i_1} \ldots x_{t-1}^{i_{t-1}}$$
$$\deg(m(x_0, x_1, \ldots, x_{t-1})) = \sum_{j=0}^{t-1} i_j \tag{2}$$
$$\deg(f(x_0, x_1, \ldots, x_{t-1})) = \max_{\forall (i_0, i_1, \ldots, i_{t-1}) \in Z_q^t} \left\{ \deg(m_{i_0,i_1,\ldots,i_{t-1}}) \right\} \tag{3}$$

Based on the degree of the polynomial function, given by the expression (3), a multivariate polynomial is classified as constant for $\deg(f) = 0$, linear for $\deg(f) = 1$, and nonlinear function for $\deg(f) > 1$.

Also when $q = 2$, $f(x_0, x_1, \ldots, x_{t-1})$ is a boolean function in t variables, i.e.,

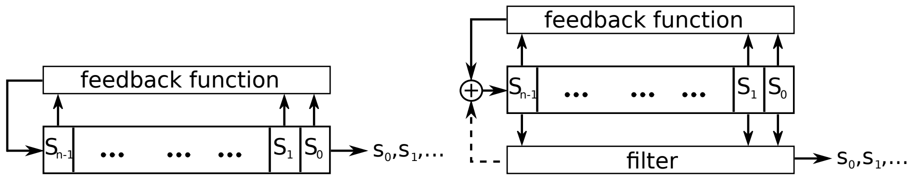
$$f(x_0, x_1, \ldots, x_{t-1}) = \sum_{\forall (i_0, i_1, \ldots, i_{t-1}) \in Z_2^t} c_{i_0, i_1, \ldots, i_{t-1}} x_0^{i_0} x_1^{i_1} \ldots x_{t-1}^{i_{t-1}} \tag{4}$$

Feedback Shift Registers (FSR) An n-stage shift register over a finite field $\mathcal{F}$ is an array of n registers (stages), denoted S_i, $i = n-1, \ldots, 0$. Each stage holds a value from the underlying finite field $\mathcal{F}$. The parameter n is also referred to as the *length of the FSR*. This memory array is shifted with each step $S_i \to S_{i-1}$ for $i = n-1, \ldots, 1$, and the vacant register S_{n-1} is updated with a new value obtained from the feedback function $f(x_0, \cdots, x_{n-1})$, a multivariate polynomial function in n variables, hence the name *feedback shift register* (FSR). One of the stages is used to generate the output and each time the FSR is clocked, that stage produces a new element $s_i \in \mathcal{F}$. In this way, the FSR produces a sequence of elements:

$$\underline{s} = \{s_k\} = s_0, s_1, s_2, \ldots \tag{5}$$

where s_k's satisfy the following recursive relation

$$s_{k+n} = f(s_k, s_{k+1}, \cdots, s_{k+n-1}), k = 0, 1, \cdots,$$



(a) n-stage FSR (b) n-stage FSR with a filter

Fig. 1: Top level schematic of an n-stage FSR

A simple schematic of an n-stage FSR is shown in Figure 1(a), with the output sequence produced by stage S_0.

The feedback f of the FSR is a polynomial function in $t = n$ variables $x_0, x_1, \ldots, x_{n-1}$ as defined in expression (1), whereby the variable x_i corresponds to the stage S_i, $i \in Z_n$, the residue ring modulo n where n is a positive integer. Integer n is the length of the FSR. A linear feedback yields a linear and a nonlinear feedback a nonlinear feedback shift register, which is an LFSR and an NLFSR, respectively.

In case of the LFSR, the feedback is given by $f(x_0, \cdots, x_{n-1}) = \sum_{i=0}^{n-1} c_i x_i$ which can be represented with an univariate polynomial

$$h(y) = y^n + \sum_{j=0}^{n-1} c_j y^j \tag{6}$$

where y^j corresponds to the stage S_j for $j \in Z_n$, and y^n to the new value computed by the feedback. Coefficients c_j, $j \in Z_n$ of the polynomial in (6) belong to the underlying field $\mathcal{F}$.

At any given moment, the stages of the FSR hold n values from the underlying finite field, and can be written as a vector of length n: $(s_0, s_1, \ldots, s_{n-1}) \in \mathcal{F}^n$. This vector is called the *state* of the FSR and the state right after loading the *initial state.* The output sequence $\underline{s}$ is completely determined by the feedback f and the initial state.

Filtering Generators A typical structure of a filtering generator is shown in Figure 1(b): it consists of a filter, i.e. a nonlinear multivariate polynomial function, applied to an LFSR with n stages. Let $(s_k, s_{k+1}, \ldots, s_{k+n-1}) \in \mathcal{F}^n$ be the kth state of the LFSR, $g(x_0, \cdots, x_{t-1})$, a multivariate polynomial in t variables, where $t \leq n$, and $(d_0, \cdots, d_{t-1})$, a selection of t tap positions in the state, i.e., $0 \leq d_0 < d_1 < \cdots < d_{t-1} < n$. The output sequence $\underline{a} = \{a_k\}$ is given by

$$a_k = g(s_{k+d_0}, \cdots, s_{k+d_{t-1}}), \quad k = 0, 1, \cdots .$$

This is referred to as a filtering generator where g is a called a filtering function, or simply filter, and $\underline{a} = \{a_k\}$, a filtering sequence.

Example 1. For $q = 2$, $n = 5$, $t = 3$, let $\{s_k\}$ be a binary sequence generated by the LFSR with the feedback $y^5 + y^3 + 1$, a selection of tap positions $(d_0, d_1, d_2) = (0, 2, 3)$, and filtering function $g(x_0, x_1, x_2) = x_0 + x_1x_2$. Then the output sequence, i.e., a filtering sequence, is given by

$$a_k = s_k + s_{k+2}s_{k+3}, \quad k = 0, 1, \cdots$$

2.2 GAP, Synthesis and Related Work

GAP (Groups, Algorithms, Programming) is a specialized computer algebra system, originally intended for group theory, but evolved to include vector spaces, algebras, matrices, polynomials, etc [6]. The proposed toolkit consists of two packages written in the GAP language: the package FSR, which can be used as stand-alone package, and the FSRtoVHDL package, which requires the package FSR. GAP is included in SageMath[8], which allows both FSR and FSRtoVHDL to be loaded and used in SageMath as well. There is a SageMath package *Cryptography* [9], implementing *LFSRCryptosystem* over the finite field $\mathbb{F}_2$, but it does not support extension fields. A simple Mathematica package, called Symbolic Linear Feedback Shift Registers [10] can generate bit sequences from LFSRs. The FSR package presented in this work is capable of working with LFSRs, NLFSRs and filtering functions, defined over both prime and extension fields. To the best of authors knowledge, this toolkit is the unique in its ability to generate VHDL from a mathematical description containing finite field arithmetic. For integer, fixed point, and floating point arithmetic, MathWorks[7] offers an extensive range of embedded software and hardware support, from ARM microprocessors, Altera and Xilinx FPGAs to PARROT Minidrones. Similarly, tools like MATLAB Coder, Simulink Coder, Embedded Coder generate C and C++ code and HDL Coder can generate synthesizable Verilog and VHDL code. The FSRtoVHDL package is situated on top of Register-Transfer-Level, but lower than High-Level Synthesis, which transform a behavioural description into RTL designs [14]. The presented toolkit generates synthesizable VHDL modules suitable for implementation on FPGAs or ASICs. The ASIC implementation results for the examples shown in this paper were obtained for a 65nm CMOS ASIC technology using Synopsis Design Compile for synthesis. The FPGA results are obtained for Xilinx Spartan 3 devices using Xilinx-ISE. All results were obtained post place-and-route.

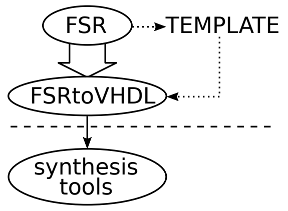
3 Toolkit for Generating Hardware Modules

This section begins with an overview of the toolkit and hardware-design workflow (Section 3.1), then describes the FSR package (Section 3.2), how to build top-level modules (Section 3.3), and finally the FSRtoVHDL module (Section 3.4).

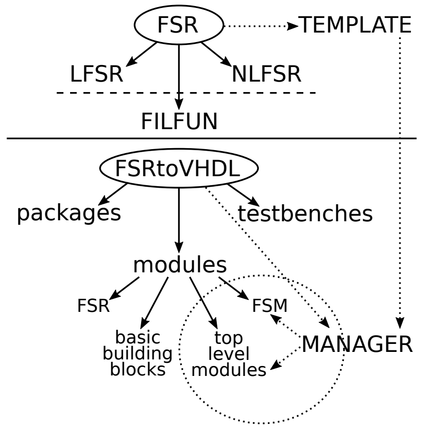
3.1 Overview and Workflow

The toolkit consists of two GAP packages: FSR and FSRtoVHDL. The FSR package is used to create various FSRs and filters, which are then translated to

VHDL modules using the FSRtoVHDL package, i.e. the FSRtoVHDL package takes care of the design entry. The produced VHDL code is used as input to synthesis tools to evaluate the design for different metrics. This design flow is captured in Figure 2(a). The dashed horizontal line marks the end of design entry.



(a) The design-flow abstraction



(b) The FSR and FSRtoVHDL packages

Fig. 2: Toolkit overview

The toolkit significantly reduces the amount of human effort for both software implementation of a cipher using GAP and the design entry in VHDL. The design of the toolkit was guided by the following principles:

1. The toolkit lies in the intersection of finite field arithmetic and hardware design (with VHDL as the preferred choice for design entry).
2. Hardware-style thinking involves a modular approach: a cipher can be implemented as a collection of basic modules, which were identified as LFSRs, NLFSRs and FILFUN filters.
   (a) For most (N)LFSRs used in practice, the feedback function can be modelled by a multivariate polynomial (Example 3, Case studies 1 and 2).
   (b) Arbitrarily complicated NLFSRs can be implemented by connecting an (N)LFSR and FILFUN(s) (Example 4).
   (c) A complex FILFUN filter can be implemented by connecting multiple simple FILFUNs (Case studies 1 and 2).
3. Recognizing and exploiting structural similarities between LFSRs, NLFSRs and FILFUNs, from both mathematical and hardware perspective, reduces the number of implemented objects, functions and methods.
4. A cipher as a collection of basic modules must be represented in a well structured manner with sufficient information for generation of the top-level VHDL module (Case studies 1 and 2).
5. Highly structural FSR package design is mandatory for VHDL generation.
6. For small designs (finite fields up to $\mathbb{F}_{2^{16}}$), optimizations can be handled by hardware synthesis tools without resorting to specialized rewriting or algebraic techniques.

The structural similarities from mathematical point of view (item 3) are explained in Table 1 in section 3.2. Items 4 and 5 are related to the transition from GAP objects to VHDL code, which was one of the biggest challenges in developing the toolkit. The problem in item 4 was tackled with a *template* used to connect the FSR modules in hardware and additional GAP objects, implemented as a part of the FSRtoVHDL package, e.g. an *inout* field in the template to capture top-level input and output ports. The *template* is the only input that the user must provide: it must specify indeterminates, all parameters needed by FSR constructors and the actual FSR objects must be included. Item 5 is partially addressed by the good design of FSR objects: all the information for VHDL implementation of a particular FSR is included in the GAP object in form of GAP attributes, e.g. the underlying finite field defines which VHDL data type to use for the signals.

Detailed structure of the toolkit is shown in Figure 2(b). The objects LFSR, NLFSR and FILFUN are implemented in the FSR package. The FSRtoVHDL package contains VHDL generation functions, which generate hardware modules complying with typical hardware coding conventions. The *template* is used to provide information to the toolkit's function named *manager*. *Manager* generates a simple spreadsheet which is used by the designer to specify connections between source and target ports within the module. Based on this information, the *manager* invokes a sequence of appropriate VHDL generation functions. As mentioned in item 6, the toolkit supports designs over small finite fields, because GAP uses the Zech's logarithm representation of elements for finite fields up to $\mathbb{F}_{2^{16}}$ and switches to polynomial representation for larger fields.

3.2 The GAP Package FSR: Feedback Shift Registers

The GAP package FSR implements feedback shift registers (Section 2.1). It allows creation, initialization and running of both LFSRs and NLFSRs, and can compute some of their properties, e.g. the internal state size of any FSR or the period of an LFSR. The FSR package also implements filtering functions, named FILFUNs. A FILFUN is an object of type FSR without feedback, shifting, or storing, whose functionality is defined by a multivariate polynomial. The justification for such a design decision is twofold: (i.) filtering functions are similar to NLFSR feedback functions and (ii.) FSRs with output filter are common, indicating that they will be used together. The structural similarities between the three FSR objects (item 3 in section 3.1) are captured in Table 1. The differences between the FSRs and FILFUNs will be discussed shortly.

The basic components of all three objects are *state*, holding the current state $(S_{n-1},\ldots, S_0) \in \mathcal{F}^n$, and *basis* used for representation of the field elements. Constructors for the FSR objects are listed in Table 2, and their main functionality in Table 3. Table 3 differentiates between a regular and an external step and run. A stand-alone simple (N)LFSR object is self-contained: it is updated by the computed feedback value (regular step and run). Examples 2 and 3 show the regular run. The external *StepFSR* allows arbitrary filters to be added to the

Table 1: Structural similarities between LFSR, NLFSR and FILFUN objects

FSR object name	multivariate polynomial $f(x)$		FSR (feedback, memory)	output	
	linear	nonlinear		one or more state elm.	computed value of $f(x)$
LFSR	✓		✓	✓	
NLFSR		✓	✓	✓	
FILFUN	✓	✓	✓		✓

Table 2: Constructors for FSR objects

FSR constructor			FSRtoVHDL comments
name	mandatory	optional	
LFSR	$\mathbb{F}_q$, $h(y)$ from eq. (6)	basis, $(d_0, \ldots d_{t-1})$	$q = 2$ or $q = 2^m$
NLFSR	$\mathbb{F}_q$, $f(x_0, \ldots, x_{n-1})$ from eq. (1), n	basis, $(d_0, \ldots d_{t-1})$	$q = 2$ or $q = 2^m$
FILFUN	$\mathbb{F}_q$, $f(x_0, \ldots, x_{t-1})$ from eq. (1)	basis	

$h(y) : F_q \to \mathbb{F}_q$ $(d_0, \ldots d_{t-1})$ - output taps

$f(x_0, \ldots, x_{j-1}) : \mathbb{F}_q^j \to \mathbb{F}_q$, where $j = t, n$ n - length (number of stages)

Table 3: Main menthods for FSR objects

All FSR types: LFSR, NLFSR, FILFUN		
method name	options	comments
LoadFSR	NA	load the initial state
StepFSR	regular step	FSR self-contained: compute value x
	external step	adds an external elem. to the computed value: $x + \text{ext}$
	compute the feedback/function value x, use x or $x + \text{ext}$ to: ◦ update S_{n-1} after shifting stages S_{n-1},…, S_1 for (N)LFSR ◦ output as the new element in case of FILFUN	
LoadStepFSR	regular step	FSR self-contained: compute value x
	external step	adds an external elem. to the computed value: $x + \text{ext}$
	combines methods *LoadFSR*, *StepFSR* to load the new values for variables before evaluating the function this method is used by *RunFSR* for FILFUN objects	
RunFSR	regular run	with regular step
	external run	with external step
	optional *LoadFSR* followed by sequence of *StepFSR* calls	

feedback of any (N)LFSR or it can be used e.g. to mask the output of the filtering function. The external step and run are allow the FSRs to be used directly as building blocks of many ciphers.

While the LFSR and NLFSR differ only in feedback, the filters are a bit of an exception, which is indicated by the dashed line in Figure 2(b). A filter alone does not require any feedback, shifting or stages, i.e. hardware registers:

the component *state* is used to hold the current values needed to evaluate the filtering function. The field *state* is not updated, but rather loaded anew with each step: method *RunFSR* takes a list of “initial” states as input as shown in Example 3, then calls *LoadStepFSR* for each list entry.

The FSR package also includes output formatting functions for testbench generation and drawing functions that can automatically generate tikz code. More detail can be found at https://github.com/nzidaric/fsr and its manual.

Example 2. The following example shows a regular run of an LFSR over $\mathbb{F}_{2^2}$, using the function call *RunFSR* with initial state `ist` and number of steps performed given as an argument. Stage S_0 is used to output the sequence elements.

Example 2
```
gap> K := GF(2);; x := X(K,"x");; f := x^2+x+Z(2)^0;;
gap> F := FieldExtension(K, f);; y := X(F, "y");;
gap> l := y^3+y^2+y+Z(2^2);;
gap> lfsr := LFSR(F, l);
< empty LFSR over GF(2^2)  given by FeedbackPoly = y^3+y^2+y+Z(2^2) >
gap> ist := [Z(2^2), Z(2)^0, 0*Z(2)];;
gap> RunFSR(lfsr, ist, 5);
[ 0*Z(2), Z(2)^0, Z(2^2), Z(2^2)^2, Z(2^2)^2, Z(2^2)^2 ]
```

Example 3. The following example shows a regular run of an FILFUN over $\mathbb{F}_2$, using the function call *RunFSR* but now with a sequence of inputs `inputsequence`.

Example 3
```
gap> f := x_0*x_1+x_2;; fil := FILFUN(K,f);
< FILFUN of length 3 over GF(2),
with the MultivarPoly = x_0*x_1+x_2>
gap> inputsequence := [[Z(2)^0, Z(2)^0, 0*Z(2)],[Z(2)^0, Z(2)^0,
Z(2)^0],[0*Z(2), Z(2)^0, 0*Z(2)]];;
gap> RunFSR(fil, inputsequence);
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
```

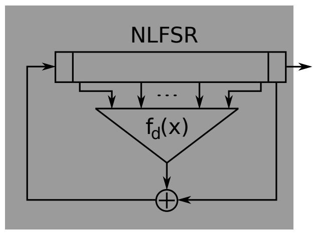
3.3 The Top-Level Modules in GAP and VHDL

The toolkit supports numerous configurations of top-level modules, ranging from simple sequence generators using a single (N)LFSR to complex designs with multiple (N)LFSRs and filtering functions. The GAP package FSR was designed in such a way that each FSR object corresponds to a VHDL hardware module. The toolkit directly supports any FSR that can be modelled as shown in Table 2. A more detailed classification can be seen in Table 4. FSRs with less conventional feedback functions can be modelled using the FSR package by splitting the feedback function into a filter and a feedback which allows direct use of (N)LFSR objects and then connecting the filter using the external connection to the FSR.

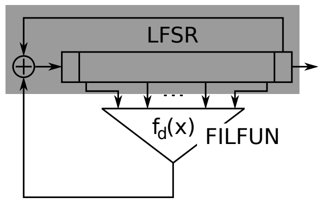
The recommended strategy for capturing top-level modules is a combination of a top-down and bottom-up approach. Before the implementation, whether in software or in hardware, the mathematical design can be transformed for the easiest representation as a collection of different FSRs. Usually, the implementation using the FSR package is straightforward: FSR objects from Table 2, for which the conditions $h(y) : \mathbb{F}_q \to \mathbb{F}_q$ hold for the LFSRs or conditions $f(x_0, \ldots, x_{j-1}) : \mathbb{F}_q^j \to \mathbb{F}_q$, where $j = t, n$, for the NLFSRs and FILFUNs, are very common and can be implemented directly. Example 4 shows a top-level design for which this process is more complicated. The FSR packages can also be used to create e.g. a 4-bit shift register (no feedback) in VHDL by creating an LFSR with a feedback x^4. The FILFUN objects can be used for arbitrary Boolean functions. The designs steps for implementation of the top-level module:

1. Represent the cryptographic module as a collection of different FSRs.
2. Identify possible modes of operation. For each mode and all FSRs define:
   - Number of steps performed
   - FSR input and possible external input
3. Capture desired behaviour in GAP using methods *LoadFSR*, *StepFSR* and *LoadStepFSR* or using the *manager* function call.

Step 2 is very important: it ensures a successful transition to VHDL and a clean finite state machine (FSM), parametrized with appropriate number of steps (clock cycles) and issuing correct control signals for the use of the external FSR inputs. Table 5 shows the modes of operation for WG-7 cipher. All the steps performed in each mode should be exactly the same; discrepancies mean that a mode exists but was not captured and the design step 3 must be repeated.



(a) original n-stage NLFSR



(b) n-stage LFSR with a filter

Fig. 3: Schematic of an NLFSR represented as an LFSR with a filter FILFUN

Example 4. Figure 3(a) represents the original schematic of an NLFSR from [15], generating a span-n sequence: the shift register itself is defined over $\mathbb{F}_2$, the t stages provide the coefficients for an element $x \in \mathbb{F}_{2^t}$, which is the input to the function $f_d : \mathbb{F}_{2^t} \to \mathbb{F}_2$. The FSR package (N)LFSR objects can have up to n output taps: in case of $t + 1$ output tap positions, the method *StepFSR* returns the contents of $t + 1$ stages, specified[1] by $(d_0, \ldots, d_t)$. Thus, the NLFSR from Figure 3(a) can be implemented as an LFSR (shaded grey in Figure 3(b)) with a FILFUN implementing function f_d. The LFSR is defined over $\mathbb{F}_2$ with feedback $x^n + 1$ and $k + 1$ tap positions. It uses an external signal from the FILFUN for external step and run.

[1] output taps in Table 2

3.4 The FSRtoVHDL Package

The FSRtoVHDL package takes an FSR object, or a collection of FSRs objects and a template specifying their relationship, as an input and generates the VHDL code for its hardware implementation. The capabilities of the FSRtoVHDL package are captured in Figure 2(b), however, the functions for generating FSMs and top-level modules are not fully implemented at this time.

Two VHDL packages are used to define the finite fields used in the design (`field_pkg.vhd`) and signals used by the FSRs (`fsr_pkg.vhd`): they contain all of the fields and FSRs used in the design. The finite fields and the FSRs are enumerated, which increases the readability of the generated code. The parameters defined in the `field_pkg.vhd` are used for basic building blocks, such as multipliers. The basic building blocks are implemented using naive methods[2] and for polynomial bases only. The VHDL modules that only need to be parametrized by the degree of $\mathcal{F}\,/\,\mathcal{K}$ are predefined, however, the blocks depending the field defining polynomial f are generated on the fly once f is known. Examples of the latter are reduction matrices and matrix-vector multipliers for multiplication by constants. FSRtoVHDL includes methods that generate corresponding matrices prior to generating their VHDL modules.

The FSRs are classified into 6 cases based of their type, underlying finite field, number of variables t and kind of nonzero coefficients. Conditions, cases implementation status and implementation comments are listed in Table 4. Each case has its own rules for generating the FSR architecture. A special case exists for the FILFUN filters: they can have one single input, i.e., use a univariate filtering function. For the small fields it is feasible to attempt a look-up table style implementation, called `const_array`[3]. The filtering function is evaluated for all field elements and stored as a constant VHDL array. The value the variable takes behaves as an address to this array.

Table 4: FSRtoVHDL classification for FSR modules

conditions				FSRtoVHDL		
FSR type	# vars t	underlying finite field	$\forall i : c_i$ belongs to	case	implementation status	comments
LFSR	NA	$\mathbb{F}_2$	$\mathbb{F}_2$	1	fully	
		$\mathbb{F}_{2^m}$	$\mathbb{F}_2$			
			$\mathbb{F}_{2^m}$	2	fully	MV† for constants
NLFSR or FILFUN	$t > 0$	$\mathbb{F}_2$	$\mathbb{F}_2$	3	fully	
	$t > 1$	$\mathbb{F}_{2^m}$	$\mathbb{F}_2$	4	partially	
			$\mathbb{F}_{2^m}$	5	partially	MV† for constants
	$t = 1$	$\mathbb{F}_{2^m}$	$\mathbb{F}_2$	6	fully	const_array architecture
			$\mathbb{F}_{2^m}$			

NA - not applicable † - matrix vector multiplier

[2] which have a good performance for small fields
[3] to differentiate it from the FPGA LUTs

4 Case Studies and ASIC Implementation Results

4.1 Case Study 1: The datapath for WG Keystream Generators

WG-7 and WG-8 are members of the Welch-Gong (WG) family of bit-oriented stream ciphers. WG ciphers generate a keystream with proven randomness and cryptographic properties. They are composed of an LFSR over an extension field and a filter function. The LFSR outputs an m-sequence, which is then filtered with the WG transformation over the same extension field.

Let m be an integer that is not a multiple of 3. The decimated[4] WG transformation from $\mathbb{F}_{2^m}$ to $\mathbb{F}_2$ consists of a WG permutation (eq. (7)) and WG transformation (eq. (8)) of $X \in \mathbb{F}_{2^m}$:

$$\mathsf{WGP}\text{-}m(X^d) = q(X^d + 1) + 1 \tag{7}$$

$$\mathsf{WGT}\text{-}m(X^d) = \mathrm{Tr}(\mathsf{WGP}(X^d)) \tag{8}$$

The polynomial $q(x) = x + x^{r_1} + x^{r_2} + x^{r_3} + x^{r_4}$ is a permutation polynomial from $\mathbb{F}_{2^m}$ to $\mathbb{F}_{2^m}$. Further details are omitted for brevity. `Example 5` shows the WG-7 GAP code using equation (7).

Example 5: beginning of the WG-7 template

```
#################    WG7 params   #################
f := x^7+x+Z(2)^0;;  F := FieldExtension(K, f);; ChooseField(F);
l := y^23+y^12+y^11+y^8+y^7+y^6+y^5+y^4+y^3+y^2+y+Z(2^7);;
exponents := [ 1, 33, 41, -23, 39 ];; d := 63;; # trace = x_0
#################    WG7  FSRs    #################
lfsr := LFSR(F, l, [Degree(l)-1]);
dwgpfun := One(F);
for j in [1..Length(exponents)] do
  r := exponents[j];
  if r<0 then r := r mod (Size(F)-1); fi;
  dwgpfun := dwgpfun + (x_0^d + One(F))^r;
od;
dwgpfil := FILFUN(F, dwgpfun);
```

Table 5 shows a filled out spreadsheet provided by the *manager*: the entries $\times$, [0], *load, init, run* and *always* are filled in by the designer. Values *load, init, run* are possible modes of operation (used by FSM), and value *always* indicates an unconditional connection. The entry "[0] run" corresponds to the trace of an element of $\mathbb{F}_{2^7}$ represented in polynomial basis using field defining polynomial $x^7 + x + 1$, that is the WG transformation (equation (8))[5].

ASIC implementation results for the WG-7 and WG-8 datapaths, created by FSRtoVHDL package, are listed in Table 6, and the FPGA implementation results for WG-8 in Table 7. The FSRtoVHDL generated WG-7dp exhibits a slightly smaller area and comparable clock frequency, expect for the highest frequency circuit. All three WG-8 FSRtoVHDL circuits reached a significantly

[4] decimation exponent $d > 1$ and $\gcd(d, 2^m - 1) = 1$

[5] $\mathbb{F}_{2^8}$ with defining polynomial $x^8 + x^4 + x^3 + x^2 + 1$: trace is bit 5, i.e. "[5] run"

higher frequency, with a 2.5× speedup for the best optimality; they are expected to outperform the WG-8 [17] implementation after the FSM is added. The FSRtoVHDL FPGA implementation, however, reached only 65% of the frequency reached by the manual WG-8 [17] implementation.

Table 5: WG-7 spreadsheet example

	i_data_1	fsr_1_o_fsr	fsr_2_o_fsr
o_data_1	×	×	[0] run
fsr_1_i_fsr	load	×	×
fsr_1_i_ext	×	×	init
fsr_2_i_fsr	×	always	×
fsr_2_i_ext	×	×	×

relationship to GAP code in `Example 5` above:

FSR object	VHDL module
`lfsr`	fsr_1
`dwgpfil`	fsr_2

Table 6: ASIC implementation results for WG-7 and WG-8 datapaths

design used	**Speed [GHz]**	**Area [GE]**	**Speed [GHz]**	**Area [GE]**	synthesis tools optimization goal
	FSRtoVHDL		**Manual design**		
WG-7dp	1.00	1320	0.91	1300	smallest area
WG-7dp	1.43	1430	1.43	1740	best optimality
WG-7dp	1.67	2260	2.00	2330	highest frequency
	FSRtoVHDL		**WG-8 † [17]**		
WG-8dp	0.83	1640			smallest area
WG-8dp	1.25	1860	0.5	1786	best optimality‡
WG-8dp	1.67	2950			highest frequency

† including FSM ‡ unknown synth. tools goal for †

Table 7: FPGA implementation results for WG-8 datapath

design used	**Speed [MHz]**	**Area [# slices]**	**Speed [MHz]**	**Area [# slices]**	FPGA device used
	FSRtoVHDL		**WG-8 † [17]**		
WG-8dp	124	74	190	137	`xc3s1000-5fg320`

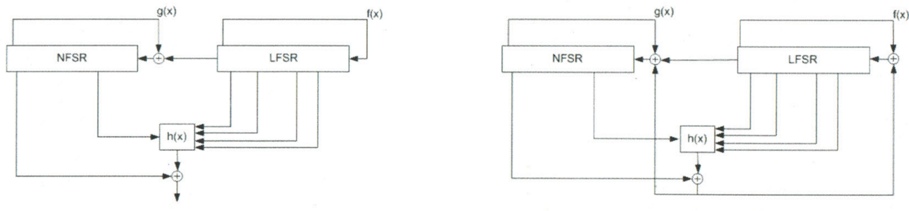
† including FSM

4.2 Case Study 2: Grain v1

Grain v1 [18,19] is one of the three Profile 2 ciphers[6] included in the eSTREAM portfolio. The structure of Grain v1 is shown Figure 4: it includes an 80-bit

[6] stream ciphers for hardware applications with highly restricted resources

LFSR (with $f(x)$), an 80-bit NLFSR (with $g(x)$) and a filtering function $h(x)$, which takes the input bits from both FSRs. Result of this function is masked by a bit from the NLFSR to produce the keystream bit.



(a) Run mode (b) Initialization mode

Fig. 4: Original schematic of Grain cipher [18]

The two modes of operation from Figure 4 are combined into a single schematic in Figure 5: the two original shift registers are presented as FSR package blocks with output taps and utilizing external step. The filled out spreadsheet to define the datapath for Grain is shown in Table 8; this table provides the information that the *manager* needs to execute Grain and generate the VHDL for Grain datapath (Figure 5). In the future, the toolkit will also be able to generate the control circuitry and top-level module for ciphers.

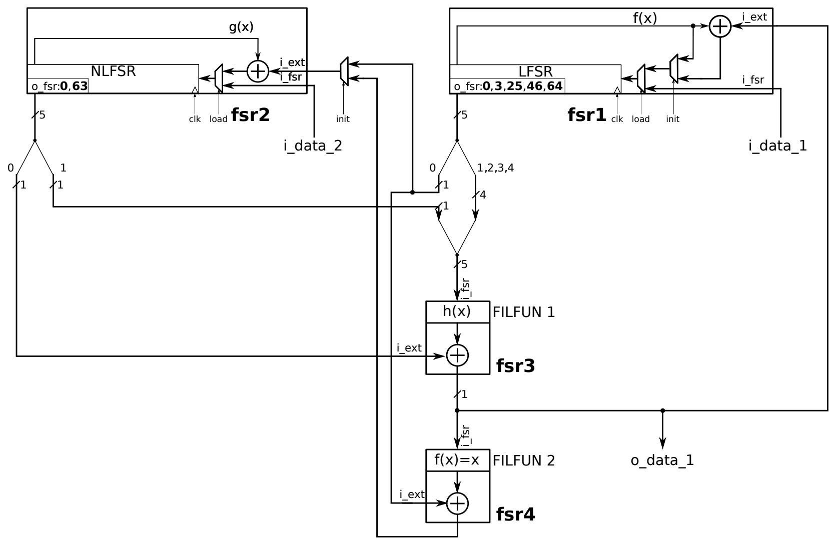


Fig. 5: Unified schematic of Grain v1 using FSR

The NLFSR (`fsr_2` in Figure 5 and Table 8) is using the external input during both initialization and running mode, hence an extra multiplexer is needed (row `fsr_2_i_ext` in Table 8). The filtering function $h(x)$, now represented as FILFUN1, takes bits 1,2,3,4 from `fsr_1` and bit 1 from `fsr_2`: this is encoded with two rules[7] in row `fsr_3_i_fsr` in Table 8. The masking bit from the NLFSR is used as external input for the filtering function $h(x)$, now represented as FILFUN1 (`fsr_3` in Table 8). The extra `XOR` gate, used for the NLFSR external input during initialization is represented as FILFUN2 identity function and an external input. While it may seem excessive to represent a simple `XOR` with a FILFUN, this aids the transition to VHDL.

Table 8: Grain spreadsheet example

	i_data_1	i_data_2	fsr_1_o_fsr	fsr_2_o_fsr	fsr_3_o_fsr	fsr_4_o_fsr
o_data_1	$\times$	$\times$	$\times$	$\times$	$\times$	run
fsr_1_i_fsr	load	$\times$	$\times$	$\times$	$\times$	$\times$
fsr_1_i_ext	$\times$	$\times$	$\times$	$\times$	init	$\times$
fsr_2_i_fsr	$\times$	load	$\times$	$\times$	$\times$	$\times$
fsr_2_i_ext	$\times$	$\times$	[0] run	$\times$	$\times$	init
fsr_3_i_fsr	$\times$	$\times$	$[1, 2, 3, 4, -1]$	$[-1, -1, -1, -1, 1]$	$\times$	$\times$
fsr_3_i_ext	$\times$	$\times$	$\times$	[0]	$\times$	$\times$
fsr_4_i_fsr	$\times$	$\times$	$\times$	$\times$	always	$\times$
fsr_4_i_ext	$\times$	$\times$	[0]	$\times$	$\times$	$\times$

Tables 9 and 10 show the ASIC and FPGA implementation results for the Grain datapath. The ASIC implementations are compared to a manual design of Grain datapath and of Grain cipher used in [17], which is to the best of authors knowledge the only post place-and-route CMOS65nm implementation of Grain. The same circuit is presented in both the best optimality and highest frequency column for the manual design of grain datapath. The FSRtoVHDL datapath implementation results are very satisfactory in comparison with the smallest area and best optimality manual circuits. The [17] reference implementation includes the FSM, which makes a direct comparison difficult. However, the smallest area FSRtoVHDL datapath is expected to reach the hardware cost of Grain from [17] after the FSM is added.

The FPGA Grain_dp results are compared with Grain cipher implementation results from [20], which includes the FSM. The generated datapath area is approximately 65% of the area for the full cipher. The speedup reached by the FSRtoVHDL datapath design is probably not representative: a drop in frequency is common after an FSM is added.

Overall, the FSRtoVHDL generated hardware is comparable to the manual designs and gives a good starting point for further manual optimizations.

[7] meaning of -1: this bit is defined in the other rule

Table 9: ASIC implementation results for Grain datapath

design used	**Speed [GHz]**	**Area [GE]**	**Speed [GHz]**	**Area [GE]**	synthesis tools optimization goal
	FSRtoVHDL		**Manual design**		
Grain_dp	1.11	977	1.00	1020	smallest area
Grain_dp	1.67	1080	1.67	1110	best optimality
Grain_dp	2.00	1610	1.67	1110	highest frequency
			Grain† [17]		
Grain	-	-	1.02	1126	unknown

† including FSM

Table 10: FPGA implementation results for Grain datapath

design used	**Speed [MHz]**	**Area [# slices]**	**Speed [MHz]**	**Area [# slices]**	FPGA device used
	FSRtoVHDL		**Grain† [20]**		
Grain_dp	228	28	196	44	`xc3s50-5pq208`

† including FSM

5 Conclusion

This work presents an automation toolkit for the rapid hardware design of cryptographic modules with filtering structures, composed of feedback shift registers and filtering functions. The toolkit consists of two packages FSR and FSRtoVHDL, written in the GAP language. A great advantage of the FSR package is the generality of the FSR objects that can be modelled, and as such, they can be used directly as building blocks of many cryptographic modules. The FSR package is the core of the toolkit for generation of hardware modules. Because the FSRs can be executed, the toolkit is also able to generate of test vectors for the (hardware) simulations.

The toolkit is based on exploitation of structural similarities between LFSRs, NLFSRs and filters, from both a mathematical and a hardware perspective. For each FSR object a corresponding hardware module can be generated, and the FSR objects themselves contain all the information needed for their execution and hardware implementation. The FSR package can be used to implement arbitrary primitives in GAP, which can operate e.g. as random number generators.

The optimization of the generated hardware is left to the synthesis tools. The results of the synthesis tools, e.g. critical path analysis, can be used to further optimize the generated hardware, or even change a part of the design entirely. Two case studies were used to show that the toolkit generated datapaths comparable with manual designs. Overall it provides a good estimate of the hardware cost for a cryptographic primitive and gives a good starting point for further manual hardware optimizations.

References

1. Robshaw, M.:"The eSTREAM Project", New Stream Cipher Designs - The eSTREAM Finalists, Springer-Verlag, Berlin Heidelberg, 2008
2. Wu, H.: "ACORN: A Lightweight Authenticated Cipher (v1)",
3. https://competitions.cr.yp.to/caesar.html
4. ETSI/SAGE Specification version 1.1: "Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification", Sept. 2006
5. ETSI/SAGE Specification version 1.6: "Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification", June 2011
6. The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.8.8*; 2017, (`https://www.gap-system.org`).
7. https://www.mathworks.com/
8. http://www.sagemath.org/
9. http://doc.sagemath.org/html/en/reference/cryptography/index.html
10. http://library.wolfram.com/infocenter/MathSource/5717/
11. Lidl, R., Niederreiter, H.: *Finite fields*, Encyclopedia of Mathematics and its Applications, Vol.20, Cambridge University Press, 1997
12. Golomb, S.W., Gong, G.: *Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*, Cambridge University Press, 2005
13. Chen, L., Gong, G.: *Communication System Security* , CRC Press, 2012
14. Coussy,P., Gajski, D.D., Meredith,M., Takach, A.: "An Introduction to High-Level Synthesis", IEEE Design & Test of Computers, Volume: 26, Issue: 4, July-Aug. 2009, pp. 8 - 17, August 2009
15. Mandal, K., Gong, G.: (2014) Generating Good Span n Sequences Using Orthogonal Functions in Nonlinear Feedback Shift Registers. In: KoĂ§ Ă‡. (eds) Open Problems in Mathematics and Computational Science. Springer, Cham
16. G. Gong, M. Aagaard, and X. Fan,"Resilience to distinguishing attacks on WG-7 cipher and their generalizations", Cryptogr. Commun., vol. 5, no.4, pp. 277-289, Dec. 2013.
17. G. Yang, X. Fan, M. Aagaard and G. Gong,"Design Space Exploration of the Lightweight Stream Cipher WG-8 for FPGAs and ASICs", The 8th Workshop on Embedded Systems Security (WESS'13), ACM Press, Article No. 8, September 29, 2013,
18. M. Hell, T. Johansson, W. Meier, "Grain - A Stream Cipher for Constrained Environments",
19. M. Hell, T. Johansson, A. Maximov, and W. Meier, "The Grain Family of Stream Ciphers",, New Stream Cipher Designs - the eSTREAM finalists, Springer-Verlag, Berlin Heidelberg, 2008
20. D. Hwang, M. Chaney, S. Karanam, N. Ton, and K. Gaj. "Comparison of fpga-targeted hardware implementations of estream stream cipher candidates", The State of the Art of Stream Ciphers, pages 151-162, 2008.