POLITECNICO
MILANO 1863

Definizione di una libreria per la sintesi ad alto livello di estensioni di sicurezza

Author: René Bwanika *Matricola 908696*

Supervisor: Prof. Christian Pilato

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

2021

Contents

1 Introduction

High-level synthesis has become a key component in hardware acceleration of applications since it can significantly reduce complexity of hardware component design by raising the abstraction to programming languages like C and C++. Thanks to HLS, non-hardware engineers can quickly create prototype while hardware engineers can build hardware design quicker for emerging fields such as artificial intelligence and edge computing. In the recent years, security and privacy concern have become more relevant especially in data driven applications. In this work, we investigate how to abstract security properties using HLS to integrate them in accelerator design. Previous work has focused on algorithm-level code obfuscation[1]. In this one, we focus on cryptography algorithms masking, showing how software level extension can be used to secure hardware solutions while keeping implementation details transparent to the designer.

2 Background

Side-channel leakage is a vulnerability of secure hardware and software implementations, caused by the physical effects of computing with secret variables. Modern implementations of cryptography algorithms must have side-channel countermeasures to prevent a malicious adversary from analyzing these leakages and from uncovering the secret variables that caused it. A **popular countermeasure is the masking of side-channel leakage** since it allows an implementation independent basis of correctness[2]. In a masking side-channel countermeasure, each secret variable is split into (at least) two randomly chosen shares. The side-channel leakage of a separate share does not reveal the secret variable because the random masks are secret, and in theory this guarantees that a correctly masked computation is secure against side-channel analysis. In our work we focus side-channel masking implementations of Advanced Encryption Standard (AES).

2.1 Advanced Encryption Standard

The block cipher AES (Advanced Encryption Standard) is a worldwide standard and one of the most popular cryptographic primitives. Designed in 1997, AES has survived numerous cryptanalytic efforts. AES is based on a design principle known as a substitution–permutation network[3]. AES is a variant of Rijndael, with a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. AES works on a 4×4 column-major order array of bytes, named the state. The majority of AES calculations are done in a particular finite field. For instance, it represents 16 bytes :

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

The key size used for an AES cipher specifies the number of transformation rounds that convert the input, named the *plaintext*, into the final output, named the *ciphertext*. The number of rounds are as follows: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, 14 rounds for 256-bit keys. Each round consists of several steps, with one that depends on the encryption key itself. A set of reverse rounds are applied to transform *ciphertext* back into the original *plaintext* using the same encryption key.

High-level scheme of the algorithm

- **KeyExpansion** – round keys are derived from the cipher key using the AES key schedule. AES requires a separate 128-bit round key block for each round plus one more.
- **AddRoundKey** – each byte of the state is combined with a byte of the round key using bitwise xor.
- A cycle of 9, 11 or 13 rounds:
    1. **SubBytes** – a non-linear substitution step where each byte is replaced with another according to a lookup table.
    2. **ShiftRows** – a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
    3. **MixColumns** – a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
    4. **AddRoundKey**

- Final round (making 10, 12 or 14 rounds in total):
    1. **SubBytes**
    2. **ShiftRows**
    3. **AddRoundKey**

2.1.1 AES boolean masking implementation

Masking randomizes the intermediate values of a cryptographic computation to avoid dependencies between these values and the power consumption [4]. It is applied on an algorithmic level and does not rely on the power consumption characteristics of the device. Each intermediate value is concealed by a random mask that is different for every execution. Basically corresponds to one of the following two secret sharing schemes with two shares: boolean secret sharing and multiplicative sharing. All linear functions in a cipher can be masked by boolean secret sharing, since the mask is changed in an easily computable way–AES AddRoundKey, ShiftRows, MixColumns. Non-linear functions are more difficult to mask, for example, AES SubBytes is non-linear, so 16 modified AES Sboxes must be computed, stored and used. This is infeaseble in some applications but SubBytes is based on the computation of the multiplicative inverse so it is compatible to multiplicative masking. To address that problem the state of art approach is to transform the boolean mask into a multiplicative mask, perform the inversion, transform the mask back and perform the affine mapping.

3 Software level abstraction for accelerator level side-channel masking

Since most masking functionalities are added on top of the unmasked algorithm flow, **we argue that they can be embedded on an already designed AES implementation with minimal changes by the designer**. We propose a fully synthesizable library that provides the designer all the necessary functionalities to mask an AES implementation with minimal changes to the original solution. The input code in C is modified by the designer by annotating secret variables, using the library provided function. The synthesizable library manages the creation of the random mask and ensure that the linear functions used in the AES implementation use the masked secret variables. The designer also add security checkpoints by calling a library function to ensure that the random mask are shifted at each iteration of the AES algorithm.

The designer can customize the security policies by selecting what masking implementation use. This solution has several advantages: it provides an infrastructure support for non-experts. Moreover, it allows hardware engineers to implement side-channel security countermeasures at an higher level along with the software implementation. **Since the library is fully synthesizable, security policies can be implemented and optimized with the rest of the accelerator logic**. The HLS tool can also introduces extra optimization to the side-channel masking without compromise to the security of the solution. In the next section we show an high level scheme explaining how to use the library.

3.1 Using the library to mask AES

We developed the library in C. The library has been fully synthesized with Bambu. We will discuss deeper about that in the next section. In this section we will show a quick example on how to use the library to secure an unmasked AES solution. To use the library, the designer includes it in the original code like with other standard libraries. The library has an entry point that handle the masking of the lookup tables, the key and the plaintext. The designer modifies the original code by calling a library function to start the masking process. Since the library handles the use of the lookup tables, the designer doesn't have to provide them in the original code. This function receives the plaintext and the key and triggers the start of the masking phase. To keep the modification to the original code as low as possible, the library saves intermediate data, resulting in a increase in memory usage compared to a solution not using the library. We will analyze the performances on the next sections. Regarding the AES algorithm, to mantain the randomness of the masking, at each cycle the designer has to call the library function "*remask*" that will handle the shift of the mask. Here, a quick scheme explaining how to use the library. In red, the modifications that the designer has to make to use the library.

Listing 1: High level use of the library

```
// Start the masking phase by calling the entry point of the library
startMaskingEncryption(state,key);
// AES algorithm
uint8_t round = 0;
AddRoundKey(0, state, key);
for (round = 1;; ++round){
  SubBytes(state);
  ShiftRows(state);
  if (round == numberRound){
        break;
    }
  remask(state); // Checkpoint: library function to shift the mask
  MixColumns(state);
  AddRoundKey(round, state, key);
}
AddRoundKey(numberRound, state, key);
}
```

4 Bambu

The HLS tool used is Bambu. Bambu is a free framework aimed at assisting the designer during the high-level synthesis of complex applications, supporting most of the C constructs (e.g., function calls and sharing of the modules, pointer arithmetic and dynamic resolution of memory accesses, accesses to array and structs, parameter passing either by reference or copy, ...)[5]. Bambu is developed for Linux systems, it is written in C++11, and it can be freely downloaded under GPL license.[6] The next figure illustrate the HLS flow of Bambu.

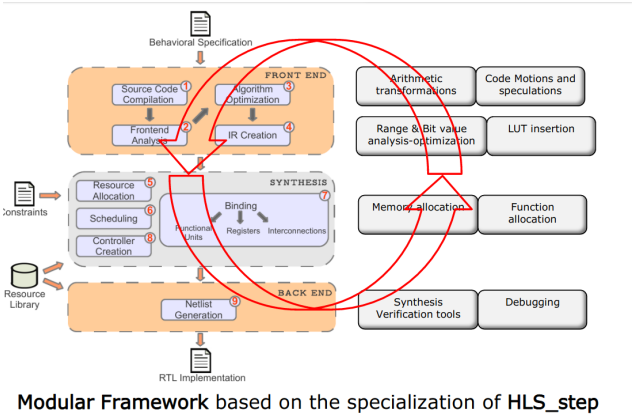

Figure 1: *Figure reproduced from BAMBU: An open-source framework for research in high-level synthesis, ICFPT 2017 tutorial*

Bambu allows us to create an hardware design of our library. Performance and/or area of the generated accelerators can be improved by tuning the design flow. Bambu has three layer of optimizations: GCC/CLANG optimizations, Bambu IR optimizations, Bambu HLS algorithms. The best design flow for every accelerator doesn't exist, so we tested multiple solutions to acquire the best design for our library. Bambu also allows to select a specifit target and a clock period. Testbench is automatically generated in Verilog by Bambu starting from random values or a configured XML file. We focused on optimization both at software level by reducing the space used by lookup table both at hardware level by leveraging Bambu optimizations. We found out that in our specific case, a trade off between memory usage and speed, was better than two different solutions especially given the fact that we had to mantain the security properties. In the next section we will discuss the performances of our solution.

5 Performances

In this chapter we analize the performances obtained with HLS by our solution comparing it with an unmasked solution and with a masked solution that implement the security countermeasure directly. The increase in memory from a standard masked solution and our library is due to the fact that to keep the implementation details transparent to the designer, we had to keep references to intermediate data.

	AES – Masked Library	AES - Masked	AES - not masked
Total cycles	1741	1324	1260
Average cycles	1741	1324	1260
BRAM bitsize	16	16	16
DATA bus bitsize	32	32	32
ADDRESS bus bitsize	11	12	10
SIZE bus bitsize	6	6	6
Internal private memory	2079	1547	768
Internal no private memory	1280	1024	1291
HLS execution time	3.38	12.21	12.239

bambu executed with: bambu –generate-tb=test.xml –simulate -O3 –clock-period=15 -fno-tree-loop-distribute-patterns main.c

Our library based solution allow to create ad-hoc masking component and results in a **huge increase in HLS execution time** with a relatively small increase in occupied.

6 Conclusion

We have analyze software level extensions and annotations to add security protections that can be later automatically synthesized with HLS. We have focused on masking implementation against side-channel attacks. Future works will be focused on other security properties that can be abstracted at software level and synthesized without affecting their security.

References

[1] Christian Pilato, Francesco Regazzoni, Ramesh Karri, and Siddharth Garg. Tao: Techniques for algorithm-level obfuscation during high-level synthesis. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 57–64, 2018.

[3] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications*, pages 277–284, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[4] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of aes. volume 3357, pages 69–83, 01 2005.

[5] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, 2013.

[6] Gnu general public license, version 3. http://www.gnu.org/licenses/gpl.html, June 2007. Last retrieved 2020-01-01.