GRINCH: A Cache Attack against GIFT Lightweight Cipher

Cezar Reinbrecht, Abdullah Aljuffri, Said Hamdioui, Mottaqiallah Taouil
Delft University of Technology – EEMCS Faculty
Delft, The Netherlands
{c.r.wedigreinbrecht,a.a.aljuffri,s.hamdioui,m.taouil}@tudelft.nl

Johanna Sepúlveda
Airbus Defense and Space
Munich, Germany
johanna.sepulveda@airbus.com

Abstract—**The National Institute of Standard and Technology (NIST) has recently started a competition with the objective to standardize lightweight cryptography (LWC). The winning schemes will be deployed in Internet-of-Things (IoT) devices, a key step for the current and future information and communication technology market. GIFT is an efficient lightweight cipher and it is used by one-fourth of the LWC candidates in the NIST LWC competition. Thus, its security evaluation is critical. One vital threat to the security are so-called logical side-channel attacks based on cache observations. In this work, we propose a novel cache attack on GIFT referred to as GRINCH. We analyzed the vulnerabilities of GIFT and exploited them in our attack. The results show that the attack is effective and that the full key could be recovered with less than 400 encryptions.**

Index Terms—**Lightweight cipher, GIFT cipher, cache attack, Micro-architectural attack.**

I. Introduction

The wide use of Internet-of-Thing (IoT) devices is transforming many domains, including the automation industry, automotive, avionics, and healthcare [1]. It is expected that by 2021, 25 billion IoT devices are deployed [1]. IoT devices are highly-constrained devices that are widely and seamless deployed and blended in their environment. The key role of such devices is the remote monitoring of critical systems and collecting of data; both put severe requirements on the security. The current standardized cryptographic algorithms are typically designed for desktop/server environments and are not suited for constrained devices [2]. To cope with this challenge, there is a strong need for secure lightweight cryptography (i.e., ciphers, hash functions and protocols).

Currently there is an ongoing NIST competition to standardize lightweight cryptography (LWC NIST competition) [2]. To this end, a portfolio of new block ciphers will emerge as a standard for lightweight authenticated encryption in the upcoming years. Among the 32 candidates of the second competition round, 7 are based on GIFT cipher [3]. GIFT is based on a substitution-permutation network (SPN) cipher and was proposed in [4] as an improvement to the well-known PRESENT cipher [5] which is part of the ISO standards ISO/IEC 29192-2:2012 and ISO/IEC 29192-5:2016. Currently, GIFT needs the least amount of operations per bit [4]. Although the computational security of GIFT has been widely studied, GIFT is prone to implementation attacks [6, 7]. During operation, the secret key may passively be inferred through side-channels. To the best of our knowledge, only two works have studied implementation vulnerabilities of GIFT [6] [7]. In [6], the authors have demonstrated a power attack on GIFT, while the authors in [7] a fault injection attack. Yet, cache attacks against GIFT have never been demonstrated before.

In this paper we propose GRINCH, the first cache attack on GIFT. Caches are usually shared memories that are used to speedup the execution of the cryptographic algorithms. However, they become a security threat when mutually accessed by multiple processes. A malicious process may gather information to reveal the secret key by: observing the execution time (time-driven attack) [8], exploiting the access pattern (access-driven attack) [9], or inferring the sequence of hits and misses (trace-driven attack) [10]. GRINCH crafts specific inputs to the cipher to extract sensitive data by observing its cache accesses. Hence, it is an access-driven cache attack. In summary, the contributions of the paper are:

- Analysis of GIFT vulnerabilities
- Implementation of the GRINCH attack
- Evaluation of the impact of the cache configuration on the attack efficiency
- Practical demonstration of the attack with two hardware platforms (SoC and MPSoC) in an FPGA
- Proposal of two countermeasures to protect GIFT

The rest of the paper is organized as follows. Section II describes the GIFT cipher. Section III presents the threat model and GRINCH attack. Section IV presents the validation results. Finally, Section V concludes the paper.

II. GIFT Cipher

Like most popular standardized symmetric cryptographic algorithms (such as Advanced Encryption System (AES) [11] and PRESENT [5]), GIFT is a cipher based on Substitution–Permutation Networks (SPNs) [4]. GIFT was proposed as an improvement to PRESENT. Despite PRESENT's simple construction of substitution and permutation operations, a particular part of the cipher that protects against differential cryptanalysis significantly increases the cost and complexity of PRESENT, as its S-Box has to satisfy branching number 3 (BN3) [4]. To overcome such drawbacks, GIFT carefully constructs the substitution and permutation blocks in conjunction, thereby reducing the requirement from BN3 to BN2 in the S-Box implementation, and hence, resulting in a lower overhead. In contrast to PRESENT, GIFT has smaller substitution blocks and uses a fewer number of rounds, achieving a very compact and high throughput design.

GIFT-64 (GIFT-128) encrypts 64 (128) bits of data with a key length of 128 bits using 28 (40) rounds. The input rounds are organized in segments of 4 bits (plaintext for the first round, otherwise intermediate states). Hence, GIFT-64 has 16 segments and GIFT-128 has 32 segments. Figure 1 shows a single round of the GIFT-64 cipher. It consists of:

*a) **SubCells**:* Each segment (4-bit) is substituted based on a non-linear function called substitution box (S-Box). The output of the S-Box is a 4-bit segment.

*b) **PermBits**:* It shuffles (i.e., reorders) the bits in all segments.

*c) **AddRoundKey**:* The two least significant bits (LSB) of each segment are XORed (i.e., exclusive-or operation) with specific bits of the key. Figure 1 shows that the two LSB bits of the first segment are XORed with key-bit 0 and key-bit 16 (see purple blocks in the Sub-Key Adding part of the figure). For the following segment, the key-bit 1 and key-bit 17 are used. This interleaved AddRoundKey process uses 32 bits of the key per round. Additionally, in this step, the most significant bit of each segment (MSB) is XORed with a specific round constant (see yellow blocks in the Constant Adding part).

*d) **UpdateKey**:* This step updates the key for the next round. First, the entire key is 32-bits circularly rotated to the right, thus replacing the used key-bits by the next 32 bits. Thereafter, the 32 used key bits (now located in the most significant part of the key) go through a local circular rotate operations, where the 2 MSB bytes are rotated by 2, and the following 2 bytes by 12, as shown in the bottom part of Fig. 1.

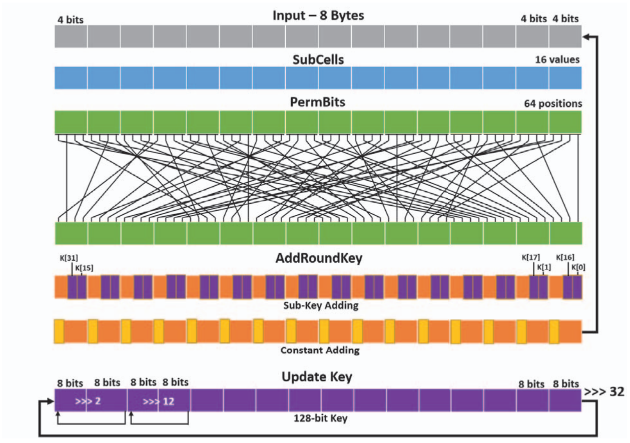

Fig. 1. Round operations of GIFT-64 cipher

III. GRINCH Attack

This section discusses the GRINCH attack. First, it describes the vulnerabilities of GIFT. Thereafter, it presents the threat model and the attack methodology.

A. GIFT Vulnerability

Similarly as in many SPN-based ciphers, GIFT includes a non-linear substitution operation (i.e., SubCells or S-Box) that substitutes each 4-bit segment with another 4-bit number. A commonly used software implementation of GIFT is based on transformation tables, where the *SubCells* operation is implemented with a lookup table whose entries are the input rounds. In such implementation, a single lookup table is used in such a way that it is accessed by each segment. Our attack focuses on the first four rounds by monitoring the key-dependant S-Box cache accesses.

The input of the S-Box (also called *index*) is the result of XORing the previous round state (round input) with the secret key. In the first round, the plaintext is used as the state and no key operation is involved. From the second round onwards, the *index* is computed from the previous state and the secret key. Therefore, when the round input and the index of the substitution table are known, it is possible to retrieve the key by calculating $Key \leftarrow Index \oplus Input$.

Fortunately, GIFT cipher uses an S-Box of 16 values, which is considered very tiny when compared with the 256 values S-Box used in the AES. As a result, the probability that an encryption uses all S-Box addresses in the first rounds is very high. Hence, an attacker that is looking for the used cache addresses after the end of the encryption process would not be able to extract useful information. Nevertheless, today's systems employ task scheduling, where tasks are pre-empted to run multiple tasks concurrently. Therefore, it is possible to access the cache while the cipher is still in its intermediate state.

GRINCH strategy is based on running multiple encryptions with different messages, each carefully crafted in order to activate the same *index* of the S-Box (in a certain segment of a certain round). The attacker can eliminate key candidates from the encryption based on the selected S-Box address until a single index remains, which the attacker subsequently can use to retrieve part of the key (see part c of Section II). Finally, the same process is repeated by targeting different segments until the full key is recovered.

B. Threat Model

IoT devices contain System-on-Chips (SoCs) that include single or multiple heterogeneous processing units, memories, peripherals, hardware accelerators and other IP hardware cores [12]. SoCs may include memory hierarchies comprising several levels of cache (e.g., L1 to L3) and DRAMs. When a cache miss occurs, data is searched throughout the cache levels and eventually looked up in the DRAM when needed.

GIFT cipher is designed to be used in such IoT devices. They use an operating system to manage and schedule multiple applications. Note that trusted cryptographic applications (e.g., GIFT cipher) share the hardware platform together with potential malicious or untrusted third-party software that could gather, process and communicate data. Taking all of this into consideration, we can assume that trusted and non-trusted applications can run on the same hardware platform, sharing resources like on-chip communication structures (e.g., bus or Networks-on-Chip), interfaces and cache memories. In this work, we define two main processes named victim and attacker. The victim process encrypts/decrypts messages using GIFT cipher. The attacker process runs external malware that manipulates the data to be encrypted and accesses the shared cache memory. In summary, the considered threat model has teh following characteristics:

- The cache used by the victim is accessible by the attacker.
- Attacker can measure the cache access time (to differentiate cache miss and hit).
- Attacker can create/manipulate plaintexts.
- Optionally, the attacker can flush the cache.

C. *Methodology*

The GRINCH attack focuses on identifying the index of one single S-Box access (i.e., one segment) of the second round. Note that in the second round the key is used for the first time. If such an index is identified, two bits of the key can be retrieved (see Figure 1). However, an attacker only can control the plaintext and not the state of the second round. Hence, to control a single access of the S-Box in the second round, the attacker has to carefully select four segments of the plaintext (i.e., input of first round). These four input segments determine the value of one segment of the second round due to the S-Box and permutation operation of the first round. As the key is unknown to the attacker, it is not possible to know upfront which S-Box index will be used. To solve that issue, an attacker can perform many encryptions while crafting the input in such a way that the targeted segment is kept active and stimulates the same target key bits (i.e., the two bits involved in the AddRoundKey). These plaintext manipulations create a condition where only one index will be accessed in the cache during all performed encryptions. When more encryptions are performed, more key candidates can be eliminated until a single candidate remains. In such cases, it is possible to reverse engineer the two bits of the key using the index and the state of the involved segment. This process is repeated 15 times for the other segments to recover the complete 32 bits of the key.

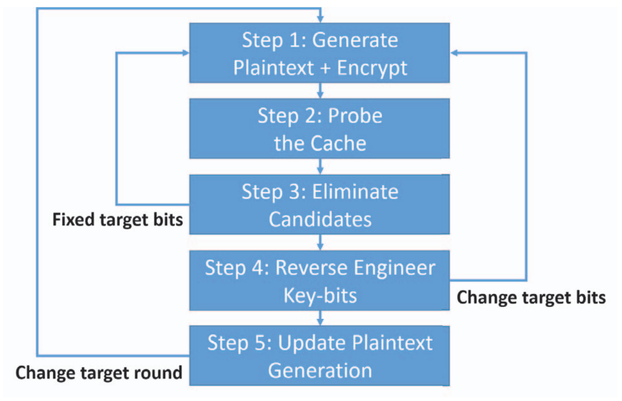
Step 1: Generate Plaintext + Encrypt
Step 2: Probe the Cache
Step 3: Eliminate Candidates
Step 4: Reverse Engineer Key-bits
Step 5: Update Plaintext Generation
Fixed target bits
Change target bits
Change target round

Fig. 2. GRINCH attack methodology

Once the attacker knows 32 bits of the key, the process can be repeated in order to attack the next round by computing the input state of the following round. Note that the key is shifted 32 bits to the right after each round (see also Figure 1). By changing the plaintext to match the targeted segments in the third round again 32 key bits can be recovered. After applying the same trick four times, the entire 128-bit key can be retrieved.

The methodology of GRINCH consists of five steps as shown in Figure 2 and described next.

Step 1 - Generate Plaintext + Encrypt: The goal is of the Step 1 is to craft the plaintext so to force the same S-Box accesses for certain key-bits. The methodology starts by defining the target key-bits, as shown in Algorithm 1. The first part of the algorithm identifies the offset of the key-bits K_i, K_j in the AddRoundKey (e.g., the offset of key bit 0 is 0 and the offset of key bit 16 is one (see AddRoundKey in Fig. 1. This is realized through the $StatusBitXorKey(K)$ function (lines 2 and 3). Thereafter, these bits are inversely permuted (lines 4 and 5) to obtain their indexes (bit_A) and (bit_B) before the PermBits function, which is equal to the output of the S-Box layer. For the attack to succeed, the bits bit_A and bit_B should always keep the same value so that the target key bits of the S-Box in the next round remain unchanged. In this attack we set these bits to 1. Hence, the inputs of the S-Boxes for these two bits must be careful chose and always lead to a 1 at the output (loop on lines 5-12). As a result, for each bit, a list of valid inputs and that will always force the same XOR operation with the target key-bits K_i, K_j is generated. Thereafter, all the plaintexts are generated based on these lists, as in Algorithm 2. For each plaintext segment, a random value is applied when it is not part of a segment where bit_A or bit_B is located, and an arbitrary index from the list is used when it is in order to make sure that the value at bit_A and bit_B is always 1 after the S-Box. At the end of this procedure, a single plaintext is generated and used for the attack.

Step 2 - Probe the Cache: To obtain the information of accessed addresses of the S-Box, the attacker can

Algorithm 1 Set target bits algorithm

```
procedure SET_TARGET_BITS($K_i, K_j$)
    $a \leftarrow StatusBitXorKey(K_i)$
    $b \leftarrow StatusBitXorKey(K_j)$
    $bit_A = Inv\_Permutation(a)$
    $bit_B = Inv\_Permutation(b)$
    for each element $X$ inside SBOX do
        if $X[bit_A] == 1$ then
            $list_A.append(Inv\_SBOX[X])$
        end if
        if $X[bit_B] == X$ then
            $list_B.append(Inv\_SBOX[X])$
        end if
    end for
    return $List_A, List_B$
end procedure
```

Algorithm 2 Plaintext generation algorithm

```
procedure GENERATE($List_A, List_B$)
    for $i \leftarrow 0$; $i < 16$; inc $i$ do
        if $i == segment(bit_A)$ then
            $Plaintext[i] \leftarrow list_A[random()]$
        else
            if $i == segment(bit_B)$ then
                $Plaintext[i] \leftarrow list_B[random()]$
            else
                $Plaintext[i] \leftarrow random()$
            end if
        end if
    end for
    return $Plaintext$
end procedure
```

perform classical cache attacks such as Prime+Probe and Flush+Reload. The former method accesses an address of the cache that evicts the victim's information. During or after the victim's operation, the attacker accesses the same address and observe if it has been used. If the victim used that address, the attacker experiences a cache miss. Flush+Reload uses the same principle, but the first part is performed with a specific command to erase (parts of) the cache (i.e., flush operation). For the GRINCH attack, the Flush+Reload method is better choice. As a flush operation is faster, the attacker can probe the cache earlier. The earlier the attacker is able to probe the cache, the easier it is to monitor individual rounds.

Step 3 - Eliminate Candidates: The goal of the Step 3 is to identify the unique index that is accessed in many different encryptions. Since the target bits involved in the add round key are fixed, one of the S-Box indexes will be present in all performed encryptions. After some iterations, it is possible to identify the index related to the target key-bits by eliminating the indexes that do not appear in all cases.

Step 4 - Reverse Engineer Key-Bits: Since the attack methodology always forces the target key-bits to be XOR-ed with ones (i.e., bit_A=bit_B=1), the attacker can simply reverse engineer these key-bits by inverting the related bits of the obtained index. This can be expressed by $Key[i] \leftarrow Index[a] \oplus 1$ and $Key[j] \leftarrow Index[b] \oplus 1$; or $Key[i] \leftarrow \neg Index[a]$ and $Key[j] \leftarrow \neg Index[b]$.

Step 5 - Update Plaintext Generation: After attacking the first round, the attacker needs to repeat the process targeting the following rounds. The complete key can be retrieved after four iterations. However, each time the target round changes, the plaintext generation algorithm has to be updated. The revealed 32 key-bits (from Step 4) need to be used to generate new plaintexts that can be used to attack the next round, i.e., the attacker can compute the intermediate round values to generate the plaintexts that force the values on the target bits in the round under attack.

D. Challenges

By analyzing the GRINCH attack methodology, the Step 2 (Probe the Cache) might be challenging due to the required timing precision in accesses the cache during the victim's operation, and due to the configuration of the cache memory. Some strategies to overcome such issues are discussed next.

Cache Probing Precision: Depending on the system configuration, the task to access the cache during the first rounds of GIFT cipher might be not feasible. Nevertheless, the attacker can still try other approaches. An interesting option is to apply power analysis to observe the cache accesses. The work in [10] has demonstrated that the power consumption may clearly reveal when cache misses and hits happens.

Cache Configuration: The configuration of the cache memory affects the attack. An important parameter is the size of the cache line. Since the GIFT S-Box only contains 16 bytes, a cache line could contain multiple elements. As a result, the accessed index is obfuscated. Nevertheless, the attack is still possible as long as the whole S-Box fits in a single cache line. Note that only the two least significant bits of the index are not controlled by the attacker, as they depend on the key-bits. This means that independently of the cache line size, the maximum number of candidates is 4. As a result of this, the attacker can continue to the next round and assume all possibilities.

IV. Experimental Results

A. Setup

The GIFT software implementation was obtained from the public repository in [13]. It has the *SubCells* and *PermBits* operations implemented through look-up tables. GIFT was deployed into two SoC platforms: i) *single processor SoC*, comprised by a processor, a shared cache L1, I/O peripherals (i.e., UART serial) and a bus as communication structure; and ii) *multi-processor SoC (MPSoC)*, a tile-based structure comprising seven processors, a shared cache L1 and I/O peripherals. All these components are interconnected through a mesh-based Network-on-chip (NoC) that uses XY deterministic routing. Both SoC platforms use the RISCY core as the processing unit. RISCY is a RISC-V architecture from the Pulpino project [14]. The shared L1 cache used in both platforms is a 16-way set-associative memory with 1024 cache lines where each cache line contains in the default case a single word consisting of 8 bits. GRINCH was executed on both platforms while performing encryptions with GIFT. Three different experiments are performed in this work:

1) **Attack Effort versus Attack Efficiency:** This experiment analyzes the amount of encryptions that are required by GRINCH to perform a full recovery of the GIFT key. This amount depends on the cache probing efficiency. We evaluate the impact of different probing moments and the impact of a flush operation during the attack. This scenario uses the cache line size of 1 word.
2) **Attack Effort versus Cache Configurations:** This experiment evaluates the impact of the cache line size on the attack. The effort is measured in terms of the amount of encryptions required to perform a full key recovery. Cache line sizes of 1, 2, 4 and 8 words per cache line are analyzed.
3) **Practical Attack Analysis:** This experiment runs the attack on the two hardware platforms (i.e., single processor SoC and MPSoC) on an FPGA. This evaluation provides practical attack efficiency results for different clock frequencies. For the single processor SoC, a task scheduler was used to emulate the RTOS operating system [15]. RTOS is a popular OS for embedded and IoT systems, which uses a *quantum time* (i.e., the timing slot that a process gets assigned to the processor) of 10 milliseconds.

For the first two experiments, RTL simulations were used to collect clean data. The third experiment, the attack was executed in an FPGA platform. The target FPGA is the Genesys 2 board which contains a Xilinx Kintex 7 device [16]. In all experiments, the pre/post-processing analysis (i.e., plaintext generation and reverse-engineering of the keys) were performed in Python.

B. Results

The results of the three set of experiments are presented next.

1) Attack Effort versus Attack Efficiency: Figure 3 shows the required amount of encryptions to perform a full key recovery of the GIFT cipher when the first round is attacked; hence, only 32 bits of the key. The horizontal axis shows the moment in time (in rounds) in which the attacker can probe the cache status. The earlier this moment, the better the attacker's efficiency. In case the attacker is able to probe the cache in the first round, approximately 100 encryptions are needed to recover 32-bit keys (as can be seen in the figure). To recover the whole key, 400 encryptions would be required. The later you probe the cache, the more contaminated the results are. The efficiency of the attack depends on the amount of noise (e.g., multiple processes disputing the processor) and the operating system configuration (i.e., defined quantum time). Considering the first round attack (i.e., the first round can be probed), only the cache sets accesses of the second round contain useful information for the attacker. The cache sets accessed in the subsequent rounds are additional sources of noise, which is reflected in the results as extra effort. Additionally, as the S-Box lookup table is small, late cache probing results in that most likely all S-Box content is present in the cache, which makes it extremely hard for the attacker to eliminate candidates (see exponential increase in complexity vs cache probing time in the figure). Moreover, the experiment also evaluated the effect of the *flush* operation. The absence of the flush operation increases the attack effort since it adds noise (includes "dirty" accesses from the first round) to the information gathered by the attacker. Note that the first round depends only on the input and it is not useful for the attacker. Hence, it only increases the effort to succeed. Our experiment does not evaluate the efficiency of rounds higher than 10, as the amount of encryption required for retrieving the key at round 10 is already too high to be considered practical in an IoT environment. Results show that the attack is practical if the adversary probes the cache before the fifth round when the *flush* operation is used, and before the forth round, otherwise.

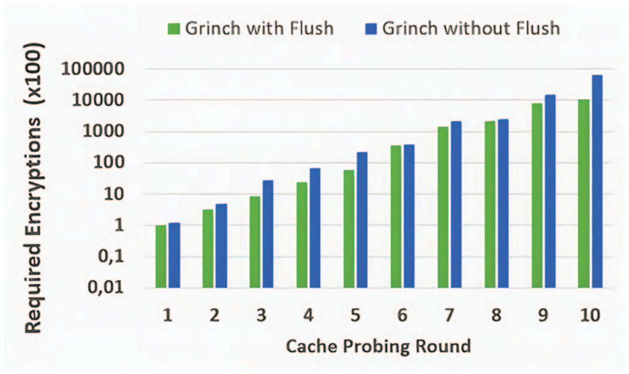

Fig. 3. Required encryptions to break 1st GIFT Round.

2) Attack Effort versus Cache Configurations: Table I shows the attack efficiency for different cache configurations. Results show that the increase of the cache line size decreases significantly the efficiency of the attack; this is measured by the amount of encryptions required

TABLE I
REQUIRED ENCRYPTIONS TO ATTACK THE FIRST ROUND.

	Attack Efficiency - Probing Round				
Cache Line Size	**1**	**2**	**3**	**4**	**5**
1 Word	96	312	840	2,448	5,864
2 Words	136	1112	11440	188536	>1M
4 Words	136	123848	>1M	>1M	>1M
8 Words	113000	>1M	>1M	>1M	>1M

TABLE II
ATTACK EFFICIENCY (ROUNDS) OF PERFORMED ATTACKS.

	Clock Frequency		
Platform	**10 MHz**	**25 MHz**	**50 MHz**
Single-processing SoC	2	4	8
Multi-processing SoC	1	1	1

to perform a full key recovery. Note that the experiments with more than 1 million encryptions were drop-out before finishing as they are not considered practical. However, the attack is still practical when the attacker probes the cache within the first or second rounds. Therefore, the success of GRINCH depends both on the precision and ability of the attacker to probe the cache in the correct moment of time and on the cache memory configuration.

3) Practical Attack Analysis: Table II shows the practical implementation of GRINCH. The results show the round number which was successfully probed by the GRINCH. For the single processor SoC, the GRINCH was able to probe the cache during the second round when operating at the lowest frequency (10 MHz). This result is interesting as many IoT devices are expected to operate at this frequency. In contrast, when the SoC is operating at higher frequencies, the GRINCH was only able to probe the cache at the fourth and eighth rounds for 25 MHz and 50 MHz, respectively. For the multi-processor SoC, the GRINCH was very efficient and probed the cache during the first round. Since the malware runs on its own dedicated processor, the attacker can write content to the shared cache as desired. As observed during experiments, in the fastest scenario (i.e., encryption running at 50 MHz), the time between different rounds was about 1.2 milliseconds. This time is much higher than accessing the shared memory on a different tile, which took approximately 400 nanoseconds consisting of the processor delay, Network-on-Chip latency and cache memory response time.

C. Potential Countermeasures

From the analysis of the GIFT cipher and the observed results it is possible to create two protection strategies. The first countermeasure is to eliminate the look-up table vulnerability. For the S-Box, the proposed method is to set the cache line to 8 bytes and reshape the S-Box from 16 rows of 4 bits to 8 rows of 8 bits. As an overhead, you have to select the right 4 bits at the output. The second countermeasure is to modify the *UpdateKey* operation of the GIFT cipher. Currently, the first four rounds uses directly the bits of the key, which makes GRINCH attack possible. If the UpdateKey of the first round prepare the sub-key to be used in the next round by applying some computation with bits that were not used yet, the key retrieval would not be possible. This requires however cryptanalysis that goes beyond the scope of this paper.

V. CONCLUSION

In this work we proposed GRINCH, the first cache attack on GIFT. GRINCH is based on a customized access-driven cache attack that exploits the access pattern to the GIFT S-Box. Despite the use of small lookup tables, we show that by exploiting microarchitectural characteristics of the SoC, secret information of GIFT is leaked. As a result, the full secret key can be retrieved by an adversary with less than 400 encryptions. We explored the impact of the cache configuration in the efficiency of the attack and show its practical realization in SoC and MPSoC. As future work, we aim to implement new protection mechanisms and further explore the effect of the memory hierarchy on the effectiveness of the attack.

REFERENCES

[1] Gartner, Inc, "Iot penetration statistics," 2018, https://www.gartner.com/.
[2] NIST, "Lightweight cryptography," 2020, https://csrc.nist.gov/projects/lightweight-cryptography.
[3] NIST, "Lightweight cryptography round 2 candidates," 2020, https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates.
[4] S. Banik *et al.*, "Gift: A small present," Cryptology ePrint Archive, Report 2017/622, 2017, https://eprint.iacr.org/2017/622.
[5] A. Bogdanov *et al.*, "Present: An ultra-lightweight block cipher," in *CHES*, P. Paillier and I. Verbauwhede, Eds., 2007.
[6] J. Zhang *et al.*, "Power analysis attack on a lightweight block cipher gift," in *International Conference on Computer Engineering and Networks*, 2020.
[7] S. Patranabis *et al.*, "Scadfa: Combined sca+dfa attacks on block ciphers with practical validations," *IEEE Transactions on Computers*, 2019.
[8] D. J. Bernstein, "Cache timing attacks on aes," Available at: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, April 2005, accessed at 2016-12-31.
[9] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes." Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.
[10] O. Acıiçmez and Çetin K. Koç, "Trace-driven Cache Attacks on AES," in *International Conference on Information and Communications Security*, 2006.
[11] J. Daemen and V. Rijmen, *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.
[12] A. Vajda, *Multi-core and Many-core Processor Architectures.*, 2011.
[13] GIFT Block CIpher Group, "GITHUB Repository of GIFT cipher implementation," Available at: https://github.com/giftcipher/gift, note = Accessed at 2020-09-21,.
[14] M. Gautschi *et al.*, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE TVLSI*, 2017.
[15] C. V. Penumuchu, *Simple Real-Time Operating System: A Kernel Inside View for a Beginner*. Trafford Publishing, 2007.
[16] Xilinx, "kintex-7," Online, 2020, https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html.