An optimized Bitsliced Masked Adder for ARM Thumb-2 Controllers

Enrico Pozzobon[a,b], Sebastian Renner[a], Jürgen Mottok[a], Václav Matoušek[b]
[a] Laboratory for Safe and Secure Systems, OTH Regensburg, Germany
[b] Department of Computer Science and Engineering, University of West Bohemia, Czech Republic
{enrico.pozzobon, sebastian1.renner, juergen.mottok}@othr.de, matousek@kiv.zcu.cz

2022 International Conference on Applied Electronics (AE) | 978-1-6654-9482-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/AE54730.2022.9919884

***Abstract*—The modular addition is used as a non-linear operation in ARX ciphers because it achieves the requirement of introducing non-linearity in a cryptographic primitive while only taking one clock cycle to execute on most modern architectures. This makes ARX ciphers especially fast in software implementations, but comes at the cost of making it harder to protect against side-channel information leakages using Boolean masking: the best known 2-shares masked adder for ARM Thumb micro-controllers takes 83 instructions to add two 32-bit numbers together. Our approach is to operate in bitsliced mode, performing 32 additions in parallel on a 32-bit microcontroller. We show that, even after taking into account the cost of bitslicing before and after the encryption, it is possible to achieve a higher throughput on the tested ciphers (CRAX and ChaCha20) when operating in bitsliced mode. Furthermore, we prove that no first-order information leakage is happening in either simulated power traces and power traces acquired from real hardware, after sufficient countermeasures are put into place to guard against pipeline leakages.**

Index Terms*—Boolean masking, modular addition, side-channel, bitsliced, ARM Thumb

I. Introduction

The modular addition is used as a cryptographic primitive in add–rotate–XOR (ARX) ciphers, because it is a non-linear operation with some degree of diffusion that can be achieved (for some modulo) in a single clock cycle on most processor architectures. For example, the addition modulo 4294967296 (or 32-bit modular addition) can be executed with a single instruction on ARM cores commonly found in cheap Microcontroller units (MCUs). Thanks to the properties of the modular addition, ARX ciphers such as Chacha20 can perform encryption and decryption operations efficiently without hardware acceleration.

The disadvantage of using modular addition as a cryptographic primitive is that it becomes harder to protect the cipher against side-channel attacks using Boolean masking. Jungk et al. [1] explains these challenges, and contributes the fastest known 2-share masked software threshold implementation of the modular adder for the ARM Thumb-2 architecture. In particular, the adder proposed by Jungk et al. [1] takes 83 instructions to perform the 32-bit modular addition of two input numbers, each provided as 2 shares, without causing first order side channel information leakage through power traces simulated through Micro-Architectural Power Simulator (MAPS).

A. Boolean Masking

Boolean masking protects against side channel attacks up to the $(n-1)$-th order by splitting the input and the output of the cryptographic operation in n shares. In other words, probing a side-channel (such as power utilization) at up to $n-1$ points in the circuit and in time does not give enough information to derive secret data (such as the key) from a masked cryptographic implementation.

The input x is masked by performing an exclusive-or (XOR) operation with $(n-1)$ random numbers $(r_1, ..., r_{n-1})$, then the masked input will be composed of $n+1$ shares: $(r_1, ..., r_{n-1}, x \oplus r_1 \oplus ... \oplus r_{n-1})$, noting that the original input information is still obtainable by XORing together all the n shares. The encryption and decryption algorithms are then implemented to operate on the n shares with care to not leak information by ensuring that the power utilization of each operation (estimated for example by the Hamming weight of each value being written in a core register) is not correlated to any secret information. In particular, [2] and [3] give tools on how to prevent that the construction of a Boolean masked algorithm leads to information leakage though side-channels. Finally, the output of the cryptographic algorithm can be obtained by XORing the n output shares.

B. Bitslicing

Bitslicing has been used in cryptography to protect against cache timing attacks on S-Boxes on algorithms such as AES, as well as to implement Boolean Masking in other ciphers [4].

In a bitsliced realization of an algorithm, the variables are "sliced" along the bits so that a n-bit variable will be sliced in n parts that will be stored in n different registers. Since each register is only be able to contain one bit of each variable, the remaining bits of the registers are used to execute multiple parallel copies of the algorithm, such as encrypting multiple blocks of plaintext in parallel.

C. Search of Boolean maskings

Gross et al. [5] and Biryukov et al. [6] show how exhaustive search can be used to find the optimal first-order Boolean masking of the AND and OR operations. However, the time for the exhaustive search approach grows exponentially with the number of input variables, the required shares, and available instructions on the desired architecture.

An upper bound can be estimated on the size of the search space by evaluating the maximum number of different columns that can be written within the truth table where the inputs are the input shares, which is $2^{(2^{(n \cdot m)})}$ where m is the number of inputs and n is the number of shares. As each expression which combines the inputs will result in one of these $2^{(2^{(n \cdot m)})}$ columns, searching a Boolean masking within corresponds to searching a group of n of these columns that when XORed together give the desired unmasked output, as well as the list of operations that leads from the input columns to this output.

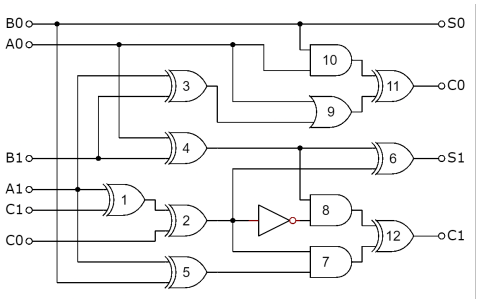

Fig. 1. First-order leakage-free shared full adder network implemented using 12 ARM Thumb-2 instructions. The `NOT` gate is not counted since ARM has the BIC instruction that can perform the `AND` and `NOT` operations in one instruction.

When the problem is the `AND` operation with 2 inputs and 2 shares, the total number of possible combinations of the shared inputs is 2^{16} and the algorithm described in [6] finds an optimal implementation with just 6 ARM instructions in 11539239 iterations. When the problem is a Full Adder with 3 inputs (A, B and Carry) and 2 shares, the total number of possible combinations of the shared inputs is 2^{64}, and the algorithm does not find a solution in several days of execution on a modern personal computer.

II. Contribution

Our contribution is a masked full adder with no first-order leakages, which takes three inputs (A, B and C^i), and returns two outputs (S and C^o), using the bitwise operations found on any inexpensive ARM Thumb-2 core. This full adder can be iterated across all the bits of a bitsliced variable as a simple ripple carry adder, summing 32 variables in parallel on the target architecture. The definition of the output of the full adder is:

$$S = A \oplus B \oplus C^i \tag{1}$$

$$C^o = ((A \oplus B) \wedge C^i) \vee (A \wedge B) \tag{2}$$

The first and most obvious way to implement the full adder would be to take the `SecAnd` and `SecOr` gadgets from [6] and simply using them to realize the circuit in formulas 1 and 2. This would lead to a full adder with no first-order leakage realized in 22 instructions (6 instructions for each `AND` and `OR` gate, 2 instructions for each `XOR` gate, with one `XOR` gate shared amongst the two formulas). Therefore 22 is the upper bound on the number of instructions necessary to implement the masked full adder.

Each of the two outputs can be considered as a separate problem, but the sum output S is simple to realize because it is only made of `XOR` operations and any partition of the input shares in two groups is a suitable masking of S as long as either group is not empty and not equal to an input variable. Meanwhile, output C^o is harder to realize since it is a non-linear function of the inputs.

A. Search algorithm

Our search algorithm is an optimization of the one used in [6], but caches a truth table column for each node to be able to instantly check for leakage though the technique explained in [3] where all the rows of a truth table are grouped by their secret (unmasked) input and summed group-wise to easily compare if the Hamming weight is uniform across different secret inputs.

A first basic unguided version of our exhaustive search algorithm tries to create a massive truth table where all the possible combinations of the inputs fulfilling the conditions from [3] are stored.

Each column is stored as a tuple (n, f, x, y), where n is a 64-bit integer encoding the binary values of the truth table (note that 64=2^6 and 6 is the number of binary inputs), f is a binary operation selected from the pool of available bitwise operations available on the target architecture, and x, y are the indices of the two columns selected as operands for the operation such that $n = f(n_x, n_y)$ if n_x and n_y are the value of n for the columns at positions x and y. This way it will be possible to calculate what was the sequence of operations necessary for reaching each column in the truth table.

At the start of the algorithm, the truth table will only contain 6 columns representing the identity function of the 6 inputs (a_0, a_1, b_0, b_1, c_0, c_1). At each iteration, the truth table is extended by adding all the columns which can be obtained by combining any two present columns with any of the available operations, excluding those columns which don't fulfill the condition of equal Hamming weights between unmasked groups from [3]. If a column is found which has identical binary values to one that was already found, it is not added to the table, since by construction the previous one had a lower cost in instructions (logic gates). Whenever the combination of two columns with a `XOR` operation results in one of the searched outputs (S or C^o), that output is removed from the list of searched outputs, and the algorithm terminates when the searched outputs list is empty.

The algorithm described until now is already much faster than the one used in [6], but it is still takes an exceedingly large number of iterations to find a full adder and runs out of memory within a few hours on a modern personal computer.

To further optimize the algorithm, we remember that a masked S output can be obtained by dividing the 6 inputs in 2 groups and then XORing the inputs within each group. Since these `XOR` operations are going to be necessary regardless of how the output C^o will be computed, we first perform an exhaustive search with just the `XOR` operation to find all the 114 possible columns in the truth table that can be generated with just this instruction. We call this first set of columns obtainable only through linear operation (`XOR`) the "Linear Expansion Layer". Multiple ways to compute S are found in this "Linear Expansion Layer", but the decision on which one to use will be made after C_o is found to reuse as many nodes as possible between the two necessary outputs.

Then, we use these columns as a start for a single iteration of the algorithm, but this time all the bitwise operations allowed by the target instruction set will be iterated (for ARM Thumb-2, these are `EOR`, `AND`, `ORR`, `BIC`, `ORN`). The set of columns obtained though this iteration will be called the "Non-Linear Layer" of the logic net.

At this point, a non-uniform 4-share C^o output can already be found between the columns of the non-linear layer. To collapse the shares into an uniform 2-share output, we just run the search algorithm for one more iteration using only the `XOR` operation as was done for the linear expansion layer, generating a new set of columns called "Share Collapsing Layer", which finally contains a 2-share C^o output.

Now, starting from the found output shares of C^o, we

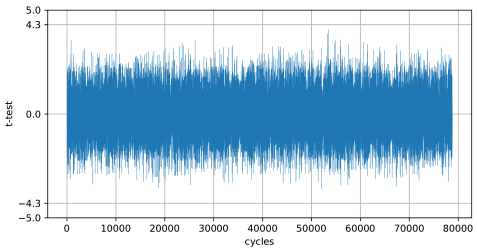

Fig. 2. T-Test performed on 100000 traces acquired from CRAX encryptions simulated using the MAPS software. As the t-test line never exceeds the $[-5, 5]$ interval, it can be assumed that no first-order information leakage is taking place.

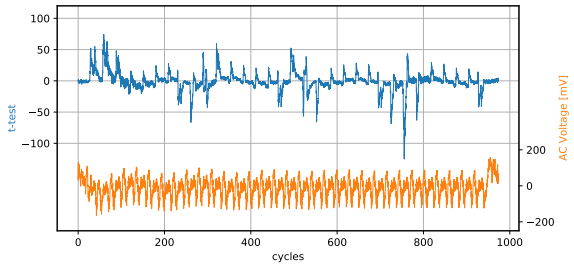

Fig. 3. T-Test performed on 10000 traces acquired from an STM32F103C8T6 MCU, leakage is visible and is shown by the spikes of the t-test line surpassing the threshold of 5.

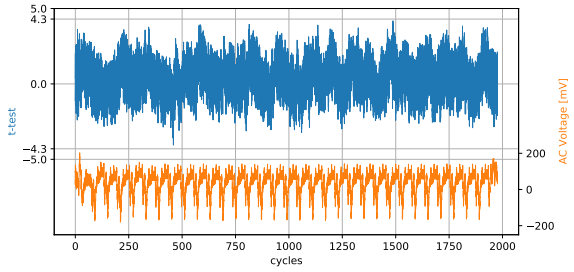

Fig. 4. T-Test performed on 10000 traces acquired from an STM32F103C8T6 MCU using the hardened bitsliced CRAX algorithm to protect against pipeline and other micro-architectural leakages.

can explore the truth table backwards using the values x and y from the stored tuples described earlier, and finally construct the first-order-leakage-free full adder network. The found adder is made out of 12 instructions, 11 of which are necessary to compute C^o. The resulting network is shown in figure 1.

Algorithm 1 2-shares masked full adder

Require: $A = a_0 \oplus a_1; B = b_0 \oplus b_1; C^i = c_0 \oplus c_1$
Ensure: $S = s_0 \oplus s_1; C^o = c_0 \oplus c_1$

1: $t_1 \leftarrow a_1 \oplus c_1$
2: $t_2 \leftarrow c_0 \oplus t_1$
3: $t_3 \leftarrow a_1 \oplus b_1$
4: $t_4 \leftarrow a_0 \oplus b_1$
5: $t_5 \leftarrow a_1 \oplus b_0$
6: $t_6 \leftarrow t_4 \oplus t_2$
7: $t_7 \leftarrow t_5 \wedge t_2$
8: $t_8 \leftarrow t_4 \wedge \neg t_2$
9: $t_9 \leftarrow t_3 \vee a_0$
10: $t_{10} \leftarrow a_0 \wedge b_0$
11: $t_{11} \leftarrow t_9 \oplus t_{10}$
12: $t_{12} \leftarrow t_8 \oplus t_7$
13: $s_0 = b_0; s_1 = t_6$
14: $c_0 = t_{11}; c_1 = t_{12}$

III. Evaluation

To prove that the computed logic network does not leak cryptographic material, a bitsliced realization of the CRAX and ChaCha20 encryption algorithms was realized and tested both on a simulator (MAPS) and on a real hardware MCU (STM32F103C8T6).

32 of the shown full adder networks are combined in sequence to form the 32-bit adder that is necessary for the tested algorithms. On the first full adder in the sequence, the C^i input must be initialized with a random bit in each slice to preserve the uniformity property of the input; this is expected and consistent with the randomness requirements obtained by [1].

A. Leakage evaluation

Figure 2 shows a t-test obtained from the power traces acquired from the MAPS simulation, with pipeline leakage simulation disabled, as this was the methodology used to test for leakage by [1]. When implementing the described logic network in a software implementation, one must pay attention to prevent compiler optimizations from optimizing the masking away, as well as to avoid accidentally leaking information through register reuse. This is achieved by programming the masked cryptographic primitives directly in assembly and using different registers for operating on shares of the same secret variable.

Pipeline leakages as well as other internal register leakages are expected to be different for every different MCU manufacturer, so these hardware-specific results are hard to compare across different publications. Figure 3 shows that the presented algorithm still exhibits leakages when tested on a real STM32F103C8T6 MCU, thus a hardened version of the adder was developed specifically to fix leakages caused by the pipeline registers and the Memory Data Register (MDR).

Three different types of leakages were exhibited on the real hardware and required hardening.

- The guidelines from [7] were used to avoid these leakages through the A and B registers used to cache the operands of the Arithmetic logic unit (ALU) by the pipeline.
- Variables being cached on the stack memory caused leakage through the reuse of MDR which was solved by inserting dummy LDR and STR instructions to clear it.
- Some NOP instructions were added before branch instructions in loops to prevent speculative loads of some registers in the pipeline which caused leakages.

This new hardened version takes almost double the number of cycles to perform an encryption, but does not leak even on real hardware (see figure 4).

B. Performance Evaluation

To ensure that the presented results are comparable with the previous research, the adder was tested by implementing the ChaCha20 encryption algorithm without hardening against pipeline leakages, as done in [1].

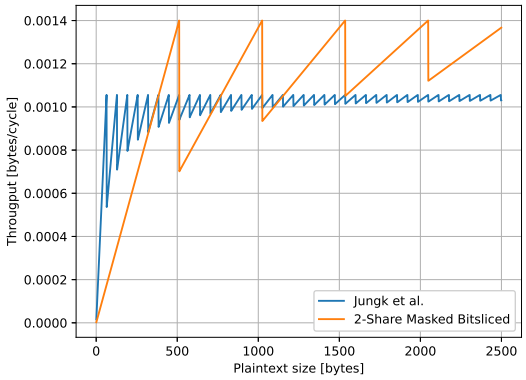

Fig. 5. Throughput of the Chacha20 algorithm protected through either the Jungk et al. [1] masked adder or through the bitsliced adder presented in this paper, for different message sizes. As expected, the bitsliced algorithm is more performant when the message size is larger.

TABLE I
CODE SIZES, MEMORY UTILIZATION AND THROUGHPUT OF THE TESTED IMPLEMENTATIONS OF CHACHA20.

Implementation	Code	Memory	Cycles per byte
Unprotected Optimized	3174	228	160.8
Unprotected Jungk et al.	488	56	215.7
Masked Jungk et al.	1212	316	947.2
Proposed Bitsliced Masked	1024	2260	701.5

Figure 5 shows that the presented algorithm is only beneficial when the message size is larger than 1152 B, due to the introduction of the cost of bitslicing.

Table I further shows another disadvantage of this bitsliced approach, highlighting that the memory usage is much larger than the non-bitsliced approaches. This is explained by observing that in order to efficiently exploit the parallelism of bitslicing, it is necessary to keep in memory multiple blocks of the cipher at the same time (in the case of ChaCha20, 8 blocks are processed in parallel on a 32-bit ARM Thumb-2 processor).

Finally, figure 6 shows a comparison of the number of cycles necessary to encrypt a 512 bytes payload with both CRAX and ChaCha20 using different implementations, highlighting the overhead introduced by the bitslicing.

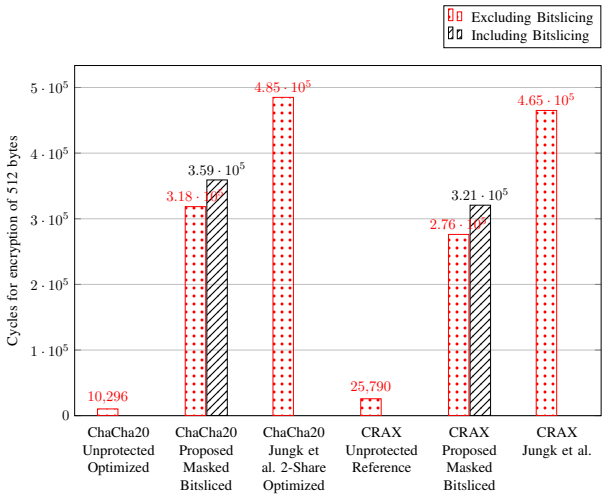

Fig. 6. Benchmark results for software implementations of the ChaCha20 and CRAX encryption algorithms using different adders. For the proposed bitsliced masked implementations, the number of cycles is presented both including and excluding the bitslicing operation.

IV. CONCLUSION

We showed how it is possible to construct an optimized 2-shares Boolean masked full adder using 12 instructions instead of 22 by using a modified version of the algorithm presented by [6].

In optimal situations (plaintexts larger than 1200 B or whose size is a multiple of 512 B) the proposed full adder allows for implementing the ChaCha20 and CRAX encryption algorithms up to 26% faster than the best known 2-shared masked adder [1] on inexpensive ARM Thumb-2 microcontrollers.

We also highlighted two big weaknesses of this approach: for small payloads, when the parallelism of bitslicing isn't exploited, the proposed algorithm is 24 times slower than the best known, and in every situation it uses 7 times more memory on the stack, which could be a problem in low power controllers.

While ARX ciphers have been historically difficult to protect efficiently against side-channel attacks using Boolean masking in software, our contribution helps reduce the number of cycles necessary for an encryption operation, which are especially precious in low power embedded microcontrollers such as the one we performed our tests on.

REFERENCES

[1] B. Jungk, R. Petri, and M. Stöttinger, "Efficient side-channel protections of ARX ciphers," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 3, pp. 627–653, Aug. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7289

[2] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security*, P. Ning, S. Qing, and N. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 529–545.

[3] H. Groß, K. Stoffelen, L. D. Meyer, M. Krenn, and S. Mangard, "First-order masking with only two random bits," in *Proceedings of ACM Workshop on Theory of Implementation Security Workshop, TIS@CCS 2019, London, UK, November 11, 2019*, B. Bilgin, S. Petkova-Nikova, and V. Rijmen, Eds. ACM, 2019, pp. 10–23. [Online]. Available: https://doi.org/10.1145/3338467.3358950

[4] S. Belaïd, P.-E. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff, "Tornado: Automatic generation of probing-secure masked bitsliced implementations," Cryptology ePrint Archive, Report 2020/506, 2020, https://ia.cr/2020/506.

[5] H. Groß, "Sharing is caring - on the protection of arithmetic logic units against passive physical attacks," in *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015, New York, NY, USA, June 23-24, 2015, Revised Selected Papers*, S. Mangard and P. Schaumont, Eds., vol. 9440. Springer, 2015, pp. 68–84. [Online]. Available: https://doi.org/10.1007/978-3-319-24837-0_5

[6] A. Biryukov, D. Dinu, Y. L. Corre, and A. Udovenko, "Optimal first-order boolean masking for embedded IoT devices," in *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, T. Eisenbarth and Y. Teglia, Eds., vol. 10728. Springer, 2017, pp. 22–41. [Online]. Available: https://doi.org/10.1007/978-3-319-75208-2_2

[7] Y. L. Corre, J. Großschädl, and D. Dinu, "Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors," in *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, J. Fan and B. Gierlichs, Eds., vol. 10815. Springer, 2018, pp. 82–98. [Online]. Available: https://doi.org/10.1007/978-3-319-89641-0_5