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 a known plaintext-ciphertext pair. To our best knowledge, this is the first successful bitstream modification attack on SNOW 3G.**

***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 the physical security of cryptographic algorithm FPGA implementations. 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 the 3G UMTS standard, 128-EEA1 and 128-EIA1 algorithms in the 4G LTE standard, and 128-NEA1 and 128-NIA1 algorithms in the NG 5G standard. The popularity of SNOW 3G is to a large extent due to its known resistance to classical cryptanalysis.

One of the most popular types of physical attacks on FPGAs is the *reverse engineering* of bitstream. Reverse engineering enables copying designs which cost millions of dollars to develop [1]. One of the countermeasures against reverse engineering is bitstream *obfuscation*. 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 an algorithm's security. Several reverse engineering tools have been created for older Xilinx FPGA families [2]–[6]. A full Verilog-to-bitstream flow has been developed for Lattice iCE40 [7]. Last year, information about the bitstream format of the latest Xilinx 7 series FPGA has been revealed [8], [9].

Bitstream *encryption* is another countermeasure which, in theory, should protect against reverse engineering. In reality, however, the security of encryption (or any other cryptographic algorithm) is not greater than the security of its secret key generation/storage mechanism. It is known that the encryption key used for the Advanced Encryption Standard (AES)-256-based bitstream encryption of Altera and Xilinx FPGAs can be extracted from a side-channel analysis [10]–[12].

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

Previous Work. In several works [13]–[15] 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 [16]. 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. 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 [17], an attack using 22 transient fault injections to recover the key of SNOW 3G is described. In [18] it is stated that stream ciphers, including SNOW 3G, may be vulnerable to faults caused by intentional electromagnetic interference. In [19], a cache timing attack capable of recovering the internal state of SNOW 3G from empirical timing data is illustrated.

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 the LUTs implementing the gate in the bitstream. As a result, the SNOW 3G algorithm becomes weaker and its key can be recovered from a keystream.
- 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 any previous attack.
- We created a tool which automatically finds a k-input LUT implementing a given k-variable Boolean function and all Boolean functions within the same P equivalence class[1] in the bitstream.

[1]Two Boolean functions belong to the same P equivalence class, if f can be transformed into g through permutation of inputs [20].

978-3-9819263-4-7/DATE20/©2020 EDAA

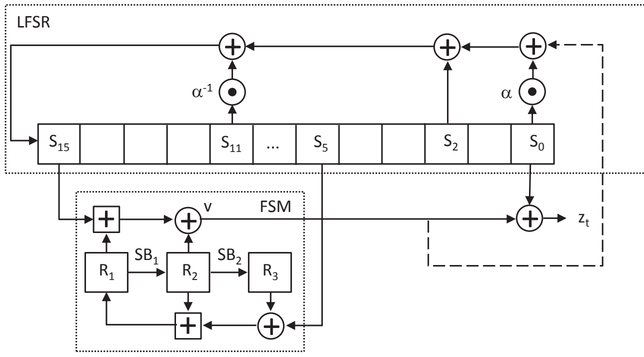

Fig. 1. Block diagram of SNOW 3G.

II. SNOW 3G Design description

SNOW 3G is a 32-bit word-based stream cipher constructed from a Linear Feedback Shift Register (LFSR) and a Finite State Machine (FSM) as shown in Fig. 1. The gates denoted by "$\oplus$" and "$\boxplus$" stand for the bit-wise 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 [21].

The state of the LFSR is a vector S consisting of the sixteen 32-bit stages $(s_0, s_1, \ldots, s_{15})$. The FSM consists of the three 32-bit registers R_1, R_2 and R_3. 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.

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

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

where **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 by the dashed line in Fig. 1. Then, the following two-step round is repeated 32 times:

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

No keystream is generated during the initialization.

III. Attack on SNOW 3G

In this section we describe our attack on SNOW 3G implemented in a modern FPGA. We used a VHDL implementation that was kindly provided to us by the authors of SNOW 3G. As for a software model, we used C code from [22].

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 $\mathcal{B}$

1: $\mathcal{L} = \emptyset$;
2: $P_k =$ COMPUTEPERMUTATIONS(k);
3: $P_r =$ COMPUTEPERMUTATIONS(r);
4: **for** each $p \in P_k$ **do**
5: $\quad F =$ GETTRUTHTABLE(l, p);
6: $\quad B = \xi(F)$; /* permutes the truth table F */
7: $\quad B = (B_1||B_2||\ldots||B_r), |B_i| = |B_j|, \forall i, j \in \{1, 2, \ldots, r\}$;
8: $\quad m = 2^k/r - 1$;
9: $\quad$ **for** each i from 0 to $|\mathcal{B}| - (r-1)d$ **do**
10: $\quad\quad$ **if** i is not marked **then**
11: $\quad\quad\quad$ **for** each $(j_1, j_2, \ldots, j_r) \in P_r$ **do**
12: $\quad\quad\quad\quad$ **if** $((b_i, \ldots, b_{i+m}) = B_{j_1})\&((b_{i+d}, \ldots, b_{i+d+m}) = B_{j_2})\&\ldots\&((b_{i+(r-1)d}, \ldots, b_{i+(r-1)d+m}) = B_{j_r})$ **then**
13: $\quad\quad\quad\quad\quad \mathcal{L} = \mathcal{L} \cup \{l\}$;
14: $\quad\quad\quad\quad\quad$ MARK(i);
15: $\quad\quad\quad\quad$ **end if**
16: $\quad\quad\quad$ **end for**
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **end for**
20: **return** $\mathcal{L}$

A. Choosing fault injection point

From the SNOW 3G design description in Section II, 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 the output of the XOR gate marked by v in Fig. 1 to constant-0. Note that SNOW 3G is a 32-bit word-oriented cipher, therefore, v represents a set of 32 2-input XORs.

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

$$
\begin{array}{ll}
S^0 = \gamma(K, IV), & S^1 = L(\gamma(K, IV)), \\
\ldots, & \\
S^{31} = L^{31}(\gamma(K, IV)), & S^{32} = L^{32}(\gamma(K, IV))
\end{array}
$$

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

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^{32}. We can reverse the LFSR 32 steps backwards, from S^{32} to S^0, to get $\gamma(K, IV)$ and hence the key K. An LFSR with a known characteristic polynomial is easy to reverse.

B. Finding LUTs

The pseudo-code of the algorithm for finding k-LUTs implementing a given Boolean function f in the bitstream is shown as Algorithm 1.

TABLE I

Output	Boolean function implemented by k-LUT	n	LUT
z_t	$f_1 = (a_1 \oplus a_2 \oplus a_3)a_4a_5a_6$	18	
	$f_2 = (a_1 \oplus a_2 \oplus a_3)a_4a_5\overline{a}_6$	99	LUT_1
	$f_3 = (a_1 \oplus a_2 \oplus a_3)a_4\overline{a}_5\overline{a}_6$	60	
	$f_4 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5\overline{a}_6$	13	
	$f_5 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4\overline{a}_5$	1	
	$f_6 = (a_1 \oplus a_2 \oplus a_3)\overline{a}_4a_5$	16	
	$f_7 = (a_1 \oplus a_2 \oplus a_3)a_4a_5$	0	
s_{15}	$f_8 = (a_1 \oplus a_2)\overline{a}_3a_4a_5 \oplus a_6$	24	LUT_2
	$f_9 = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4a_5 \oplus a_6$	0	
	$f_{10} = (a_1 \oplus a_2)\overline{a}_3\overline{a}_4\overline{a}_5 \oplus a_6$	0	
	$f_{11} = (a_1 \oplus a_2)a_3a_4a_5 \oplus a_6$	5	
	$f_{12} = (a_1 \oplus a_2)a_4a_5 \oplus a_3a_6$	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	
	$f_{21} = (a_1 \oplus a_2)\overline{a}_4 \oplus \overline{a}_3a_6$	0	

First, the truth table F of the function f is derived. Then, F is mapped according to the obfuscation function ξ which is specific for a given FPGA family. The obfuscation is typically performed in two steps. F, is first permuted as $\xi : F \rightarrow B$, where ξ is a mapping of type $\{0,1\}^k \rightarrow \{0,1\}^k$ (e.g. for Xilinx 7 series FPGA the defining table of ξ can be found in [8]). Then, vector B is partitioned into r sub-vectors of equal size which are placed on a fixed offset d from each other in the bitstream in varying order. For bitstreams of size less than 10MB and $k = 6$, our C implementation of $\text{FINDLUT}()$ takes less than 4 sec to execute for a given f.

We derived the candidate functions as follows. From the block diagram of SNOW 3G we can see 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.

Apart from the initialization and keystream generation the cipher should be able to load the values of key K and IV. Therefore, it is clear that $c \geq 2$. Table I enumerates possible Boolean expressions for $c = 2$ and 3 for different encodings of control variables. Note that, since $\text{FINDLUT}()$ evaluates all possible permutations of k inputs, it is sufficient to consider $c + 1$ choices of encodings rather than 2^c.

After verifying the candidates as explained next, we found that three LUTs listed in the last column contain the target gate v. FPGA technology mappers usually re-use nodes which are already mapped while searching for k-feasible cuts [23]. Nodes are therefore often covered by more than one LUT.

C. Verifying LUTs

The target gate v should be included in two paths: one to the output z_t and one to the feedback loop. For both paths, we verified the candidates in Table I starting from the ones with the largest number of matches.

1) Path to z_t: For the path to z_t, the candidate f_2 has 99 matches, $|\mathcal{L}_{f_2}| = 99$. To check if a LUT l covers v, for each $l \in \mathcal{L}_{f_2}$, we can 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 $w = 32$). 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 it is possible to identify 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 I 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. Therefore, we can make a hypothesis that 24 LUT_2 and 8 LUT_3 implement v on the feedback loop path.

D. Modifying the bitstream

The fault $\alpha : v = 0$ can be injected into LUT_2 and LUT_3 by modifying their functions as:

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

However, for $\text{LUT}_1[i]$ 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. All possible 3^{32} combinations have to be considered to find which pair (a_1, a_2), (a_1, a_3) or (a_2, a_3) corresponds to the inputs of v. In the next subsection we show that we can distinguish among the XOR's inputs in linear time in the word size by making the keystream *key-independent*.

1) Making keystream key-independent: Keystream can be made key-independent by loading the LFSR with an all-0 vector instead of $\gamma(K, IV)$ 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 the value of 0, 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 modifies the functions on the feedback path as (1), disconnecting 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 for each $i \in \{1, 2, \ldots, 32\}$. First we check if (a_1, a_2) are the inputs of v in $\text{LUT}_1[i]$:

1) Modify the content of $\text{LUT}_1[i]$ 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 $\text{LUT}_1[i]$.

The above procedure requires at most 64 keystream computations to find which variables of f_2 correspond to the inputs of v in $\text{LUT}_1[i]$, for all $i \in \{1, 2, \ldots, 32\}$.

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 be re-computed and replaced in the modified bitstream. Alternatively, the CRC check can be disabled. In our experiments, we used the latter approach.

2) Loading the LFSR with 0s: The fault which causes the LFSR to be loaded with the all-0 vector can be injected by finding in the bitstream $\mathcal{B}$ all the MUXes that load the initial state into the LFSR and modifying them to load constant-0. If the key K and IV are loaded in parallel, each such MUX has the jth element of the initial state as one input and the LFSR stage s_{j+1} as another input, for $j \in \{0, 1, \ldots, 14\}$. These MUXes are likely to be implemented in pairs by dual-output LUTs, resulting in 240 LUT_{2MUX} in total. If the key is hard-coded, the MUX expression may get optimized.

The most difficult part is finding MUXes which load the stage s_{15}. In our case, MUXes for the stage s_{15} were represented by four different types of expressions.

E. Key extraction

After 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 by modifying f_2, f_8 and f_{19} to f_2^α, f_8^α and f_{19}^α as explained above, one can load the faulty bitstream $\mathcal{B}^\alpha$ into the FPGA, compute the keystream, and recover the key K as described in Section III-A.

To verify if the recovered key K is correct, a fault-free keystream can be simulated using the software model of SNOW 3G. If this keystream is the same as the one computed by the FPGA loaded with the fault-free bitstream $\mathcal{B}$, K is the correct key.

IV. Conclusion

We demonstrated that it is possible to extract a key from an FPGA implementation of SNOW 3G by bitstream modification.

Acknowledgment

This work was supported in part by the research grants 2017-05232 and 2018-03964 from VINNOVA.

References

[1] P. Trott, "Preventing overbuilding and cloning of electronic systems secure production programming." Microsemi Corporation Report, 2015.
[2] J.-B. Note and É. Rannaud, "From the bitstream to the netlist.," in *FPGA*, vol. 8, pp. 264–264, 2008.
[3] 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.
[4] 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.
[5] 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.
[6] 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.
[7] C. Wolf and M. Lasser, "Project IceStorm." http://www.clifford.at/icestorm/.
[8] 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.
[9] SymbiFlow Team, "Project X-Ray." https://prjxray.readthedocs.io/en/latest/.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] 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.
[20] S. Hurst, D. Miller, and J. Muzio, *Spectral Techniques in Digital Logic*. Academic Press, 1985.
[21] 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.
[22] Jake "KsirbJ" Brisk, "C++ code for SNOW 3G." https://github.com/KsirbJ/SNOW-3G.
[23] 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.