Bitstream Modification Attack on SNOW 3G

Michail Moraitis Elena Dubrova
Department of Electronics, Royal Institute of Technology (KTH)
Electrum 229, 196 40 Stockholm, Sweden
{micmor,dubrova}@kth.se

***Abstract*—SNOW 3G is one of the core algorithms for confidentiality and integrity in several 3GPP wireless communication standards, including the new Next Generation (NG) 5G. It is believed to be resistant to classical cryptanalysis. In this paper, we show that SNOW 3G can be broken by a fault attack based on bitstream modification. By changing the content of some look-up tables in the bitstream, we reduce the non-linear state updating function of SNOW 3G to a linear one. As a result, it becomes possible to recover the key from the keystream. To our best knowledge, this is the first successful bitstream modification attack on SNOW 3G. We propose a countermeasure which blows-up the number of candidate points for fault injection, making the presented attack infeasible in practice. *Index Terms*—SNOW 3G, stream cipher, fault attack, FPGA, bitstream modification, reverse engineering.**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are used in many applications, including data centers, automotive, aerospace, defense, medical, wired and wireless communications. Many of these applications require cryptographic protection of data. This brings the need for evaluating physical security of FPGA implementations of cryptographic algorithms. It is particularly important to evaluate the algorithms recommended by standards. In this paper, we focus on SNOW 3G stream cipher which is the core of UEA2 and UIA2 algorithms in 3G UMTS standard, 128-EEA1 and 128-EIA1 algorithms in 4G LTE standard, and 128-NEA1 and 128-NIA1 algorithms in the new NG 5G standard. The popularity of SNOW 3G is to a large extent due to its known resistance to classical cryptanalysis [2]–[6].

One of the most popular type of physical attacks on FPGAs is *reverse engineering* of the bitstream. Reverse engineering enables copying designs which cost millions of dollars to develop [7].

Reverse engineering is difficult to stop. One of the countermeasures is *obfuscation* of the bitstream. For example, for Look-Up Table (LUT)-based FPGAs, the truth table of the Boolean function defining a k-input LUT is not stored as one block of 2^k consecutive bits in the bitstream. Rather, it is first permuted and then partitioned into several blocks which are located on given offsets from each other. Obfuscation algorithms are proprietary and kept secret. Unfortunately, history shows that secrecy is not sufficient for assuring algorithm's security.

Several reverse engineering tools have been created for older Xilinx FPGA families [8]–[12]. A full Verilog-to-bitstream flow has been developed for Lattice iCE40 [13]. Last year, information about the bitstream format of the latest Xilinx series 7 FPGA has been revealed [14], [15].

Bitstream *encryption* is yet another countermeasure which, in theory, should protect against reverse engineering. In reality, however, security of encryption (or any other cryptographic algorithm) is not greater than security of its secret key storage/generation mechanism. It is known that the encryption key used for the Advanced Encryption Standard (AES)-256 based encryption of bitstreams in Altera and Xilinx FPGAs can be extracted by a side-channel attack [16]–[18].

A short version of this paper was presented DATE'2020 conference [1], 9-13 March 2020, Grenoble, France.

Cryptographically secure methods such as *Message Authentication Codes* (MACs) or *digital signatures* can be used to assure bitstream authenticity and integrity. However, if we take a closer look on how these methods are currently implemented in FPGAs, we can see that they are vulnerable to physical attacks. For example, in Xilinx series 7 FPGA, the MAC-then-encrypt approach is used. This means that a bitstream B is first authenticated by computing its 256-bit MAC (HMAC) with a key K_A. The bitstream with the MAC appended

is then encrypted with a key K_E. The encryption key K_E is stored on-chip. However, the authentication key K_A is stored in the bitstream, in two places (see Fig. 1). This method of storing K_A is believed to be secure due to of the follow-up encryption step. However, since the encryption key K_E can be extracted by a side-channel attack [16]–[18], the encrypted bitstream can be decrypted, revealing K_A.

To summarize, currently available methods do not seem to be sufficient to protect against reverse engineering of FPGA bitstreams. Apart from the IP theft, reverse engineering enables the attacker to do meaningful modifications of the bitstream. This attack vector has not been thoroughly explored yet.

Previous Work. In several works [19]–[21] it has been shown that direct bitstream manipulation is feasible in practice. Swierczynski et al. pioneered attacks in which all LUTs implementing the AES S-box in the bitstream are modified to weaken the AES algorithm [22], [23]. Our attack is based on the same idea except that, in our case, LUTs implementing a specific XOR gate rather than S-boxes are modified. So, our fault injection point is of finer granularity. Cryptographic designs typically contain many XORs, thus finding a specific one is challenging.

We are not aware of any previous bitstream modification attack on SNOW 3G. Three other types of fault attacks have been presented. In [24], an attack using 22 transient fault injections to recover the key of SNOW 3G is described. In [25] it is stated that stream ciphers, including SNOW 3G, may be vulnerable to faults caused by intentional electromagnetic interference. In [26], a cache timing attack capable of recovering the internal state of SNOW 3G from empirical timing data is described.

Our Contributions. The main contributions of this paper are:

- We present a fault attack on an FPGA implementation of SNOW 3G in which a stuck-at-0 fault is injected into a specific XOR gate by changing the content of LUTs implementing the gate in the bitstream. As a result,
  SNOW 3G algorithm becomes weaker and its key can be recovered from the keystream. To our best knowledge, this is the first successful bitstream modification attack on SNOW 3G.
- We show that by exploring the bitstream in a *key-independent* setting, we can reduce the complexity of some search tasks from exponential to linear. This idea has not been exploited in previous bitstream modification attacks.
- We created a tool which automatically finds a k-input LUT implementing a given k-variable Boolean function and all Boolean

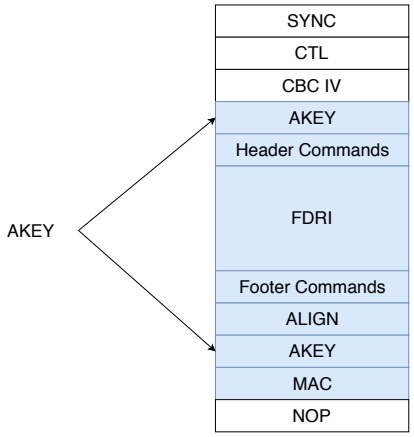

Fig. 1. Xilinx bitstream format (Virtex-6/7) [28]. Area in blue is authenticated and encrypted

functions within the same P equivalence class[1] in the bitstream. The tool is intended to assist in evaluating resistance of FPGAs to reverse engineering and bitstream modification.

- We propose a countermeasure against the presented attack which increases the number of candidate regions for fault injection exponentially, making the presented attack infeasible in practice.

Paper Outline. The paper is organized as follows. Section II gives a background on FPGA technology mapping. Section III reviews the SNOW 3G design. Section IV describes a general strategy for attacking FPGA implementations of encryption algorithms by bitstream modification. Section V presents features of the attack specific for SNOW 3G. Section VI proposes countermeasures. Section VII concludes the paper.

II. Background

In this section, we give a brief introduction to FPGA technology mapping, see [29] for more details.

A. Notation

Let $N = (V, \mathcal{E})$ denote a Boolean network, where V represents a set of gates and primary inputs and $\mathcal{E} \subseteq V \times V$ describes the nets connecting the gates. $Fanin(v) \subset V$ and $Fanout(v) \subset V$ sets of a node $v \in V$ are defined as $Fanin(v) = \{u \mid (u, v) \in \mathcal{E}\}$ and $Fanout(v) = \{u \mid (v, u) \in \mathcal{E}\}$, respectively. $PI \subset V$ and $PO \subset V$ denote the primary inputs and outputs of N, respectively.

The set of all nodes in the transitive fanin/fanout of v are denoted by $TrFanin(v)$ and $TrFanout(v)$, respectively.

B. FPGA technology mapping

An FPGA consists of an array of programmable logic blocks, programmable interconnect, and input/output pads. Many of the commercial FPGAs use LUT based logic blocks (Xilinx, Intel). A k-input LUT, k-LUT, can be programmed to implement any Boolean function of up to k variables.

The technology mapping problem in the case of k-LUT-based FPGAs consists of finding a functionally equivalent k-LUT network for a general Boolean network $N = (V, \mathcal{E})$ [29].

Algorithms for FPGA technology mapping use different strategies for finding the best k-LUT network for N and different objective function, such as minimal area [30] or depth [31], or both [32] easy routability [33], or power minimization [34].

A typical FPGA technology mapper traverses nodes $v \in V$ in backwards topological order from POs to PIs, and compute LUTs rooted in v by finding k-feasible cuts for v [32]. A set of nodes $C \subset V$ is a *cut* of a node v if any path from a PI to v passes through at least one node in C. Node v itself is a trivial cut. A cut C is called k-feasible if $|C| \leq k$. Each k-feasible cut C of v corresponds to a k-LUT which covers nodes in $TrFanin(v) \cap (\bigcup_{\forall c \in C} TrFanout(c))$ and has nodes of C as inputs and v as output. Cuts can be computed using the maximum flow algorithm, or cut enumeration technique [35]. Typically FPGA technology mappers reuse vertices which are already mapped while searching for k-feasible cuts.

[1]Two Boolean functions belong to the same P equivalence class, if f can be transformed into g through permutation of inputs [27].

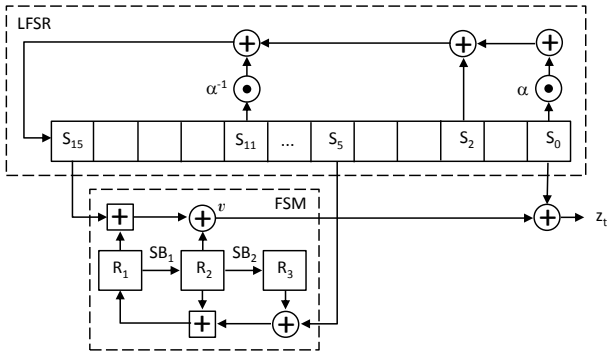

Fig. 2. Block diagram of SNOW 3G during keystream generation

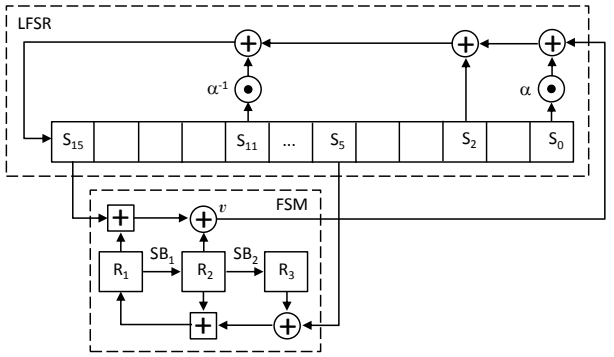

Fig. 3. Block diagram of SNOW 3G during initialization

III. SNOW 3G Design description

SNOW 3G belongs to the class of *binary additive stream ciphers*. A binary additive stream cipher generates a stream of pseudo-random symbols, called the *keystream*, Z, based on a secret key K and an initialization vector, IV. To encrypt, the keystream is combined with the plaintext, typically by the bitwise XOR. To decrypt, the ciphertext is XORed with the keystream. SNOW 3G is a *word-oriented* stream cipher. It generates a 32-bit word of keystream at each clock cycle.

The main blocks of SNOW 3G design are a Linear Feedback Shift Register (LFSR) and a Finite State Machine (FSM) (see Fig. 2). The gates denoted by "$\oplus$" and "$\boxplus$" stand for the bitwise XOR operation and the integer addition modulo 2^{32}, respectively. The gates denoted by "$\alpha\odot$"/"$\alpha^{-1}\odot$" perform a byte shift of the 32-bit input word to the left/right and then XOR the result with the output of the 8-bit into 32-bit mapping MUL_α/DIV_α whose definition can be found in [36].

The state of the LFSR is a vector consisting of the sixteen 32-bit stages $S = (s_0, s_1, \ldots, s_{15})$. The LFSR's feedback function is

defined by a primitive polynomial over $GF(2^{32})$. The FSM consists of the three 32-bit registers R_1, R_2 and R_3. The register R_1 is updated as $(s_5 \oplus R_3) \boxplus R_2$. The registers R_2 and R_3 are updated by the S-boxes SB_1 and SB_2, respectively, their definition can be found in [36].

The 32-bit words of the keystream, z_t, are produced by XORing the content of the stage s_0 with the FSM output word W computed as $W = (s_{15} \boxplus R_1) \oplus R_2$.

At the initialization stage, the LFSR is loaded with a combination of an 128-bit key K consisting of four 32-bit words k_0, k_1, k_2, k_3 and an 128-bit IV consisting of four 32-bit words iv_0, iv_1, iv, iv_3. The combination $\gamma(K, IV)$ is defined as follows:

$$
\begin{aligned}
s_{15} &= k_3 \oplus iv_0 & s_7 &= k_3 \\
s_{14} &= k_2 & s_6 &= k_2 \\
s_{13} &= k_1 & s_5 &= k_1 \\
s_{12} &= k_0 \oplus iv_1 & s_4 &= k_0 \\
s_{11} &= k_3 \oplus \mathbf{1} & s_3 &= k_3 \oplus \mathbf{1} \\
s_{10} &= k_2 \oplus \mathbf{1} \oplus iv & s_2 &= k_2 \oplus \mathbf{1} \\
s_9 &= k_1 \oplus \mathbf{1} \oplus iv_3 & s_1 &= k_1 \oplus \mathbf{1} \\
s_8 &= k_0 \oplus \mathbf{1} & s_0 &= k_0 \oplus \mathbf{1}
\end{aligned}
$$

where $\mathbf{1}$ is the all-1s word.

The registers R_1, R_2 and R_3 of the FSM are loaded with 0s. The output of the FSM is connected to the XOR gate as shown in Fig. 3. Then, the following two-step rounds are repeated 32 times:

1) The FSM is clocked, producing W.
2) The LFSR is clocked, consuming W.

No keystream is generated during the initialization.

IV. General strategy

In this section, we describe a general strategy for attacking an FPGA implementation of an binary additive stream cipher by bitstream modification. In principle, other types of cryptographic algorithms, e.g. block ciphers or hash functions, can be attacked similarly.

A. Attack Model

We use the attack model commonly used for bitstream modification attacks, namely we assume that:

1) The encryption algorithm under attack, E, is implemented in an SRAM-based FPGA.
2) The attacker has the bitstream $\mathcal{B}$ implementing E.
3) The attacker has a physical access to the FPGA.
4) $\mathcal{B}$ is neither encrypted, nor authenticated.

B. Main steps

The presented attack consists of the following steps:

1) Based on the cryptographic algorithm E under attack, decide on target gate v for fault injection and the type of fault α to be injected.
2) Guess which k-variable Boolean function f implements v in the netlist $N = (V, \mathcal{E})$ realizing E.
3) Find in the bitstream $\mathcal{B}$ the set of all candidates into k-LUTs implementing f.
4) Verify if the guess is correct.
5) Repeat the steps 2-4 until the guess is correct.

Algorithm 1 An algorithm for finding k-LUTs implementing a given Boolean function f in an FPGA bitstream.

Name: FINDLUT($\mathcal{B}, f, k, d, r$)
Input: Bitstream $\mathcal{B} = (b_0, \ldots, b_{|\mathcal{B}|-1}), b_i \in \{0,1\}$, Boolean function f of up to k variables, number of LUT's inputs k, offset d, number of partitions r
Output: Set $\mathcal{L}$ of candidates into k-LUTs implementing f in $\mathcal{B}$

```
L = ∅;
P_k = ComputePermutations(k);
P_r = ComputePermutations(r);
for each p ∈ P_k do
    F = GetTruthTable(l, p);
    B = ξ(F); /* permutes the truth table F */
    B = (B_1||B_2||...||B_r), |B_i| = |B_j|, ∀i, j ∈ {1, 2, ..., r};
    m = 2^k/r − 1;
    for each i from 0 to |B| − (r − 1)d do
        if i is not marked then
            for each (j_1, j_2, ..., j_r) ∈ P_r do
                if ((b_i, ..., b_{i+m}) = B_{j_1})&((b_{i+d}, ..., b_{i+d+m}) = B_{j_2})&...&((b_{i+(r−1)d}, ..., b_{i+(r−1)d+m}) = B_{j_r}) then
                    L = L ∪ {l};
                    Mark(i);
                end if
            end for
        end if
    end for
end for
return L
```

C. Finding LUTs in a bitstream

The pseudo-code of the algorithm for finding k-LUTs implementing a given Boolean function in the bitstream is shown as Algorithm 1. FINDLUT() takes as input the following parameters:

1) Bitstream under attack, $\mathcal{B}$.
2) Target Boolean function $f : \{0,1\}^k \to \{0,1\}$.
3) Number of LUT inputs k.
4) Offset d (depends on the FPGA).
5) Number of partitions r (depends on the FPGA).

It returns a set $\mathcal{L}$ of candidate LUTs implementing f in $\mathcal{B}$.

First, the truth table $F = (F[0], F[1], \ldots, F[2^k - 1]), F[i] \in \{0,1\}$, for $i \in \{0, 1, \ldots, 2^k - 1\}$ of the function f is derived. Then, F is mapped according to the obfuscation function which is specific for a given FPGA family. The obfuscation is typically performed in two steps. First, F is permuted using a bijective mapping $\xi : F \to B$ of type $\{0,1\}^k \to \{0,1\}^k$. The resulting vector $B = (B[0], B[1], \ldots, B[2^k - 1])$ is then partitioned into r sub-vectors $B_1, B_2, \ldots, B_r$ of equal size which are placed on a fixed offset d from each other in the bitstream, in varying order.

For Xilinx 7 Series FPGAs which we use in our experiments the obfuscation is implemented as follows. The truth table F is permuted according to the mapping ξ is defined in Table I [14]. The resulting vector B is partitioned into $r = 4$ sub-vectors B_1, B_2, B_3, B_4 of equal size

$$
\begin{aligned}
B_1 &= (B[0], \ldots, B[15), \\
B_2 &= (B[16], \ldots, B[31]), \\
B_3 &= (B[32], \ldots, B[47), \\
B_4 &= (B[48], \ldots, B[63]).
\end{aligned}
$$

which are placed on the fixed offset $d = 404$ from each other in the bitstream, in two different orders.

1) B_1, B_2, B_3, B_4 for functions implemented in SLICEL LUTs
2) B_4, B_3, B_1, B_2 for functions implemented in SLICEM LUTs.

To the best of our knowledge, this is the first time this information has been revealed.

If a LUT is found in the bitstream, its it added to the set $\mathcal{L}$ of previously computed candidate LUTs.

The set $\mathcal{L}$ returned by FINDLUT() may contain false positives. We verify if $L \subseteq \mathcal{L}$ is the set of LUTs implementing f in $\mathcal{B}$ as follows:

TABLE I
OBFUSCATION FUNCTION OF XILINX 7 SERIES FPGA [14]

a_6	a_5	a_4	a_3	a_2	a_1	F	$B = \xi(F)$
0	0	0	0	0	0	F[0]	B[63]
0	0	0	0	0	1	F[1]	B[47]
0	0	0	0	1	0	F[2]	B[62]
0	0	0	0	1	1	F[3]	B[46]
0	0	0	1	0	0	F[4]	B[61]
0	0	0	1	0	1	F[5]	B[45]
0	0	0	1	1	0	F[6]	B[60]
0	0	0	1	1	1	F[7]	B[44]
0	0	1	0	0	0	F[8]	B[15]
0	0	1	0	0	1	F[9]	B[31]
0	0	1	0	1	0	F[10]	B[14]
0	0	1	0	1	1	F[11]	B[30]
0	0	1	1	0	0	F[12]	B[13]
0	0	1	1	0	1	F[13]	B[29]
0	0	1	1	1	0	F[14]	B[12]
0	0	1	1	1	1	F[15]	B[28]
0	1	0	0	0	0	F[16]	B[59]
0	1	0	0	0	1	F[17]	B[43]
0	1	0	0	1	0	F[18]	B[58]
0	1	0	0	1	1	F[19]	B[42]
0	1	0	1	0	0	F[20]	B[57]
0	1	0	1	0	1	F[21]	B[41]
0	1	0	1	1	0	F[22]	B[56]
0	1	0	1	1	1	F[23]	B[40]
0	1	1	0	0	0	F[24]	B[11]
0	1	1	0	0	1	F[25]	B[27]
0	1	1	0	1	0	F[26]	B[10]
0	1	1	0	1	1	F[27]	B[26]
0	1	1	1	0	0	F[28]	B[9]
0	1	1	1	0	1	F[29]	B[25]
0	1	1	1	1	0	F[30]	B[8]
0	1	1	1	1	1	F[31]	B[24]
1	0	0	0	0	0	F[32]	B[55]
1	0	0	0	0	1	F[33]	B[39]
1	0	0	0	1	0	F[34]	B[54]
1	0	0	0	1	1	F[35]	B[38]
1	0	0	1	0	0	F[36]	B[53]
1	0	0	1	0	1	F[37]	B[37]
1	0	0	1	1	0	F[38]	B[52]
1	0	0	1	1	1	F[39]	B[36]
1	0	1	0	0	0	F[40]	B[7]
1	0	1	0	0	1	F[41]	B[23]
1	0	1	0	1	0	F[42]	B[6]
1	0	1	0	1	1	F[43]	B[22]
1	0	1	1	0	0	F[44]	B[5]
1	0	1	1	0	1	F[45]	B[21]
1	0	1	1	1	0	F[46]	B[4]
1	0	1	1	1	1	F[47]	B[20]
1	1	0	0	0	0	F[48]	B[51]
1	1	0	0	0	1	F[49]	B[35]
1	1	0	0	1	0	F[50]	B[50]
1	1	0	0	1	1	F[51]	B[34]
1	1	0	1	0	0	F[52]	B[49]
1	1	0	1	0	1	F[53]	B[33]
1	1	0	1	1	0	F[54]	B[48]
1	1	0	1	1	1	F[55]	B[32]
1	1	1	0	0	0	F[56]	B[3]
1	1	1	0	0	1	F[57]	B[19]
1	1	1	0	1	0	F[58]	B[2]
1	1	1	0	1	1	F[59]	B[18]
1	1	1	1	0	0	F[60]	B[1]
1	1	1	1	0	1	F[61]	B[17]
1	1	1	1	1	0	F[62]	B[0]
1	1	1	1	1	1	F[63]	B[16]

1) Use the FPGA to generate a keystream Z.
2) For each $l \in L$, modify the content of l to l^α in $\mathcal{B}$, where α is the fault to inject.
3) Load $\mathcal{B}^\alpha$ into the FPGA.
4) Use the FPGA to generate a keystream Z^α.
5) Analyze Z^α to extract the key K.
6) Simulate the keystream Z^* using a software model. If $Z^* = Z$, K is correct.

Note that FPGAs usually use Cyclic Redundancy Check (CRC) to verify integrity of bitstream frames. However, since CRC generator polynomials are public, CRC checkbits can re-computed and replaced in the modified bitstream $\mathcal{B}^\alpha$. Alternatively, the CRC check can be disabled. In our experiments, we used the latter approach.

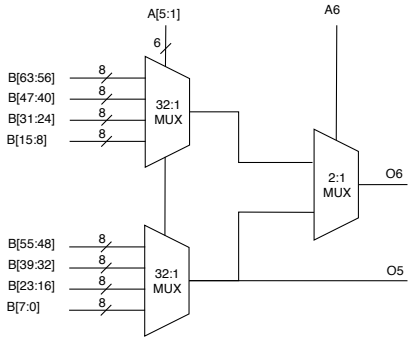

Fig. 4. Xilinx 6-LUT structure [14]

V. ATTACK ON SNOW 3G

A. Verifying LUTs

We applied the strategy described in the previous section to a design of SNOW 3G implemented in a Xilinx Artix-7 FPGA. In the experiments, we used a VHDL implementation of SNOW 3G kindly provided to us by the authors of SNOW 3G. The C code from [37] was used as the software model of SNOW 3G.

B. Choosing fault injection point

From SNOW 3G design description in Section III, we can see that, if we stuck the output word of the FSM to 0 during the initialization, the FSM will not affect the next state of the LFSR. As a result, the non-linear state updating function of the LFSR is reduced to a linear one L defined by the LFSR's characteristic polynomial over $GF(2^{32})$. Such a fault can be injected by fixing to constant-0 the XOR gate marked by v in Fig. 2 and 3. Note that the SNOW 3G is a 32-bit word-oriented cipher. Therefore, v represents a set of 32 2-input XOR gates.

In presence of the fault $\alpha : v = 0$, the LFSR goes through the following states during the initialization:

$$\begin{aligned} S^0 &= \gamma(K, IV) \\ S^1 &= L(\gamma(K, IV)) \\ &\dots \\ S^{32} &= L^{32}(\gamma(K, IV)) \end{aligned}$$

where S^i is the LFSR state at the ith initialization round, for $i \in \{0, 1, \dots, 32\}$, and $\gamma(K, IV)$ is defined in Section III.

If we let SNOW 3G generate 16 words of the keystream in presence of the fault $\alpha : v = 0$, the result is the LFSR state S^{33}. We can reverse the LFSR 33 steps backwards, from S^{33} to S^0, to get $\gamma(K, IV)$ and hence the key K. An LFSR with a known characteristic polynomial is easy to reverse [38].

C. Finding LUTs

We implemented the algorithm FINDLUT() in order to automate the search for LUTs in the bitstream. For bitstreams of size less than 10MB and $k = 6$, our tool takes less than 4 sec to execute for a given f. Table II shows results for different candidate LUTs covering the target gate v. After verifying the candidates, we found that three LUTs listed in the last column contain the target gate v. As we mentioned in Section II-B, FPGA technology mappers usually re-use nodes which are already mapped while searching for k-feasible cuts. This means that nodes are often covered by more than one LUT.

TABLE II
NUMBER OF OCCURRENCES OF DIFFERENT FUNCTIONS IN THE BITSTREAM.

Output	Boolean function implemented by 6-LUT	n	LUT
z_t	$f_1 = (a_1 \oplus a_2 \oplus a_3)a_4a_5a_6$	12	
	$f_2 = (a_1 \oplus a_2 \oplus a_3)a_4a_5\overline{a}_6$	81	LUT_1
	$f_3 = (a_1 \oplus a_2 \oplus a_3)a_4\overline{a}_5\overline{a}_6$	52	
	$f_4 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5\overline{a}_6$	6	
	$f_5 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5$	1	
	$f_6 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4a_5$	12	
	$f_7 = (a_1 \oplus a_2 \oplus a_3)a_4a_5$	1	
s_{15}	$f_8 = (a_1 \oplus a_2)\overline{a}_3a_4a_5 \oplus a_6$	24	LUT_2
	$f_9 = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4a_5 \oplus a_6$	3	
	$f_{10} = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4\overline{a}_5 \oplus a_6$	0	
	$f_{11} = (a_1 \oplus a_2)a_3a_4a_5 \oplus a_6$	3	
	$f_{12} = (a_1 \oplus a_2)a_4a_5 \oplus a_3a_6$	0	
	$f_{13} = (a_1 \oplus a_2)a_4a_5 \oplus \overline{a}_3a_6$	0	
	$f_{14} = (a_1 \oplus a_2)a_4\overline{a}_5 \oplus a_3a_6$	0	
	$f_{15} = (a_1 \oplus a_2)a_4\overline{a}_5 \oplus \overline{a}_3a_6$	0	
	$f_{16} = (a_1 \oplus a_2)\overline{a}_4\overline{a}_5 \oplus a_3a_6$	0	
	$f_{17} = (a_1 \oplus a_2)\overline{a}_4\overline{a}_5 \oplus \overline{a}_3a_6$	0	
	$f_{18} = (a_1 \oplus a_2)a_4 \oplus a_3a_6$	0	
	$f_{19} = (a_1 \oplus a_2)\overline{a}_4 \oplus a_3a_6$	8	LUT_3
	$f_{20} = (a_1 \oplus a_2)a_4 \oplus \overline{a}_3a_6$	0	
	$f_{21} = (a_1 \oplus a_2)\overline{a}_4 \oplus \overline{a}_3a_6$	2	

Next, we explain how we guessed the candidate functions listed in Table II. The FPGAs used in our experiments use 6-input dual-output LUTs (see Fig. 4). These LUTs can implement either a single Boolean function of 6 independent variables, or two Boolean functions of 5 dependent variables.

Based on the block diagram of SNOW 3G shown in Fig. 2 and 3, we can conclude that for both, initialization and keystream generation modes, v is likely to be covered by a k-LUT which implements an XOR of three or more inputs in a combination with multiplexers (MUXes) which switch between different modes of operation. Thus, the number of inputs in the XOR is bounded by the $k - c$, where c is the number of control variables.

From the specification of SNOW 3G [36], it is clear that $c \geq 2$. Apart from the initialization and keystream generation, the cipher should be able to load the values of key K and IV. Table II lists possible Boolean expressions for $c = 2$ and 3. Since we do not know how control variables are encoded, we need to consider different possibilities. However, since FINDLUT() evaluates all possible permutations of k inputs, it is sufficient to consider $c + 1$ choices rather than 2^c.

The next subsection describes how we verified the candidates.

D. Verifying LUTs

The target gate v is contained in two paths: to the output z_t and to the feedback loop. First, we verify the candidates with the largest number of matches.

1) Path to z_t: For the path to z_t, the candidate f_2 has 81 matches, $|\mathcal{L}_{f_2}| = 81$. To check if a LUT l covers v, for each $l \in \mathcal{L}_{f_2}$, we modify the content of l in $\mathcal{B}$ from f_2 to the constant-0, $\alpha : f_2 = 0$, load the faulty bitstream $\mathcal{B}^\alpha$ into the FPGA, and compute w words of the keystream (we used $w = 16$). If the ith bit of each 32-bit word of the keystream is 0, then l passes the check. All elements of $\mathcal{L}_{f_2}$ which overlap with l in $\mathcal{B}$ are removed from $\mathcal{L}_{f_2}$ (because two valid LUTs cannot overlap in a bitstream). If l does not pass the check we remove l from $\mathcal{L}_{f_2}$.

In this way we identified 32 LUTs of $\mathcal{L}_{f_2}$ which implement the ith XOR of v on the path to z_t. In the sequel, we refer to these LUTs as $\text{LUT}_1[i]$, for $i \in \{1, 2, \ldots, 32\}$.

2) Feedback loop path: For the feedback loop path, none of the candidate LUTs in Table II has 32 or more matches. However, the sum of matches for f_8 and f_{19} is $|\mathcal{L}_{f_8}| + |\mathcal{L}_{f_{19}}| = 32$. This is expected since the operations "$\alpha \odot$"/"$\alpha^{-1} \odot$" perform a byte shift of the 32-bit input word to the left/right and then XOR the result with the output of the 8-bit into 32-bit mapping MUL_α/DIV_α. Due to the byte shift, the implementations of SNOW 3G may process 24 bits of the word in one way and the remaining byte in another.

Note that the sum of matches for f_9, f_{11} and f_{21} is also 8. However, by examining their byte positions in the bitstream we can see that they are the same as for f_{19}. Therefore, we make a hypothesis that 24 LUT_2 corresponding to f_8 and 8 LUT_3 corresponding to f_{19} implement v on the feedback loop path.

E. Modifying the bitstream

To check our hypothesis, we need to apply the procedure described in Section V-A to the faulty bitstream $\mathcal{B}^\alpha$ in which the fault $\alpha : v = 0$ is injected into 32 $\text{LUT}_1[i]$, $\forall i \in \{1, 2, \ldots, 32\}$, 24 LUT_2 and 8 LUT_3. We know that the fault $\alpha : v = 0$ can be injected into LUT_2 and LUT_3 by modifying their functions as:

$$\begin{array}{lcl} f_8 = (a_1 \oplus a_2)\overline{a}_3a_4a_5 \oplus a_6 & \rightarrow & f_8^\alpha = a_6 \\ f_{19} = (a_1 \oplus a_2)\overline{a}_4 \oplus a_3a_6 & \rightarrow & f_{19}^\alpha = a_3a_6, \end{array} \quad (1)$$

but for the $\text{LUT}_1[i]$ we do not know which variables of $f_2 = (a_1 \oplus a_2 \oplus a_3)a_4a_5\overline{a}_6$ correspond to the inputs of the gate v. Clearly, they are one of the pairs (a_1, a_2), (a_1, a_3) or (a_2, a_3). The 3rd variable of the XOR corresponds to s_0. But we cannot distinguish among XOR's inputs by analyzing keystream since the key is unknown and hence keystream cannot be predicted. So, all possible 3^{32} combinations have to be considered to find which of the pairs (a_1, a_2), (a_1, a_3) or (a_2, a_3) are inputs of v.

However, if we make the keystream *key-independent*, we will be able to distinguish among XOR's inputs in constant time by computing two keystreams. Key-independence gives us another degree of freedom in exploring bitstreams. This idea has not been exploited in previous bitstream modification attacks.

1) Making keystream key-independent: Keystream can be made key-independent by loading the all-0 vector instead of $\gamma(K, IV)$ into the LFSR at the initialization stage. On one hand, the LFSR initialized to the all-0 state will remain in the all-0 state if the feedback path contains the fault $\alpha : v = 0$. On the other hand, the FSM initialized to the all-0 state will end up in a non-0 state, independently of the LFSR state. This allows us to distinguish between the input s_0, which always has 0 value, and the inputs of v.

Let $\mathcal{B}^{\alpha_1,\beta}$ be the bitstream $\mathcal{B}$ with the two faults injected. The fault β causes the all-0 vector to be loaded into the LFSR instead of $\gamma(K, IV)$. We explain how β can be injected in the next subsection. The fault α_1 sets $v = 0$ in all LUTs implementing v on the feedback path by modifying their functions as (1). In its essence, the modification (1) disconnects the FSM from the LFSR during the initialization.

To distinguish between the input s_0 and the inputs of v, the following loop is repeated. First we check if (a_1, a_2) are the inputs of v in $\text{LUT}_1[i]$, for all $i \in \{1, 2, \ldots, 32\}$:

1) Modify the content of all $\text{LUT}_1[i]$ in $\mathcal{B}^{\alpha_1,\beta}$ from $f_2 = (a_1 \oplus a_2 \oplus a_3)a_4a_5\overline{a}_6$ to $f_2^{\alpha_2} = a_3a_4a_5\overline{a}_6$, where $\alpha_2 : v = 0$ in $\text{LUT}_1[i]$.
2) Load the faulty bitstream $\mathcal{B}^{\alpha_1,\alpha_2,\beta}$ into the FPGA.
3) Compute w words of the keystream.

If the ith bit of each word of the keystream is 0, (a_1, a_2) are the inputs of v in $\text{LUT}_1[i]$. Otherwise, repeat the steps 1-3 for the pair (a_1, a_3). If the ith bit of each word of the keystream is 0, (a_1, a_3) are the inputs of v in $\text{LUT}_1[i]$. Otherwise, (a_2, a_3) are the inputs of v in $\text{LUT}_1[i]$.

TABLE III
KEY-INDEPENDENT KEYSTREAM GENERATED BY SNOW 3G WHEN THE FSM OUTPUT IS STUCK TO 0 DURING THE INITIALIZATION STAGE AND THE LFSR IS INITIALIZED TO ALL-0 STATE.

t	z_t
1	a1fb4788
2	e4382f8e
3	3b72471c
4	33ebb59a
5	32ac43c7
6	5eebfd82
7	3a325fd4
8	1e1d7001
9	b7f15767
10	3282c5b0
11	103da78f
12	e42761e4
13	c6ded1bb
14	089fa36c
15	01c7c690
16	bf921256

The above procedure requires two keystream computations to find which variables of f_2 correspond to the inputs of v in LUT$_1[i]$, for all $i \in \{1, 2, \ldots, 32\}$.

2) Loading the LFSR with 0s: The fault which causes the LFSR to load the all-0 vector can be injected by finding in the bitstream $\mathcal{B}$ all MUXes used to load $\gamma(K, IV)$ into the LFSR and modifying them to load 0s instead. If the key K and IV are loaded in parallel, each LFSR stage s_j, $j \in \{0, 1, \ldots, 14\}$, takes as input the output of a MUX which has s_{j-1} as one input and the jth element of $\gamma(K, IV)$ as another. Such a MUX can be implemented by 16 6-input dual-output LUTs, LUT$_{MUX2}$, each implementing a 2-to-1 MUX functionality for each of the two outputs. This would result in a total of 240 LUTs of type:

$$f_{MUX2} = a_6(a_1a_2 + \overline{a}_1a_3) + \overline{a}_6(a_1a_4 + \overline{a}_1a_5),$$

where "+" in the Boolean OR, a_6 is the input dedicated to switching between the two outputs of a dual-output LUT (see Fig. 4), and a_1 is the control input of the MUX. Note that, if the key is stored in the bitstream, the above expression may get optimized. To load the all-0 vector instead of the initial state $\gamma(K, IV)$, all LUT$_{MUX2}$ are modified to

$$f^{\alpha}_{MUX2} = a_6\overline{a}_1a_3 + \overline{a}_6\overline{a}_1a_5,$$

where $\alpha : \gamma(K, IV) = 0$. The reduction above assumes that the initial state $\gamma(K, IV)$ is loaded into the LFSR when the MUX control input has value $a_1 = 1$.

The LUTs implementing MUXes which load $k_3 \oplus iv_0$ into the stage s_{15} can be found by searching through all possible candidates similarly to the procedure described in Section VC. Then, the LUTs are modified to load 0 instead.

The FPGA loaded with the modified bitstream generates the keystream shown in Table III. One can verify that the above keystream is correct by simulating it using the software model of SNOW 3G in which the LFSR is initialized to all-0 state and the FSM output is stuck to 0 during the initialization stage.

3) Key extraction: After the fault $\alpha : v = 0$ is injected into 32 LUT$_1[i]$, $\forall i \in \{1, 2, \ldots, 32\}$, 24 LUT$_2$ and 8 LUT$_3$ by modifying f_2, f_8 and f_{19} to $f^{\alpha}_2, f^{\alpha}_8$ and f^{α}_{19} as explained above, we load the faulty bitstream $\mathcal{B}^{\alpha}$ into the FPGA, compute the keystream Z^{α}, and recover the key K by analyzing Z^{α} as described in Section V-B.

The FPGA loaded with $\mathcal{B}^{\alpha}$ generates the keystream shown in Table IV.

This keystream corresponds to the first LFSR state after the initialization, S^{33}. We can reverse the LFSR 33 steps backwards

TABLE IV
KEYSTREAM GENERATED BY SNOW 3G WHEN THE FSM OUTPUT IS STUCK TO 0 DURING THE INITIALIZATION AND THE KEYSTREAM GENERATION STAGES.

t	z_t
1	3ffe4851
2	35d1c393
3	5914acef
4	e98446cc
5	689782d9
6	8abdb7fc
7	a11b0377
8	5a2dd294
9	5deb29fa
10	c2c6009a
11	a82ee62f
12	925268ed
13	d04e2c33
14	3890311b
15	e8d27b84
16	a70aeeaa

TABLE V
THE RECOVERED INITIAL LFSR STATE S^0.

i	s_i
0	d429ba60
1	7d3a4cff
2	6ad3b6ef
3	b77e00b7
4	2bd6459f
5	82c5b300
6	952c4910
7	4881ff48
8	d429ba60
9	6131b8a0
10	b5cc2dca
11	b77e00b7
12	868a081b
13	82c5b300
14	952c4910
15	a283b85c

to get the initial state S^0 shown in Table V. From $s_4 - s_7$, we can conclude that the key is:

2bd6459f 82c5b300 952c4910 4881ff48

One can verify that the key is correct by simulating the keystream using a software model of SNOW 3G.

VI. COUNTERMEASURES

As we mentioned in Section II-B, FPGA technology mappers usually re-use nodes which are already mapped while searching for k-feasible cuts. This often makes the size of LUTs larger (LUT's *size* is the number of nodes covered by a LUT). Large LUTs have a distinct structure and may be an easier target for reverse engineering.

We recommend FPGA designers to constrain technology mappers to generate a k-LUT cover with smaller LUTs, ideally covering only one node. In this case locating a common gate such as an XOR in a bitstream becomes a much more difficult problem. For example, in Xilinx products, such constraints can be applied with the use of primitives like KEEP or DONT_TOUCH in the source code. To minimize performance penalty due to the non-optimal depth of the k-LUT cover, critical paths may be covered with larger LUTs and non-critical paths with smaller ones.

As a proof of concept, we applied the proposed countermeasure to SNOW 3G and evaluated its resistance. In the unprotected SNOW 3G implementation, the critical path (6.313 ns delay) is between the

TABLE VI
NUMBER OF OCCURRENCES OF DIFFERENT FUNCTIONS IN THE PROTECTED BITSTREAM.

Output	Boolean function implemented by 6-LUT	n
z_t	$f_1 = (a_1 \oplus a_2 \oplus a_3)a_4a_5a_6$	20
	$f_2 = (a_1 \oplus a_2 \oplus a_3)a_4a_5\overline{a}_6$	48
	$f_3 = (a_1 \oplus a_2 \oplus a_3)a_4\overline{a}_5\overline{a}_6$	28
	$f_4 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5\overline{a}_6$	5
	$f_5 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5$	0
	$f_6 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4a_5$	8
	$f_7 = (a_1 \oplus a_2 \oplus a_3)a_4a_5$	17
s_{15}	$f_8 = (a_1 \oplus a_2)\overline{a}_3a_4a_5 \oplus a_6$	0
	$f_9 = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4a_5 \oplus a_6$	0
	$f_{10} = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4\overline{a}_5 \oplus a_6$	0
	$f_{11} = (a_1 \oplus a_2)a_3a_4a_5 \oplus a_6$	0
	$f_{12} = (a_1 \oplus a_2)a_4a_5 \oplus a_3a_6$	0
	$f_{13} = (a_1 \oplus a_2)a_4a_5 \oplus \overline{a}_3a_6$	0
	$f_{14} = (a_1 \oplus a_2)a_4\overline{a}_5 \oplus a_3a_6$	0
	$f_{15} = (a_1 \oplus a_2)a_4\overline{a}_5 \oplus \overline{a}_3a_6$	0
	$f_{16} = (a_1 \oplus a_2)\overline{a}_4\overline{a}_5 \oplus a_3a_6$	0
	$f_{17} = (a_1 \oplus a_2)\overline{a}_4\overline{a}_5 \oplus \overline{a}_3a_6$	0
	$f_{18} = (a_1 \oplus a_2)a_4 \oplus a_3a_6$	0
	$f_{19} = (a_1 \oplus a_2)\overline{a}_4 \oplus a_3a_6$	0
	$f_{20} = (a_1 \oplus a_2)a_4 \oplus \overline{a}_3a_6$	0
	$f_{21} = (a_1 \oplus a_2)\overline{a}_4 \oplus \overline{a}_3a_6$	0

registers R_1 and R_2, where S-box is evaluated by a block RAM (BRAM) lookup. The feedback path from s_{15} to s_{15} is not in the list of the ten slowest paths. We constrained the six 32-bit XORs shown in Fig. 2 to be covered by separate LUTs. Note that, as a result, the path from MUL_α to s_{15} became critical (7.514 ns delay).

To evaluate the protected design, we first applied the strategy described in Section V-C to find possible candidates covering the target XOR gate v. The results are shown in Table VI. As one can see, the obtained information is not useful.

Next, we wrote a program which finds all LUTs having a 2-input XOR as one half of their truth table and any 5-variable Boolean function as the other in the bitstream. The unconstrained search over all byte positions in the bitstream returns 481 hits. After experimenting with the functions in Table VI, we can guess where LUTs are located in the bitstream and limit the search. The constrained search over an interval of 200.000 byte positions (out of the 3.825.888 possible ones) returns 203 hits.

The 32 LUTs implementing 32 XORs with output z_t can be pruned from the set of candidates using the approach described in Section V-D. However, it does not seem possible to find which 32 out of 171 remaining candidates implement the target gate v without exhaustively searching through the $\binom{171}{32} = 4.9 \times 10^{34}$ possible choices. For each choice, all steps involved in making the bitstream key-independent have to be repeated until we find the bitstream which generates the keystream shown in Table III. This seems practically infeasible.

VII. CONCLUSION

We demonstrated that it is possible to extract the key from some FPGA implementations of SNOW 3G by the means of bitstream modification. We proposed a countermeasure which increases the number of candidate points for fault injection exponentially, making the presented attack infeasible in practice.

Our results are expected to help FPGA designers protect their products against reverse engineering and bitstream modifications.

Xilinx was notified about the results our work via psirt@xilinx.com.

REFERENCES

[1] M. Moraitis and E. Dubrova, "Bitstream modification attack on snow 3g," in *Proceedings of the 2020 Design, Automation & Test in Europe Conf. & Exhibition (DATE'20)*, 2020.

[2] A. Biryukov, D. Priemuth-Schmid, and B. Zhang, "Analysis of SNOW 3G resynchronization mechanism," pp. 327–333, 01 2010.

[3] A. Kircanski and A. M. Youssef, "On the sliding property of SNOW 3G and SNOW 2.0," *IET Information Security*, vol. 5, no. 4, p. 199, 2011.

[4] J. GUAN, L. DING, and S.-K. LIU, "Guess and Determine Attack on SNOW3G and ZUC [J]," *Journal of Software*, vol. 6, pp. 1324–1333, 2013.

[5] M. S. N. Nia and T. Eghlidos, "Improved Heuristic guess and determine attack on SNOW 3G stream cipher," in *7'th Int. Symp. on Telecommunications (IST'2014)*, pp. 972–976, IEEE, 2014.

[6] A. Biryukov, D. Priemuth-Schmid, and B. Zhang, "Multiset collision attacks on reduced-round SNOW 3G and SNOW 3G$^{\oplus}$," in *Int. Conf. on Applied Crypt. and Network Security*, pp. 139–153, Springer, 2010.

[7] P. Trott, "Preventing overbuilding and cloning of electronic systems secure production programming." Microsemi Corporation Report, 2015.

[8] J.-B. Note and É. Rannaud, "From the bitstream to the netlist.," in *FPGA*, vol. 8, pp. 264–264, 2008.

[9] Z. Ding, Q. Wu, Y. Zhang, and L. Zhu, "Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 299–312, 2013.

[10] T. Zhang, J. Wang, S. Guo, and Z. Chen, "A comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code," *IEEE Access*, vol. 7, pp. 38379–38389, 2019.

[11] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *22nd Int. Conf. on Field Programmable Logic and Applications (FPL)*, pp. 735–738, IEEE, 2012.

[12] J. Yoon, Y. Seo, J. Jang, M. Cho, J. Kim, H. Kim, and T. Kwon, "A bitstream reverse engineering tool for FPGA hardware trojan detection," in *Proceedings of the 2018 ACM SIGSAC Conf. on Computer and Communications Security*, pp. 2318–2320, ACM, 2018.

[13] C. Wolf and M. Lasser, "Project IceStorm." http://www.clifford.at/icestorm/.

[14] M. Jeong, J. Lee, E. Jung, Y. H. Kim, and K. Cho, "Extract LUT logics from a downloaded bitstream data in FPGA," in *2018 IEEE Int. Symp. on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2018.

[15] SymbiFlow Team, "Project X-Ray." https://prjxray.readthedocs.io/en/latest/.

[16] A. Moradi, D. Oswald, C. Paar, and P. Swierczynski, "Side-channel attacks on the bitstream encryption mechanism of Altera Stratix II: facilitating black-box analysis using software reverse-engineering," in *Proceedings of the ACM/SIGDA Int. Symp. on Field programmable gate arrays*, pp. 91–100, ACM, 2013.

[17] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs," in *Proceedings of the 18th ACM Conf. on Computer and communications security*, pp. 111–124, ACM, 2011.

[18] A. Moradi and T. Schneider, "Improved side-channel analysis attacks on Xilinx bitstream encryption of 5, 6, and 7 series," in *Int. Workshop on Constructive Side-Channel Analysis and Secure Design*, pp. 71–87, Springer, 2016.

[19] M. Alderighi, S. D'Angelo, M. Mancini, and G. R. Sechi, "A fault injection tool for SRAM-based FPGAs," in *9th IEEE On-Line Testing Symp., 2003. IOLTS 2003.*, pp. 129–133, July 2003.

[20] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware Trojan insertion by direct modification of FPGA configuration bitstream," *IEEE Design & Test*, vol. 30, no. 2, pp. 45–54, 2013.

[21] P. Swierczynski, G. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (BiFI) – automated fault attacks against SRAM-based FPGAs," *IEEE Trans. on Computers*, vol. 76, pp. 1–1, 2018.

[22] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA trojans through detecting and weakening of cryptographic primitives," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, pp. 1236–1249, Aug 2015.

[23] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice—hardware trojan against a high-security USB flash drive," *Journal of Cryptographic Engineering*, vol. 7, pp. 199–211, Sep 2017.

[24] B. Debraize and I. M. Corbella, "Fault analysis of the stream cipher SNOW 3G," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 103–110, IEEE, 2009.

[25] J. Takahashi, Y.-i. Hayashi, N. Homma, H. Fuji, and T. Aoki, "Feasibility of fault analysis based on intentional electromagnetic interference," in *2012 IEEE Int. Symp. on Electromagnetic Compatibility*, pp. 782–787, IEEE, 2012.

[26] B. B. Brumley, R. M. Hakala, K. Nyberg, and S. Sovio, "Consecutive S-box lookups: A Timing Attack on SNOW 3G," in *Int. Conf. on Information and Communications Security*, pp. 171–185, Springer, 2010.
[27] S. Hurst, D. Miller, and J. Muzio, *Spectral Techniques in Digital Logic*. Academic Press, 1985.
[28] S. M. Trimberger and J. J. Moore, "Fpga security: Motivations, features, and applications," *Proceedings of the IEEE*, vol. 102, pp. 1248–1265, Aug 2014.
[29] S. Hassoun and S. Tsutomu, *Logic Synthesis and Verification*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.
[30] R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based FPGAs," in *Proceedings of the 28th ACM/IEEE Design Automation Conf.*, pp. 227–233, 1991.
[31] K.-C. Chen, J. Cong, Y. Ding, and A. Kahng, "Dag-map: Graph-based FPGA technology mapping for delay optimization," *IEEE Desings and Test of Computers*, vol. 9, pp. 7–20, September 1992.
[32] M. Teslenko and E. Dubrova, "Hermes: LUT FPGA technology mapping algorithm for area minimization with optimum depth," in *IEEE/ACM Int. Conf. on Computer Aided Design*, pp. 748–751, Nov 2004.
[33] M. Schlag, J. Kong, and P. K. Chan, "Routability-driven technology mapping for lookup table-based FPGAs," in *Proceedings of the Int. Conf. on Computer Design*, pp. 86–90, 1992.
[34] H. Li, S. Katkoori, and W.-K. Mak, "Power minimization algorithms for LUT-based FPGA technology mapping," *ACM Trans. on Design Automation of Electronic Systems*, vol. 9, no. 1, pp. 33–51, 2004.
[35] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. ACM Intl. Symp. on FPGA*, pp. 29–35, February 1999.
[36] ETSI/SAGE, "Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. document 2: SNOW 3G specification," 2009. http://cryptome.org/uea2-uia2/uea2-uia2.htm.
[37] Jake "KsirbJ" Brisk, "C++ code for SNOW 3G." https://github.com/KsirbJ/SNOW-3G.
[38] S. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.