UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc « Distributed Systems, Security and Emerging Information Technologies »

ΠΜΣ «Κατανεμημένα Συστήματα, Ασφάλεια και Αναδυόμενες Τεχνολογίες»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Development of hardware countermeasures for embedded systems security using High-Level Synthesis** Ανάπτυξη αντιμέτρων υλικού για την ασφάλεια ενσωματωμένων συστημάτων με χρήση High-level Synthesis
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Amalia-Artemis Koufopoulou** Αμαλία-Άρτεμις Κουφοπούλου
Father's name: Πατρώνυμο:	**Emmanouil** Εμμανουήλ
Student's ID No: Αριθμός Μητρώου:	ΜΠΚΣΑ19011
Supervisor: Επιβλέπων:	**Michael Psarakis, Associate Professor** Μιχαήλ Ψαράκης, Αναπληρωτής Καθηγητής

February 2022/ Φεβρουάριος 2022

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Panagiotis Kotzanikolaou **Associate Professor**	**Michael Psarakis** **Associate Professor**	**Athanasios Papadimitriou** **Assistant Professor**
Παναγιώτης Κοτζανικολάου Αναπληρωτής Καθηγητής	Μιχαήλ Ψαράκης Αναπληρωτής Καθηγητής	Αθανάσιος Παπαδημητρίου Επίκουρος Καθηγητής

Abstract

Today’s exponential needs for lightweight applications dictate the diminution of time-to-market requirements hardware-oriented applications are usually associated with, along with implementation restrictions compared to other computer systems. At the same time, the process cannot abate the anticipated level of security by any means. High-Level Synthesis (HLS) tools have proved themselves as a vital assistant in such a process, since it allows developers to use familiar, high-level language (HLL) along with optimization strategies to formulate the desired functionality, defined in a hardware description language (HDL). As a result, hardware development can become an easier, quicker process, leaving room for verification and validation on an early stage. On the downside, this methodology has not yet been tested thoroughly regarding the quality of the generated output compared to traditional HDL development flow. For that purpose, an AES cryptographic algorithm and some known side-channel attack countermeasures applied over it have been put through the HLL-to-HDL workflow offered by Vivado HLS tool, using different Synthesis directives. The resulting designs were finally compared by means of timing and are utilization, two key characteristics for embedded applications. It was finally determined that, while the default settings of the HLS tool offer a result of acceptable quality, the use of the directives under scope can either benefit of worsen those two aspects. The use of such configurations then should be considered regarding the underlying architecture as well as the needs of applications.

Keywords : Embedded Systems, High-level Synthesis (HLS), AES, Side-channel Attacks countermeasures, directives

Περίληψη

Η εκθετική αύξηση των αναγκών για χρήση ενσωματωμένων «ελαφρών» εφαρμογών απαιτεί τη μείωση του χρόνου ανάπτυξης που συχνά σχετίζεται με τις εφαρμογές που στοχεύουν χρήση στο Υλικό, και παράλληλα, την επιβολή αυστηρότερων περιορισμών υλοποίησης, συγκριτικά με άλλα υπολογιστικά συστήματα. Την ίδια στιγμή, το επίπεδο της Ασφάλειας δεν πρέπει να παραμερίζεται. Τα εργαλεία High-level Synthesis (HLS) είναι σε θέση να παρέχουν σήμερα την απαιτούμενη υποστήριξη στην επίλυση των προβλημάτων ανάπτυξης ενσωματωμένων εφαρμογών. Επιτρέπουν τη χρήση γνωστών γλωσσών προγραμματισμού ανώτερου επιπέδου (high-level languages – HLL) για να περιγράψουν στην ζητούμενη συμπεριφορά, την εύκολη επιβολή βελτιστοποιήσεων πάνω σε αυτή, αναλαμβάνοντας τελικά την αυτόματη μετατροπή σε μία γλώσσα περιγραφής Υλικού (Hardware-description language – HDL). Ως αποτέλεσμα, η διαδικασία της ανάπτυξης μπορεί να γίνει ευκολότερη και ταχύτερη, επιτρέποντας την περαιτέρω επαλήθευση της λειτουργικότητας από τα πρώιμα στάδια της διαδικασίας. Στον αντίποδα, η μεθοδολογία δεν έχει χρησιμοποιηθεί σε επαρκή βαθμό, ως προς την ποιότητα των αποτελεσμάτων της. Για αυτό το σκοπό, η μελέτη αξιοποίησε τον κρυπτογραφικό μηχανισμό AES, καθώς και αντίμετρα που στοχεύουν ευπάθειες του Υλικού απέναντι σε επιθέσει πλάγιο καναλιού (side-channel attacks), καθώς και την ποικιλία των παραμετροποιήσεων που παρέχει το εργαλείο Vivado HLS, υπό την μορφή ντιρεκτίβων (Synthesis directives). Οι σχεδιάσεις που προέκυψαν συγκρίθηκαν μεταξύ τους, ως προς τις μετρικές των χρόνου και του χώρου, δύο σημαντικά χαρακτηριστικά που πρέπει να λαμβάνονται υπόψη κατά την ανάπτυξη ενσωματωμένων εφαρμογών.Συμπερασματικά, προκύπτει ότι η προκαθορισμένη χρήση του εργαλείου αποδίδει μια ικανοποιητική ποιότητα ως προς της εξεταζόμενες μετρικές, ενώ η χρήση των ντιρεκτίβων μπορεί να τις επηρεάσει σε σημαντικό βαθμό -αρνητικά ή θετικά-, συνεπώς η χρήση τους θα πρέπει να εξετάζεται διεξοδικά ως προς τους περιορισμούς της πλατφόρμας εφαρμογής αλλά και τη φύση της εφαρμογής.

Λέξεις-κλειδιά: Ενσωματωμένα Συστήματα, High-level Synthesis, AES, αντίμετρα επιθέσεων πλάγιου καναλιού, ντιρεκτίβες

Contents

1. Introduction

Embedded systems can be viewed as autonomous computing units, each with its own hardware and software, capable of performing a small range of tasks, as defined by its manufacturer. It can operate independently or as a part of a larger system and is able to interact with elements such as sensors through integrated peripherals. One of the essential characteristics of embedded systems is their tight constraints. Their pervasive nature dictates that such systems shall operate under unstable environmental conditions, have minimal requirements in terms of cost, size, power consumption, while maintaining a satisfying level of performance, reliability and lifetime [1].

In the recent years, and in the brink of an Internet-of-Things (IoT) revolution, attention has been drawn over the use of embedded systems. IoT aspires to connect "anyone with anything at anywhere, anytime" [2]**Error! Reference source not found.**. Briefly, IoT involves low-cost devices acting as the end nodes of a system that gathers information of any kind and transfers them through the Internet, to similar devices or processing units. IoT applications generally employ embedded systems at the end nodes (edge devices) of the "IoT mesh"[1] to achieve the desired functionality. The resulting system is highly heterogenous, but the devices involved manage to collaborate "seamlessly", through a network that offers high speed communications.

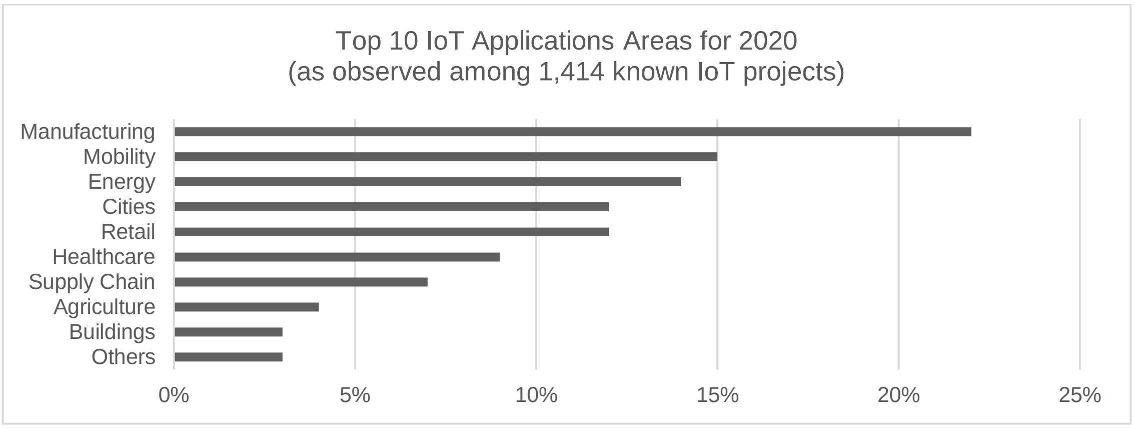


Figure 1. IoT trends as recorded in 2020 [3]

As shown in Figure 1, IoT application has taken over multiple sectors, aiming to resolve core issues, optimize tasks and allow the development of novel methodologies in an effective way. This can be heavily attibuted to the platforms involved. Most of the devices taking part in IoT systems heavily rely on integrated circuits (IC) to achieve optimality [4]. Those are small, cheap, energy efficient yet highly computationally capable electronic modules. The countless applications based on pervasive technologies need to be oriented towards those devices, hence, traditional programming associated with general-purpose computing systems is exchanged for developing over specialized hardware units. The term h**ardware acceleration** is used to encapsulate that powerful approach.

Gate-level development requires the use of programming languages that could capture a digital hardware's attributes. The most known consideration is that of concurrency, when computations are performed individually – and in extend in parallel-, whenever the input is available. This contrasts the fundamentals of some of the most commonly used procedural, "high-level" languages (HLL), whose goal is to offer an abstract, algorithmic manner of development, concealing any architectural concerns from the developer. For that reason, **hardware-description languages** (HDL) were developed to encompass such matters, while remaining independent of the electronic technology the design is applied over, with the most common of them being Verilog and VHDL.

[1] A highly connected topology used to describe the interconnectivity of IoT devices.

Along with them, Electronic design automation (EDA) tools appeared to aid the design process. Their workflow generally comprises of:

- The **logic development**, usually in a hardware description language (HDL) which results in an abstract model representing the behavior of data -signals- among the registers (register-transfer level - RTL).
- The **RTL synthesis**, which transforms the RTL into electronic components and wiring, thus creating a netlist.
- The **Physical Synthesis,** which places the resulting netlist over the target technology. Place-and-route step is the slowest HDL-to-bitstream operation, since a complex optimization needs to be made: find out the shortest paths among the computational blocks so that the design is optimal.
- Among these steps, the best practice is to perform **simulations** to verify the correct functionality of design in terms of behaviour or timing.

The first problem discussed over this thesis is that **developing and verifying applications directly on gate-level is, by definition, a daunting, slow task**. This stems from the fact that the functionality doesn't target a processor that operates in a serial, easily predictable manner. Instead, it is implemented over logic cells, which require a deep understanding to determine their relationships and behavior from the developer's perspective. While the additional complexity that comes with it is mostly an acceptable trade-off [5], today's rising demands press for changes in the process followed. A novel methodoly to tackle those this issue is presented in Chapter 2 of this thesis.

The second objective tackled concerns the **security aspect of hardware accelerators**, examined extensively in Chapter 3. It is obvious that security must be enabled, especially over a technology that has penetrated every aspect of everyday life. Numerous examples of the importance of embedded systems' security can be presented. Perhaps the most indicative example are "smart cards", a personal card with its own microprocessor. Nowadays, such devices can possibly contain a variety of personal information in order to identify their owner and allow ubiquitous access to services. In addition to their flexibility, smart cards are constantly improved to accommodate sometimes multiple, more complex capabilities.

Security in those novel platforms becomes of utmost importance then, considering a number of factors. The first is the extended attack surface of such systems. As of 2019, the number of connected devices exceeded 7 billion [6], a value that will exponentially increase with the adoption of 5G technology, which will bring a greater variety of applications requiring embedded systems. The heterogeneity of the components taking part in an IoT mesh should also be taken into account, along with the lack of security advancements in the field, mainly because it is practically impossible to keep up with the rapid deployment of larger, more complex IoT applications.

The use of hardware devices specifically comes with additional challenges. The implementations must retain their low-cost, low power consumption characteristics in order to support pervasiveness. The nodes are inherently constrained in terms of computational capabilities, and they depend on other systems accessed through the network to perform any needed processes. By that means, they cannot support elaborate cryptographic mechanisms. This resulted in a surge in research over "lightweight cryptography" methodologies [7].

In addition, the implementation itself presents certain electronic phenomena related to the cipher's behavior, and that can be exploited instead of seeking weaknesses over the cryptographic algorithms. This has resulted in a new family of attacks, discussed later. It has been observed that a more computationally intense cipher can lead to more of such exploitable phenomena occurring [8].

Given those, a series of countermeasures were developed with the use of the HLS methodology, over a known cryptographic mechanism, in respects to important metrics, presented in Chapter 4. In addition, the directives used through the HLS tool under examination are presented at Chapter 5,serving as a guide for their better understanding. The directives finally used are specified in Chapter 6, defining the designs' solutions. In Chapter 7 the results are compared in regards of the metrics defined. Lastly, in Chapter 8, the conclusions are drawn and future research directions are discussed.

2. High Level Synthesis (HLS)

HLS has become a popular solution to the first problem presented above, applying the simplicity of the high-level abstraction over the design flow, performing a **behavioural synthesis**. An HLS tool takes an HLL program, and automatically produces an RTL design with the same functionality, described in HDL. This enables anyone with the basic knowledge of an HLL, usually C/ C++, to access the hardware programming advantages without the need to ponder over implementation details. At the same time, human error is substantially reduced, the resulting programs are more flexible and far easier to debug [9].

The process of transformation from HLL to HDL automatically takes over tasks that otherwise would be the designer's responsibility. The variety and order of them may vastly differ from one tool to another, but the basic tasks are [10]:

- *Data-Flow Graph (DFG)*, which is a general representation of the inputs and outputs of a design, the operations and their interoperability, and how the dataflow is shaped, given the data dependencies. This step can be extended to *Control/Data-Flow Graph (CDFG)* to include the control flow (conditional cases).
- *Resource Allocation*, which determines the resources that may be required for the implementation of a design, given information from the previous step. The resources shall be enough to implement every operation defined, but in an optimal way, meaning the minimal use of more complex components.
- *Scheduling*, the essential step for behavioural synthesis, which designates operations to clock cycles, according to the timing characteristics of both the hardware and the design. Depending on the methodology used, scheduling can be categorized to:
    - *Scheduling for unconstrained designs*, for which the only limitation is data dependencies. Those are more commonly used as an estimation rather than the realization of implementations.
    - *Scheduling for constrained designs*, which take into consideration constraints posed by resources or time limitations and given that, aim to produce the most efficient implementation possible.
- *Register Allocation*, that examines cases where registers are required (example, in data dependencies that last longer than one cycle). Lifetime Analysis can also be performed to determine data validity along the time, enable sharing and therefore minimize the number of registers that will be needed in the design.
- *Binding*, which assigns the operations scheduled to the associated hardware resource.
- *State Machine Extraction*, at which point the sequence of operations is finally defined. The state machine of the design is extracted from the scheduling step and instructs how the data are passed in the components, according to the current state.
- *Netlisting*, which results in the synthesized design. Netlisting can be performed between other steps, offering a form of verification.

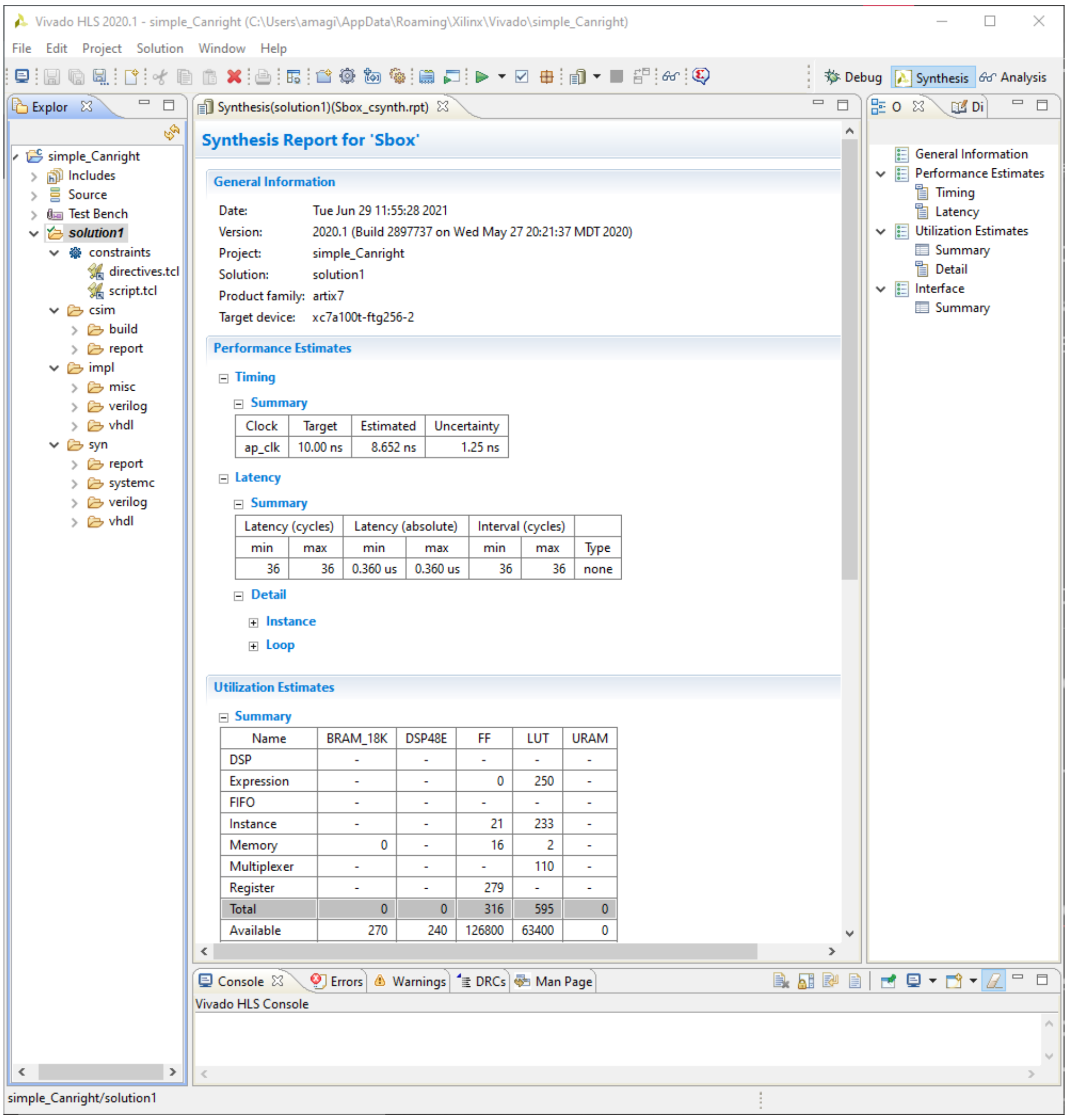
Interface Synthesis defines the form of the interface of *top module* in a design, consisting of the input and output signals, along with various control signals that allow the communication with other modules and peripherals. The inner functions become separate blocks, whose operation creates a hierarchy of modules and entities. As mentioned, the whole process is automatic, yet most HLS tools allow a level of manipulation from the user, especially in order to take advantage of hardware benefits. Testing can be then performed in the earliest stages of the design process with minimal effort, assisting design exploration.

It should be noted though, that while most HLS tools enable designers to overcome the difficulties of hardware application development, they should not entirely dismiss the fact that the code will be finally implemented for hardware and should address their limitations. Sequential-to-Concurrent logic transition needs to be taken into account while developing a design. Memory management also differs. For example, there are high-level techniques, such as dynamic memory allocation, that, though they greatly benefit the code on the higher-level, they don't synthesize well or not at all. Such methods include dynamic programming techniques, pointer usage, recursion and system calls [9].

After three decades of extensive research, trial and error, a wide variety of academic and commercial tools exist [11] [12]. The parameters involved in the choice of a tool include the existing documentation and vendor support, the target device used, the optimizations abilities, the resulting RTL in terms of metrics such as resource and power usage, latency, etc, the correctness of verification and finally the overall user experience.

2.1. Vivado HLS

The tool used in this thesis is **Vivado HLS** (version 2020.1), developed by Xilinx. It is considered part of the Vivado Design Suite, which allows developers to take a high-level algorithm to programmable logic through an extended design flow. It is an Eclipse-like IDE, as presented in Figure 2, an environment familiar to most programmers. A Vivado HLS project contains the high-level code, a testbench file, from which the inputs to-be-examined are given to the code, and at least one *solution*, in which the constraints and the optimization details of the project are defined.



Synthesis Report for 'Sbox'

General Information

Date: Tue Jun 29 11:55:28 2021
Version: 2020.1 (Build 2897737 on Wed May 27 20:21:37 MDT 2020)
Project: simple_Canright
Solution: solution1
Product family: artix7
Target device: xc7a100t-ftg256-2

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.652 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
36	36	0.360 us	0.360 us	36	36	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	250	-
FIFO	-	-	-	-	-
Instance	-	-	21	233	-
Memory	0	-	16	2	-
Multiplexer	-	-	-	110	-
Register	-	-	279	-	-
Total	0	0	316	595	0
Available	270	240	126800	63400	0

Figure 2. Vivado HLS graphical user interface. After the Synthesis, a report is available.

At the **Synthesis perspective**, the code can be developed and verified through the testbench. Debug option is also available. The solutions can be then synthesized, producing the respective functionality in VHDL or Verilog files, along with a brief report containing performance estimates. Alternatively, the more detailed **Analysis perspective** can be examined. It is an interactive tool that offers a glimpse of how the scheduling and binding affect on the given code. A good practice is to co-simulate high-level and RTL, an option Vivado HLS also offers using one of the integrated RTL simulators, along with a wave viewer. Finally, the RTL export as IP is possible, which can be imported at Vivado.

The optimizations are optional that can be applied come in the form of **directives**. Those can specify, among others, the minimum or maximum latency of a code block, the resources to be used for an alternative handling of variables (example, what memory types are used for array implementation) or limitations over them, how loops are implemented (unrolled, merged), whether pipelining is going to be applied, modifications in the dataflow and the protocols that control input and output signals. All aim to optimize the application of the code over the hardware platform. Lastly, besides the user interface, Vivado HLS can be accessed through the bash, with the use of tcl commands[13].

3. Cryptographic Acceleration

Cryptography is by no means a novel field of study. Its use can be traced back along with the first signs of written language, among many different cultures. Up until today, its concept remains the same: To find ways to effectively hide information communicated through any means, while providing a method of unveiling them, known only to the intended parties. Cryptography became a vital technique for the military and diplomatic services, defining, most notably, the outcome of the two World Wars [14].

The dawn of the Information Age revamped the interest in cryptography. While the central idea remained the same, new requirements rose because of the digitalization of information, as well as the environment those are exchanged, today in greater numbers than ever. Mathematics and Informatics rapid development allowed the creation of much more complex cryptographic algorithms (ciphers) for the uncertain digital world. The primary mathematical principles and theorems discussed in every study concerning the development and evaluation of cryptographic mechanisms were set by the emblematic work of C.E. Shannon in the late 1940’s [15][16].

The importance of cryptography is such, that nowadays, information security is a prolific field of study, as well as a thriving industry, since it has become essential for any organization operating on the cyber environment. Cryptanalysis, the study of information systems for weakness in cryptographic methods and implementations has been also proved to be a useful tool, since it allows the better understanding and evolution of cryptographic algorithms.

The main terms of cryptography that need to be noted are the **plaintext**, the initial information meant to be delivered from one end to another, and the **ciphertext**, the encrypted plaintext that is communicated. Lastly, the **cryptographic key** is a piece of information that is used as a parameter in a cipher to transform the given plaintext to the ciphertext through the encryption process, and must be kept secret. In fact, the whole success of the process depends, not on the secrecy of the algorithm used, but on the secrecy of the key. The key’s quality depends on factors such as the way it is generated, its size, etc. The key is the one that also allows the decryption, the reverse process to retrieve the initial information, to be performed.

Cryptography is part of Cybersecurity that encapsulates all the preventive measures applied over applications, systems and networks. The aim is to successfully tackle any possible threats, and when this is deemed impossible, to ensure the operational continuity of the system with the minimal possible damages. This ensures that the security of information has been thoroughly studied over the software domain, and will continue to do so, since absolute security is not feasible. It is critical then, that devices that operate in the uncertain cyber environment must apply a satisfying level of security.

3.1. HLS in Cryptography

The imperative requirement for secure applications developed for hardware along with the need for better time-to-market can be fulfilled by HLS tools. Modern cryptographic mechanisms are characterized by great complexity, an issue that can be overcomed with the use of Hardware. A common technique is the use of dedicated hardware to accelerate the computations. Therefore, known, tested cryptographic techniques existing in HLL form can be transferred easily over the hardware platforms with the use of HLS tools. Points of interest include the performance of those implementations and the level of security. Clearly, any modifications performed over the HDL-to-be code shall be oriented towards the efficient utilization of the resources and the capabilities of the hardware platform, but at the same time, cannot alter the level of security as observed in classic programming.The possibilities of HLS tools when it comes down to the production of a satisfactory cryptographic implementation have been thoroughly studied in bibliography.

The Advanced Encryption Standard (AES) is of particular interest in research work. In [17] the authors examine implementations depicting two different needs in the market today, high performance versus small area. [18] delves further into the use of directives, specifically pipelining, array manipulation and the use of BRAMS, comparing it, not only with other works, but a theoretically optimal implementation. [19] goes a step further and compares multiple implementations over different platforms with an equivalent RTL, developed from scratch. Although the authors conclude that the RTL version is, by a small margin most of the times, better than the HLS-generated, they point out the value of HLS tools rests on the ease of development they offer. Another aspect that needs to be concerned is the novel threats hardware platforms face, such as fault injections, timing attacks and power analysis attacks, among many others. Those kinds of attacks, presented in the taxonomy shown in Figure 3, can be realized due to different vulnerabilities presented in the design, either because of bad designing methods or lack of understanding of the way those operate.While HLS tools can assist the designing process by minimizing human error, the development of countermeasures has become a prolific field of research.

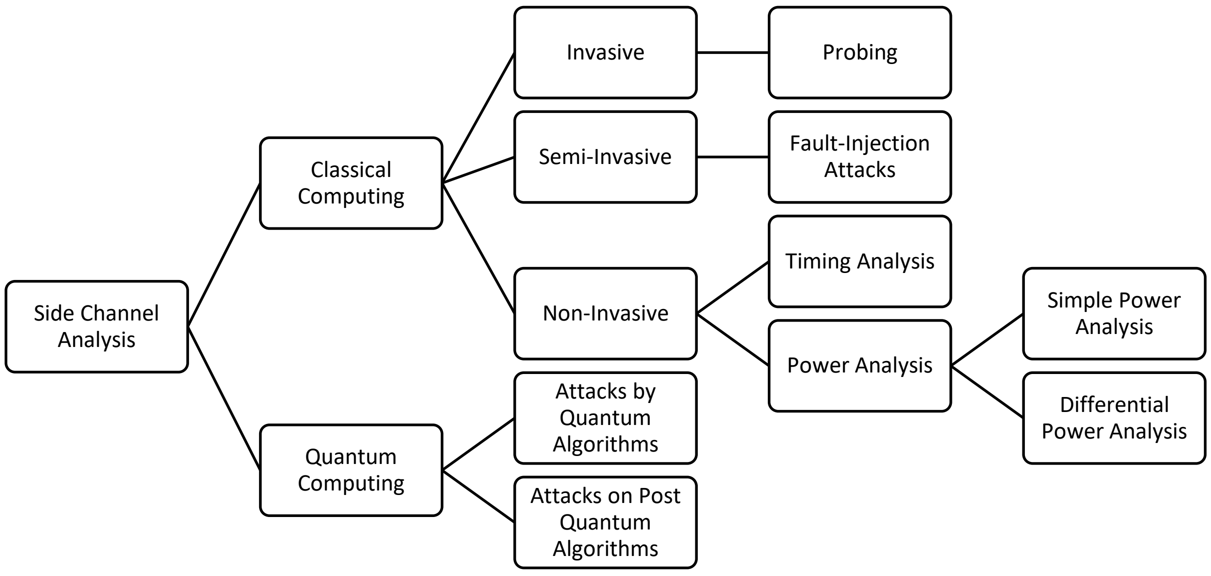


Figure 3. Side-Channel Analysis Attacks taxonomy

Attempts to develop such countermeasures especially designed to be incorporated to HLS code have been recently made, as in the case of [20], yet the level of security they provide is unsatisfactory and births questions about the reliability of the tools. [21] attempts to identify the cause that can lead to flawed designs, targeting in particular HLS Synthesis flow. Lu Zhang's work has been particularly influential to the field of HLS development where security-by-design methodologies are explored, as in [22], where machine-learning is employed to determine design patterns that are related to weaknesses and [23], where the relationship of directives and security is explored over an AES function.

Lastly, a great research field over which HLS could be used future is the development of countermeasures for quantum algorithms. As soon as quantum computing becomes accessible, many cryptographic measures will become useless, putting every application that they are applied at risk. HLS tools can support the faster development of accelerators that will be part of new countermeasures [24], given that any problems regarding the reliability of security measures will be explored more.

3.2. Side-Channel Attacks

Physical devices are particularly susceptible to a family of attacks named side-channel attacks (SCA). Those aim to extract information related to cryptographic operations indirectly, through the examination of side-channel leakage, which comes in the form of timing characteristics, electromagnetic radiation and power consumption. The physical behavior of the device has been proved to be statistically related to the data processed, whether the operations are performed on hardware (logic gates) or software level (embedded processor) [20]. The process is simple, costless and doesn't require heavy physical probing of the device, making them extremely dangerous against embedded systems [25].

In Power Analysis Attacks, sampling the power consumption leakage, a quite simple and reliable operation nowadays, allows the depiction of the device's electrical activity as points and the visualization of its behavior, on what is known as a *power trace* along the time domain. The form of a power trace is shown in Figure 4.

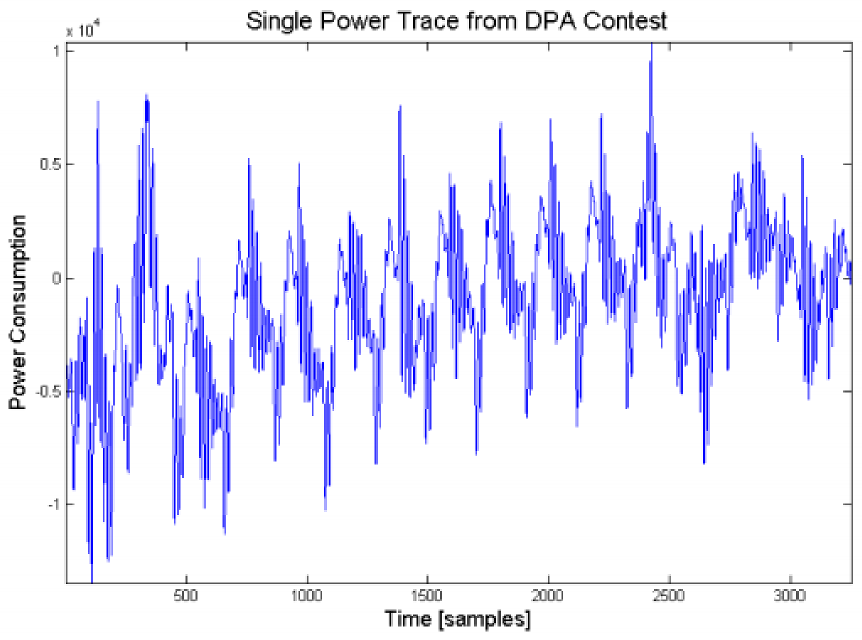


Figure 4. Example of power trace. Patterns that correspond to specific computational functions can be determined [26].

3.2.1. Power Trace Acquisition

The extraction and study of the power consumption information is capable of unveiling evidence for any means of cryptographic mechanism performed by it. This is possible because the electrical current that runs through a device's transistors indicates their activity. Note that other hardware-related phenomena can appear in this activity, such as noise, which needs to be eliminated to acquire a higher-quality power trace. Noise cancelation is trivial and can be achieved by averaging a large number of traces.

The relation of current I that runs through a device to voltage V measured across it, which is the metric that depicts the power consumption is derived from Ohm's Law:

$$I = \frac{V}{R}$$

where R indicates the resistance of the device. If R remains constant, then the current is proportionate to the voltage. Knowing that, an instrument called oscilloscope can be used to obtain the graph of voltage changes over time and therefore, the power usage.

3.2.2. Simple Power Analysis Attacks (SPA)

Power Analysis Attacks can be categorized based on the methodology followed. In SPA attacks, sensitive information can be derived directly from the traces. For example, in Figure 5, one can easily deduce the rounds o operation two cryptographic mechanisms perform. While this requires only a small amount of such measurements to extract the information, it also demands a detailed knowledge of the operations involved in the cryptographic algorithm examined, and how they correspond to the pattern the power traces form.

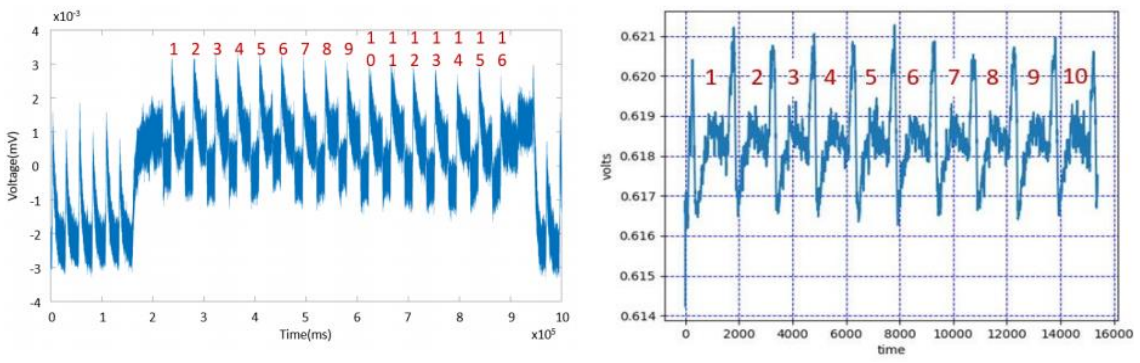


Figure 5. SPA traces depicting the rounds of DES (left) and AES (right) encryption rounds [27].

Prevention of such attacks includes the avoidance of direct use of the key in the computations performed, the addition of noise and the use of desynchronization technique. The latter involves the random addition of "useless" instructions with no real effect on the result along with the functional instructions, in order to tamper with the presented power consumption. All those result in the distortion of the observable pattern on which SPA depends on.

3.2.3. Differential Power Analysis (DPA)

DPA comprises a sub-class of an advanced type of side-channel attacks. It has prevailed over SPA, as it can overcome practical limitations such as the lack of knowledge over the cryptographic mechanisms and noise, whether it is unintentional or imposed as a defense measure. DPA attacks are based on the statistical analysis of a large number of power consumption measurements derived from the processing of different data (such as a series of known plaintexts) by the same cryptographic module. The function opted for examination must operate on data that include the key.

For every data d_i then, the power trace t_i of it, processed through a function, is extracted. The alignment of the collected power traces is important in order to reduce the required power traces. A common technique used to achieve that (at least on a simulated attack) is using a trigger signal. Additionally, to make the whole process less costly, a specific part of the traces may be examined (for example, the first or last round in a symmetric block cipher), as defined by specific peak points within the trace (see Figure 5).

Having the initial data and (a hint of) the leaked result from a function involving the key, the key is then *hypothesized*. Hypothetical keys are all possible values that can serve as a secret key in a cryptographic algorithm. For example, in AES-128, the key has a length of 128 bits, hence all values between $(0)_{10}$ and $(2^{128}-1)_{10}$ are possible keys. If every d_i goes through the same function, for each hypothesis key k_i, the hypothetical result $V_{i,j}$ of the function is calculated. This is then simulated at the logic level to render the corresponding hypothetical power consumption trace $h_{i,j}$. Most commonly, Hamming distance and Hamming weight models are used at this stage. Hamming distance is more efficient than Hamming weight, provided that information about the calculations performed are known to the attacker [28].

The real traces extracted by the device are eventually compared to the hypothesized traces. The result $r_{i,j}$ is a metric that shows how well the two traces relate. The Correlation coefficient method is widely preferred. The pair with the higher degree of relation can reveal the key used. The values $r_{i,j}$ can also serve as an indicator of how precise the estimations were. A higher precision occurs when more traces are examined. Note that this process may only unveil part of the key, depending on the way it is handled by the cryptographic algorithm. For example, in AES, the secret key goes through a key scheduling algorithm, and only the round key is derived, which corresponds to one byte of the secret key, so the process has to be repeated to unveil the rest of the key.

There are several methods of preventing DPA attacks. A signal degradation method can be used, by choosing operations that leak less information or by applying physical countermeasures. Introducing noise in the power consumption will result in higher computational requirements from the attacker's side. The cryptosystems can also apply non-linear key updates, to thwart any attempts to correlate the power traces to a specific key by preventing the gathering of the required number of samples for a reliable DPA attack. Additionally, as implied above, the addition of any means of misalignment of the power traces can hinder the process [29].

Security against DPA attacks generally occurs when independence is introduced between the intermediate results and their power traces. This can be accomplished with the use of hiding and masking methods, the main two mechanisms used to counteract first-order DPA attacks. **Hiding** is a solution that aims to the manipulation of the power consumption, either by randomizing the execution of the algorithm, or, on the hardware level, achieving a different energy expected (random or minimal) for all the operations executed rather than the one.

Masking can manipulate the intermediate results, and thus produce a different power consumption leakage, statistically independent (theoretically) from the initial results that in other case, are exploited successfully due to their dependence on sensitive data. The manipulation of the intermediate results is accomplished with the use of one or more randomly generated masks. The masking scheme is applied to the input data and is carried through the encryption operations to the produced ciphertext, from which it will be removed to give the expected ciphertext.

3.2.4. Higher-Order Attacks

As countermeasures are developed for every vulnerability discovered, new kinds of attacks of greater complexity appear, necessitating novel techniques. The idea of how a more complex (higher-order) attack could be crafted appears in the fundamental work of Kocher et al. [29] about DPA attacks. Considering an operation that handles data which have been subjected to masking, the effect of the mask over the leakage is not predictable, which is the reason behind masking's success. But this can be overcome if the joined leakage of two different inputs masked with the same value is examined. The number of traces required is higher than a first-order DPA attack, still, an approach of determining that amount in advance, utilizing statistical methods, has been developed. Moreover, *points of interest* can be determined for the attack to focus on, taking the absolute difference of a leakage model metric (ex. Hamming weight) of two joined leakages. Such a pre-process can formulate a more efficient DPA attack, overcoming the computational requirements the countermeasure sets.

Higher-order attacks (HODPA) leverage the pre-processing step. While the problem becomes multivariate, this extra step allows its reduction to a univariate problem, on which a 1st order attack can be then applied [30]. It should be noted that the complexity of such an attack itself hardly makes any effort of executing it tempting. Making a system hard enough to attack is considered a form of security (computational security).

3.3. AES

Rijndael Block Cipher is one of the most well known symmetric encryption algorithms. It was developed by Vincent Rijmen and Joan Daemn in 1999 [31] and was adopted by the U.S. National Institute of Standards and Technology (NIST) as AES in 2001, replacing the Data encryption Standard (DES and 3DES).

It is designed to be simple yet robust, fast, and transferable. It is an explementary application of the *confusion and diffusion* principle, as defined by Shannon[15]. According to his theory, **confusion** is achieved when the relationship between the encryption key and the ciphertext becomes indistinguishable. **Diffusion** on the other hand means that a modification solely on one bit of the plaintext can statistically affect half the bits of the ciphertext. This is simply realized by permutating and substituting the elements of the given plaintext. The resulting ciphertext then carries no statistical relationship to the original data. Rijndael algorithm fulfills those principles, with the use of non-linear and linear functions, performed over the data in multiple iterations. Moreover, with small modifications, its functionality is invertible, meaning that decryption of a given ciphertext is possible.

AES algorithm is categorized as a *block cipher*, meaning its operation is based on fixed-length groups of data called blocks. The use of blocks has advantages and disadvantages, leading to the creation of multiple **modes of operation.** NIST has standardized five of these modes, briefly presented below [32] [33]:

- Electronic Code Book (ECB) : The original, simplest form of AES, where each data block is encrypted individually. While it can protect data from error propagation, it can result in repeated encrypted patterns.
- Cipher Block Chaining (CBC) : In this case, each plaintext block is related to the previous ciphertext. Before the encryption process, and XOR operation is performed among them. For the first plaintext block, an *initialization vector* (IV) is used, which is the one that eventually determines the ciphertext.
- Cipher Feedback (CFB) : In this case, the plaintext block is XORed with a portion of the previous ciphertext block, encrypted and shifted by a number of bits (encryptor).
- Output Feedback (OFB) : The operation resembles CFB, with the exception that the encryptor of the previous block is used instead of the cipher block.
- Counter Mode (CTR) : In the last case, a nonce[2] value is added to a counter, increasing by one per block. The result is encrypted, before XORed with the plaintext.

Each mode can be used in specific cases, according to the cryptographic needs. The algorithm examined from now on will be the ECB version.

The algorithm's input is meant to be of specific length, either 128, 192 or 256 bits, defining AES-128, AES-192 and AES-256 variations of the algorithm. If the length of the plaintext does not fit those cases, the data are split into blocks of the required length and padding is added, if required, at the last block [34]. The corresponding key must be of the same size. That length defines two values used in the algorithm:

- N_k, which equals the key length divided by 32, meaning it can either be 4, 6 or 8 and
- N_r, which, according to N_k and N_b, can be either 10, 12 or 14.
- N_b is also defined, but it depends on the length of the *word*, as the architecture of the platform defines it and most commonly is set to 4. N_b can be equal to 6 or 8, too.

[2] A random-generated value, used only once per occasion. It is widely used in cryptographic techniques.

N_r is used to define the number of iterations (rounds) of the encryption process. N_k and N_b define the number of columns and rows of the State block, the intermediate cipher result, formed as a two-dimensional array, its elements placed column-wise, as shown in Figure 6:

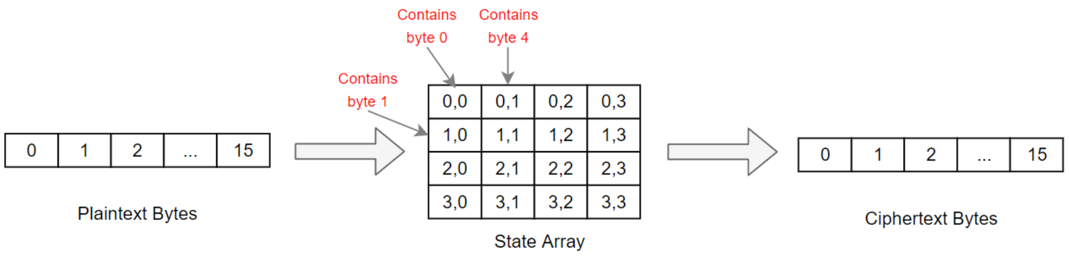


Figure 6. The State block formation for AES-128 encryption(N_k = 4, N_r = 10, N_b=4) [31].

Before the round operations begin, some preparatory operations take place. Firstly, **Key Schedule** is used to expand the given key into N_r +1 number of different round keys. In the case of AES-128, a total of (11 rounds*128 bits) = 1408 bits or 352 words need to be generated. The first round key (four words, 4 words*4 bytes*8 bits = 128 bits) is equal to the initial key. The following word is generated by the previous one, first rotated by 4-bits, then substituted through SubBytes() operation and lastly XORed with a value [rc_i, 0x00, 0x00, 0x00], with i denoting the number of the round. The result is XORed with the previous 4th word. The next 3 words are only XORed with the previous 4th word, thus the second round key is formed. The process is repeated for a total of 10 rounds, resulting in 11 round keys [35].

Each rc_i is equal to the polynomial x^{i-1}. For example, rc_1 equals x^0 = 1, rc_2 equals x^1 = x and rc_8 equals x^7. After rc_9, the polynomials are divided by the polynomial $x^8+x^4+x^3+x+1$. The results of the calculations are known and usually constant values are used, expressed in hexadecimal representation. The way polynomials are expressed in hexadecimal forms is analyzed in extend later.

Each encryption round of AES consists of four steps:

- **SubBytes(State)** : A non-linear byte substitution, applied over each of the State's bytes separately. It involves a polynomial inversion of a maximum degree 7 and an affine transformation. Most commonly, a substitution (look-up) table is used, whose construction is explained in extend later.

- **ShiftRows(State)** : It is applied over the rows of the State and shifts their content cyclically, on an offset determined by N_b. For example, in the case of N_b = 4, the first row is not shifted (shifted by 0), the second row is rotated left by 1, the third by 2 and the fourth by 3. Right rotation is used for the decryption process.

- **MixColumns(State)** : It is applied over the columns of the State this time. Considering the column as a matrix a_i, each of is multiplied by a fixed matrix, producing a new matrix b_i. The fixed matrix is a maximum distance separable (MDS) matrix, that offers the required diffusion to the algorithm. The encryption process uses the following MDS matrix to determine the State's new columns:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The last round of encryption skips the MixColumns () step, as it has provably no effect on the security of the algorithm.

- **AddRoundKey(State, RoundKey)** : Finally, a Round Key is applied to the State, with a simple XOR operation. The Round Keys are derived from a Key Schedule preparatory step, are of length N_b, and their number is equal to N_r.

The operation flow of encryption and decryption is clearly demonstrated in Figure 7:

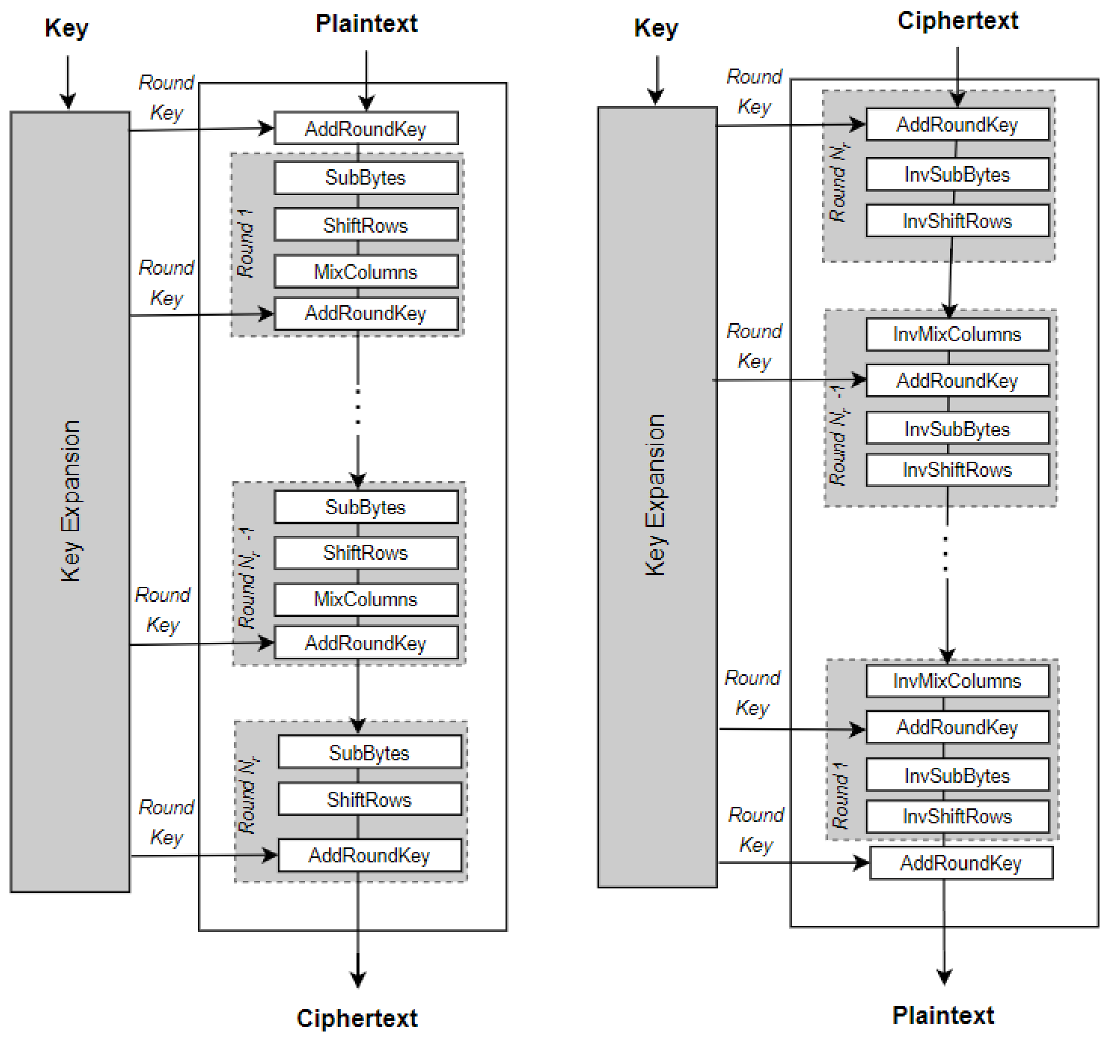


Figure 7.The AES-128 algorithm diagram for the encryption and decryption process [36]

AES is highly flexible, allowing it to be applied over the application of specific requirements, such as hardware-based applications operating in a highly interconnected network. It is proved more efficient in terms of throughput and energy-efficiency than DES versions [37].

3.3.1. Security of AES

The AES algorithm stands against cryptographic attacks today, as it provides a satisfactory security margin for the immediate future, considering the current computational power of processors, as well as the one predicted by Moore's Law. As mentioned before, it is a common practice for many modern cryptographic algorithms to depend their security on the fact that contemporary computational systems are not powerful enough to break them, or that the attack required is far too complex to compensate for its outcome.

The confidence of AES security is such that it is the most used cryptographic algorithm today. A successful key attack will result to a cessation of activities performed in the present digitalized world. Interestingly, the Rijndael's AES mechanism is considered to be the top candidate to applied on smart cards due to its minimal requirements [38].

In its simplest form, the algorithm uses 128 bits for its secret key, which means the secret key lies among 2^{128} possible values, with every bit added up to 256 doubling that range (though, in the case AES-128 is indeed broken, AES-256 could no longer be deemed as safe). A brute-force attack, the simplest attack one can perform by exhaustively examining each one of those possible keys, would not be feasible with the current technology.

The best option for a successful attack on AES is the Biblique attack [39]. It's a meet-in-the-middle type of attack, most commonly used against hash functions, that works by splitting the possible keys into two groups of 2^{2d} keys, which then are formulated into a 2^d x 2^d matrix. A key k_1 exists in one group and a k_2 in another. The encryption function is also split in a way that E_k(plaintext) and $e_{2\ k2}(e_{1\ k1}$ (plaintext)) result in the same ciphertext. The same logic can be followed for decryption: D_k(ciphertext) equals $d_{2\ k2}(d_{1\ k1}$ (ciphertext)). Since D_k(ciphertext) = $D_k(E_k$(plaintext)) = plaintext, it can easily be proven that $d_{2\ k2}$(ciphertext) = e_{1k1}(plaintext). All the possible combinations are then attempted to figure the pair of k_1,k_2 that satisfies the equitation. This can be enhanced with the use of graph theory, to attack the intermediate states of AES more efficiently. Still, this only lowers the complexity to $2^{126.1}$. Quantum-based attacks are also examined [40], but to this day, no attack of such nature is provably threatening AES.

But this is not the case for side-channel attacks. While AES holds against timing attacks, it can easily be attacked with DPA. The type of leakage examined can affect the invasiveness of the attack. Power analysis may require the physical examination of the device, in contrast to Electromagnetic analysis [41]. Still, power traces can be extracted, although of lower quality, which in extend necessitate more power traces to derive a confident result[42]. In a simulated attack [43], it has been proved that a DPA attack on a hardware implementation of AES may only require 4,000 power traces to unveil the 8 MSB of the key.

3.3.2. Mathematical Background – Galois Fields

The idea of non-linearity is to transform a given piece of information to another, without exposing the relation between the data involved. In AES, this transformation is performed over bytes, which consist of 8 bits, with the leftmost bit being the most significant bit (MSB) and the rightmost the least significant bit (LSB). With that in mind, a byte can be represented as a polynomial of degree 7:

$$a_7 \cdot x^7 + a_6 \cdot x^6 + a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x^1 + a_0 \cdot x^0$$

where the coefficients a_7 to a_0 will either be 0 or 1. A byte with the value 0x63 (hexadecimal form) can be represented as {01100011} (binary form) or as the polynomial x^6+ x^5+ x^1+ x^0, and vice versa. Going back to Key Scheduling, it is now clear that polynomials up to rc_8 can be easily represented with constants.

The most common computing architectures today, define the byte as an 8-bit vector, which can depict 2^8 or 256 different elements, from {0000 0000} to {1111 1111}. Hence polynomials over rc_8 cannot be represented, due to architectural limitations. The limited range of numbers consists of a **finite field** of numbers, also named **Galois field**. Generally, a Galois field of the form GF(p^n), where p is a prime number, contains p^n different elements, represented as an n-1 degree polynomial. The respective Galois field in this case is identified as GF(2^8) or GF(256). There are more than one ways a GF(p^n) field can be defined, resulting into multiple *isomorphic fields.*

All operations defined in a Galois field behave according to specific rules:

1. Operations performed among bytes have to return a byte defined within the finite field (closed set).
2. An expression using the same operator returns the same result, whatever the grouping of the quantities involved, as long as their order remains the same (associative property).
3. An expression connected with the same operator returns the same result, whatever the order of the quantities involved (communicative property).
4. An expression returns the same result, whether calculating it as is or in parts (distributive property).
5. Each element has an additive inverse. An additive operation - symbolized as $\oplus$ - of a quantity with its additive inverse returns the respective identity.

6. Each element, except 0, has a multiplicative inverse (reciprocal). A multiplicative operation - symbolized as $\otimes$ - of a quantity with its multiplicative inverse returns the respective identity.
7. For each operation, an identity value has to be defined. Operating with the identity quantity, leaves the other quantity unchanged.

The basic operations are addition and multiplication. Subtraction is considered as addition with the opposite value and division is considered as multiplication with the inverse value. The identities defined for each operation are 0 and 1 respectively. Since the values are behaving as polynomials, the operations will be examined from that scope.

In the classic **polynomial addition**, the coefficients of the variables with the same degree are added. For example $(x^8+1) + (x^8+x +1)$ equals $2x^8+x +2$. But in the polynomial representation of bytes, the coefficients will either be 0 or 1, and so will be the result of any possible addition among them. Hence, the proper result should be modulo 2, which corresponds to a simple **XOR operation**. The previous example would return as result $(x^2+1) \oplus (x^2+x +1)$, or for the shake of calculations {1000 0001}$\oplus${1000 0011} = {0000 0010} = 0x02. As mentioned before, **subtraction** requires the opposite value to be available. In the case of Galois fields, every element is its own opposite, making subtraction and addition essentially the same operation.

In polynomial **multiplication**, every term of one polynomial is multiplied with every term of the other polynomial. Using the same example, $(x^8+1)(x^8+x +1)$ equals $x^{64} + x^9 +2x^8 + x + 1$, which is obviously a problem given the 8-bit architecture. This time, the result needs to be modulo an *irreducible polynomial*, specific to the finite field. This implies the execution of a **division** between the multiplication's result and the irreducible polynomial, from which the remainder is kept. The irreducible polynomial is not unique for a finite field, and the search of an optimal one is an extended case of study. In the case of AES algorithm, the irreducible polynomial used is $x^8 + x^4 + x^3 + x + 1$.

Polynomial Inversion is a demanding operation, with the difficulty increasing along with the degree of the polynomial. Algorithms that take advantage of mathematical properties that involve the inverse can be used, such as the Extended Euclidean Algorithm [44] but it adds a considerable stalling to the calculations and is not ideal for hardware implementation. In the following sections the ways that the problem is overpassed in the software will be presented, and how can an optimal hardware implementation can be constructed.

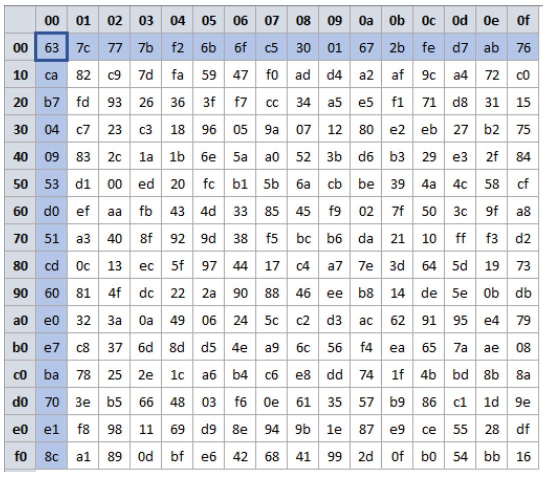
3.3.3. SBox construction

The SubBytes() function is the most demanding step of the AES cryptographic mechanism, with the intense computations performed certainly resulting in more leakages. Moreover, due to its non-linear nature, the leakages of the SubBytes() intermediate values differ greatly from one another, even if the data concerned are only one bit different. Interestingly, in [45] it is proved that a function that good against linear attacks lacks against differential ones. The operation consequently is ideal for DPA application, and secure measures are examined throughout bibliography especially for it.

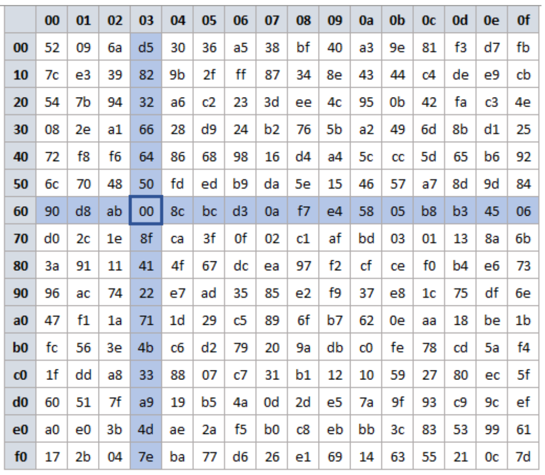
The SubBytes() operation performed in AES theoretically consists of two sub-steps :

- An **Inversion** of the given byte in GF(2^8) : If α is the data notation, α^{-1} is it's inverse, with $\alpha \otimes \alpha^{-1} = 1$. The inversion is followed by an
- **Affine Transformation**, which consists of:
  - Sum of multiple value rotations, implemented through a matrix multiplication and
  - a vector addition of $x^7+ x^6+ x^2+1$, which corresponds to the constant 0x63.

Instead of performing such intense computational operations, the results are precomputed and placed in a two-dimensional look-up-table (LUT) called SBox, thus only a mapping operation is eventually required. AES Sbox for encryption an decryption is shown in Figure 8. The input of SubBytes(), serves as two indexes, with the upper four bits indicating the row and the lower four bits indicating the column in which the substitution byte is located. The inverse SBox can be similarly constructed, with a different LUT depicting the inverse operations this time.



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16



	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 8. SubBytes(0x00) - left , equals the value of the element located in row 0, columun 0, which is 0x63. Similarly, inverse_SubBytes(0x63) – right, equals 0x00

While this idea serves a software application satisfactorily, its space requirements make it inefficient for hardware implementations. In the simplest AES version - input of 128 bits, 10 rounds of operation – parallelization is possible in hardware, due to the lack of dependency among the bytes. The 128 bits (16 bytes) of an input can use different instantiations of SubBytes(), meaning 16 Sboxes are required at the same clock cycle. Parallelization can also be applied over the 10 rounds AES-128 performs, meaning the implementation has to maintain 160 copies of Sbox, without considering any additional circuit involved. This translates to a minimum overhead of 41kB. From the security's perspective, the parallel computations may amplify the leakage of this operational step, making it vulnerable against side channel attacks. Alternative methodologies and architectures are examined to limit the power consumption of SubBytes() and in extend, making it safer. As per [46], pipelining of operations massively benefits all the implementations examined. One of them, using composite fields, is examined in detail below.

3.3.4. Canright's Compact Sbox

David Canright proposed a "compact" implementation SBox [47], developed especially for hardware applications, one that proved adequate and that is today regularly used as a basis in the research. Canright opts to perform the SubBytes computations instead of a simple mapping, applying various gate-level optimizations. At the same time his implementation takes into account the hardware limitations and the maintenance of the algorithm's security properties.

Inversion in this case is performed with the use of *tower field representation*[3] [48]. The idea behind it is that any data defined in a finite field of the form GF(p^n) can be represented in the isomorphic field GF($(p^m)^2$), where n = 2m. Hence, the finite field GF(2^8) can be viewed as a (quadratic) extension of the field GF(2^4), noted as GF(2^8) over GF(2^4) or most commonly GF(2^8)/GF(2^4). Each element in G(2^8) is mapped to exactly one element of GF(2^8)/GF(2^4). While this is suitable for hardware implementation and is in fact used in many cases as optimal, Canright generalizes this property to represent the elements of GF(2^8) in GF(2^8)/GF(2^4)/GF(2^2).

The transformation of elements from GF(2^8) to GF(2^8)/GF(2^4)/GF(2^2) -and back to GF(2^8) after the computations- requires the *change of the basis* of one field to another. The basis of a finite field is defined as a set of vectors that can produce all elements of the field through linear combinations among them. In finite fields specifically, those bases are represented as polynomials (*polynomial basis*). Additionally, if the basis vectors are linearly independent, the resulting basis is called *normal basis*. It is proved that the use of normal basis is optimal in this case. The change of basis is achieved through a matrix

3

multiplication with a *transition matrix*. Since the last change of matrix is followed by the affine transformation's matrix multiplication, the two matrices can be combined to one.

The reason to perform this change of representation is to take advantage of the simpler operations of the lower-degree fields. Instead of performing calculations on a polynomial of degree 7 (ex. {1011 0111}, as expressed in the new representation), calculations are performed on polynomials of degree 3 ({1011} and {0111}), which in turn will be performed on polynomials of degree 1 ({10},{11}, {01} and {11}). Indicatively, the inversion in GF(2^2), where the elements are represented with two bits, is a simple swap among them[4]. Along with several more calculations, the inversion in GF(2^8) can be performed.

The algorithmic details are presented in the following steps:

1. *Change of basis of A*, from GF(2^8) to GF(2^8)/GF(2^4)/GF(2^2), with the use of transition matrix A2X, consisting of the elements { 0x98, 0xF3, 0xF2, 0x48, 0x09, 0x81, 0xA9, 0xFF }. The use of those values is one of the combinations that ensures that the normal basis will be used, and the one resulting in the optimal circuit.

In the design, change of basis involves a conditional branch performed over each bit of the given input, from the utmost right (LSB) to the utmost left (MSB). Depending on whether the given bit is 1 or 0, an XOR operation is performed with the element of the static matrix that corresponds to the bit position and every previous result. For example, given the input 0x0B ($0000\ 1011_2$), the resulting representation is 0xFF for the first bit examined, 0xA9 $\oplus$ 0xFF = 0x56 for the second bit, 0x56 for the third bit, which is not examined and 0x09 $\oplus$ 0x56 = 0x5F for the fourth bit, with the rest not satisfying the condition set.

2. *Invert term A to A^{-1}*, given the following equations:

$$A^{-1} = ((A_0 \otimes_{16} B^{-1}) << 4) \,//\, (A_1 \otimes_{16} B^{-1})$$

   A is split to A_1 and A_0 – the upper and lower 4 bits of the number respectively[5]. B is the expression of the number in sub-field GF(2^4). A shifter is required to place the upper half result to the correct bits.

3. *Calculate term B*, given the following equation:

$$B = N \otimes_{16} (A_1 \oplus A_0)^2 \oplus A_1 \otimes_{16} A_0.$$

   N is a value associated with the field GF(2^4) called norm, and is derived from B. The calculation of $N \otimes_{16} (A_1 \oplus A_0)^2$ is called **Square-Scaling.** Canright proves it's "cost-free" due to the use of normal basis, meaning it can be performed without additional gates. Square-Scaling is also calculated through the subfields (see **Error! Reference source not found.**, Figure 11).

4. *Calculate term B^{-1}*, given the following equation:

$$B^{-1} = ((b_0 \otimes_4 c^{-1}) << 2) \oplus (b_1 \otimes_4 c^{-1}),$$

   with b_1, b_0 corresponding to the upper and lower two-bits of the value and c being the expression of the number in sub-field GF(2^2).

5. *Calculate term c, given the following equation:*

$$c = to\ n \otimes_4 (b_0 \oplus b_1)^2 \oplus b_1 \otimes_4 b_0$$

   with n being the norm in GF(2^2). c is split to two 1-bit values, c_1 and c_0.

[4] Inversion in GF(2^2) also equals squaring, which is the term Canright uses instead.

[5] Those sub-terms are "isolated" from the initial term (the other bits will be zeroed out) with the use of bit masking. Shifting are also required to adjust the position

6. *Calculate term c^{-1}*, which is only a bit swap between c_1 and c_0:

$$c^{-1} = (c_0 << 1) \,//\, c_1,$$

7. *Change of basis of A^{-1}*, from GF(2^8)/GF(2^4)/GF(2^2) to GF(2^8), with the use of transition matrix X2S, consisting of the elements { 0x58, 0x2D, 0x9E, 0x0B, 0xDC, 0x04, 0x03, 0x24 }. The combination again ensures the normal basis will be used. In addition, it incorporates part of the affine transformation. Again the matrix results in the most optimal circuit possible.

8. Add affine constant 0x63 to the result.

The high-level code is given by Canright and was used in Vivado HLS. It should be mentioned that, while the code uses 32-bit (int) values, the operations affect only the 8 lower bits, The remaining upper length is zeroed out with bit masking. RTL synthesis would optimize out every unused wiring.

As displayed in Figure 9, the implementation requires two mapping operations G256_newbasis() and one inversion G256_inv(). The latter requires one square-scaling G16_sq_sc() instance, one G16_inv() and three multiplications G16_mul(). Those in turn will contain two inversion G4_sq() instances and one G4_scl_N2(), one G4_scl_N2(), two G4_sq() and three G4_mul(), and three G4_mul() and one G4_scl_N() respectively, as depicted in Figure 10. A more detailed view over their functionality is offered in Figure 11. The requirement of the many different instances doesn't necessarily require the same number of dedicated modules to be synthesized. Several optimizations can be applied in order to take advantage of concurrency and pipelining possibilities and benefit areas of timing and resource utilization. These are concerns examined in Chapter 5, where the Directives are further studied.

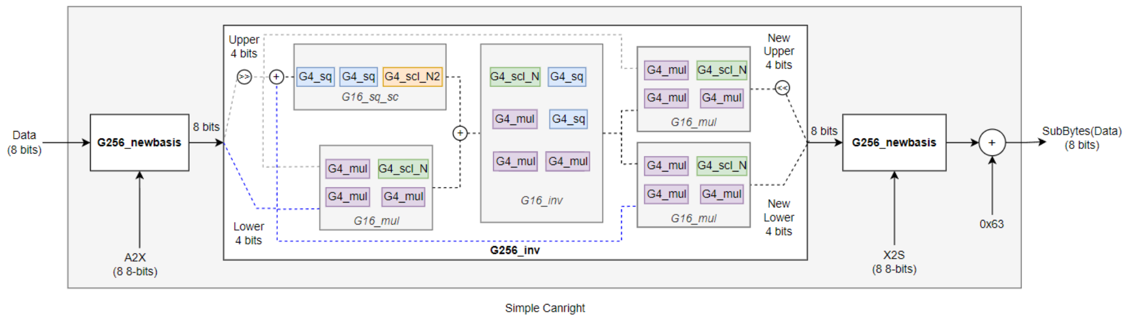


Figure 9. Simple Canright Implementation modules : Top function and sub-modules

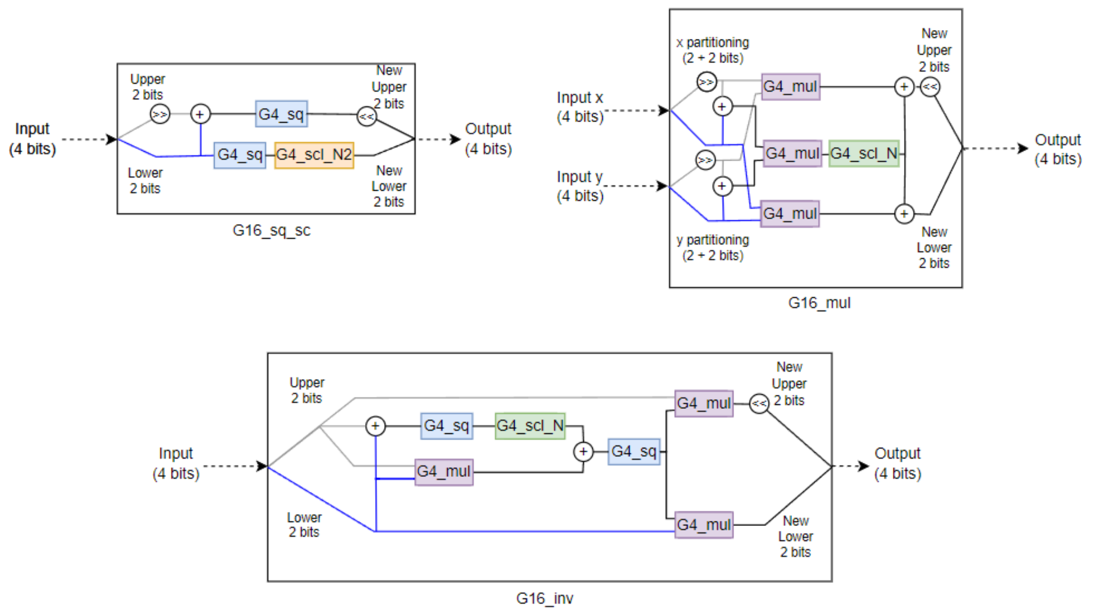


Figure 10. Simple Canright Implementation modules : G16 functions' implementation details

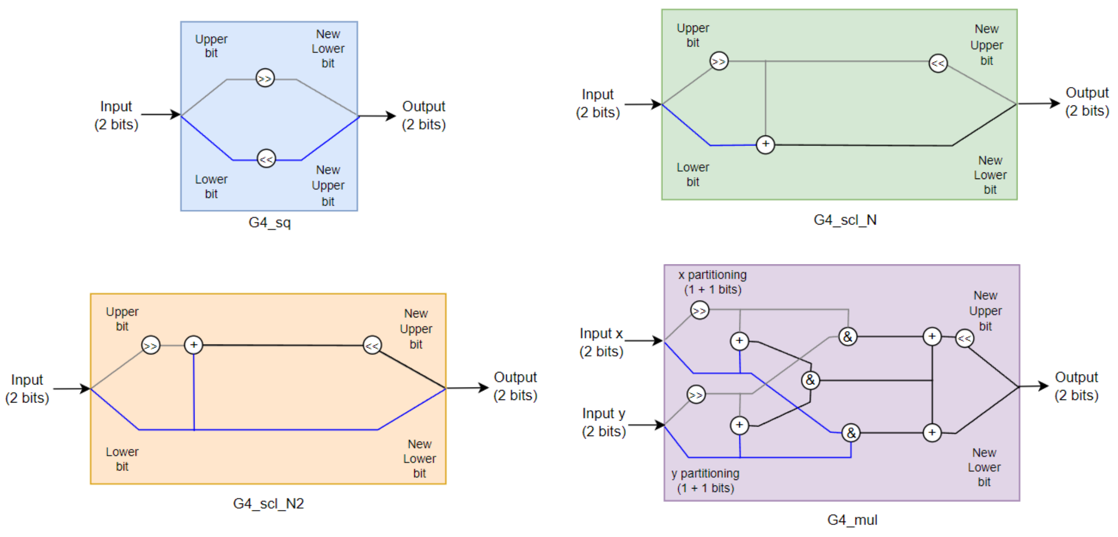


Figure 11.Simple Canright Implementation modules : G4 functions' implementation details

3.4. Secure Implementations

Each of the implementations examined in the following sections concerns four countermeasures built over Canright's Compact SBox, (simple Canright from now on, which will act as a "baseline" implementation), implemented with six different set of directives. The effect of those sets of directives results in different designs, whose behaviour will be examined through metrics provided by the Vivado HLS.

3.4.1. Countermeasures Methodology

The countermeasures discussed implement, first of all, the masking mechanism used as a defensive measure against side-channel attacks. As mentioned before, masking eliminates the dependency between sensitive information, as carried through the cryptographic operations, and the power leakage. This is achieved with the introduction of random values called masks. Moreover, it is an ideal countermeasure to use for an HLS tool, since it can be easily defined in high-level code. Masking can be performed in two ways:

- **Additive masking**, where the mask is added to the data. The addition here is Boolean, hence XOR is used. It is ideal for linear operations, where the mask is preserved as is, and so it can be removed by simply applying XOR operation with the same value.
- **Multiplicative masking**, where the mask is multiplied with the data. The multiplication in this case is polynomial. While it is costly, it can be applied over non-linear functions, unlike additive masking. Lastly, multiplicative masking is susceptible to zero-attacks, since the value zero is mapped to itself, and should be handled differently[49].

The second kind of countermeasures is identified as Correlated Noise Generation (CNG) countermeasures [50]. This logic can be used for power leakage manipulation, if the input data are properly handled. The countermeasure is examined in further detail in the following sections.

3.4.2. Masked Canright

Canright extended the idea of Compact SBox by introducing masks in his original design [51]. The theory behind this implementation is presented in [49] in an effort to develop a computationally cost-efficient implementation that would be sound against first-order side-channel attacks. As mentioned above, the SubBytes() process is a non-linear operation and additive masking is more costly when applied to it, unlike other AES functions. At the same time, multiplicative masking suffers from zero attacks and is generally too costly to be used in a lightweight implementation.

The use of tower field representation introduces linearity to the process. Specifically, inversion in GF(4), which consists of a bit swap, is a permutation operation, and thus essentially, a linear operation, making the use of additive masking more efficient.

However, only the data are essentially masked to this point. Multiple masks need to be introduced in order to conceal the intermediate computations in the tower field representation and furthermore they need to be independent to each other. In total, four independent masks are required:

- one 8-bit M_i to be used for the masking of the term A ($A \oplus M_i = A'$). M_i will be used separately as an input for the masked SubBytes().
- one 4-bit Q to be used for the masking of the term B',
- one 2-bit r to be used for the masking of the term c',
- one 4-bit T to be used for the masking of the term B^{-1}'

The masked cryptographic process will result in an 8-bit output mask S that can be used to remove the input mask from the result. It's computation is required since its bits are used in the calculations. Masked Canright's algorithm is defined below:

0. The input data of the masked SybBytes() are given masked: A' is ($A \oplus M_i$), with M_i being the input mask.

1. *Change of basis of A'*, from $GF(2^8)$ to $GF(2^8)/GF(2^4)/GF(2^2)$, with the use of A2X[6].

2. *Change of basis of M_i*, from $GF(2^8)$ to $GF(2^8)/GF(2^4)/GF(2^2)$, with the use of A2X, thus producing M.

[6] Oswald et al. suggest the masking of that (linear) operation too with another mask, but Canright skips it.

3. *Change of basis of M*, from $GF(2^8)/GF(2^4)/GF(2^2)$ to $GF(2^8)$ with the use of X2S, thus producing S.

4. *Invert element A' to A^{-1}'* . A' is split to A_1' and A_0'. A^{-1}' is defined in the following equations:

$$A^{-1}{}' = ((A_1^{-1}{}' << 4) \,//\, A_0^{-1}{}',$$

with A_1^{-1}', A_1^{-1}' calculated from:

$$A_1^{-1}{}' = S_1 \oplus A_0' \otimes_{16} B^{-1}{}' \oplus A_0' \otimes_{16} T \oplus M_0 \otimes_{16} B^{-1}{}' \oplus M_0 \otimes_{16} T$$
$$A_0^{-1}{}' = S_0 \oplus A_1' \otimes_{16} B^{-1}{}' \oplus A_1' \otimes_{16} T \oplus M_1 \otimes_{16} B^{-1}{}' \oplus M_1 \otimes_{16} T$$

M and S are is similarly split to 4-bit values M_1, M_0, S_1 and S_0.

5. *Calculate element B' from the following equation:*

$$B' = Q \oplus N \otimes_{16} (A_0' \oplus A_1')^2 \oplus N \otimes_{16} (M_1 \oplus M_0)^2 \oplus A_0' \otimes_{16} A_1' \oplus A_1' \otimes_{16} M_0 \oplus A_0' \otimes_{16} M_1 \oplus M_1 \otimes_{16} M_0$$

It is important to ensure the addition of Q is performed ahead of any other calculation concerning B. B' is split to b_1' and b_0'.

6. *Calculate element B^{-1}'* from the equation below:

$$B^{-1}{}' = ((b_1^{-1}{}' << 2) \,//\, b_0^{-1}{}',$$

with b_1^{-1}', b_0^{-1}' given from the following equations:

$$b_1^{-1}{}' = t_1 \oplus b_0' \otimes_4 c^{-1}{}' \oplus b_0' \otimes_4 r^2 \oplus q_0 \otimes_4 c^{-1}{}' \oplus q_0 \otimes_4 r^2$$
$$b_0^{-1}{}' = t_0 \oplus b_1' \otimes_4 c^{-1}{}' \oplus b_1' \otimes_4 r^2 \oplus q_1 \otimes_4 c^{-1}{}' \oplus q_1 \otimes_4 r^2$$

Q and T are is similarly split to 2-bit values q_1, q_0, t_1 and t_0. r^2 is r with its bits r_1,r_0 swapped.

7. *Calculate element c' from the following equation:*

$$c' = r \oplus n \otimes_4 (b_0' \oplus b_1')^2 \oplus n \otimes_4 (q_1 \oplus q_0)^2 \oplus b_0' \otimes_4 b_1' \oplus b_1' \otimes_4 q_0 \oplus b_0' \otimes_4 q_1 \oplus q_1 \otimes_4 q_0$$

8. *Calculate element c^{-1}'*, which is only a bit swap between c_1' and c_0'.

$$c^{-1}{}' = ((c_0^{-1}{}' << 1) \,//\, c_1^{-1}{}',$$

9. *Change of basis of A^{-1}'*, from $GF(2^8)/GF(2^4)/GF(2^2)$ to $GF(2^8)$, with the use of X2S.

10. Add affine constant 0x63 to the result, which is linear and won't affect the masking.

After those steps, A^{-1}' = ($A^{-1} \oplus$ S), S being the transformed M_i, and ($A^{-1} \oplus$ S) $\oplus$ S = A^{-1}. S will be calculated internally and won't be returned as a result, but it can be computed separately to validate the correctness of the resulting ciphertext.

As an example, the value 0x00 is given as an input to this implementation. From Figure 8, it is known that the expected result of the SubBytes(0x00) is 0x63. In this case, the four randomly generated values M_i = 0x5D, Q = 0x64, r = 0x7F and T = 0x29 are given as parameters, from which we use a specific number of bits for each mask. The value 0xCE is returned as a result, for which 0xCE $\oplus$ S needs to be equal to 0x63. S is calculated to be 0xAD, which verifies the relationship between the initial and the given data. Q, r and T don't affect the result, but alter any observed intermediate results. Masked Canright's interface is presented in Figure 12, opposite to the interface of Simple Canright's interface. Figure 13 depicts a use case of Masked Canright.

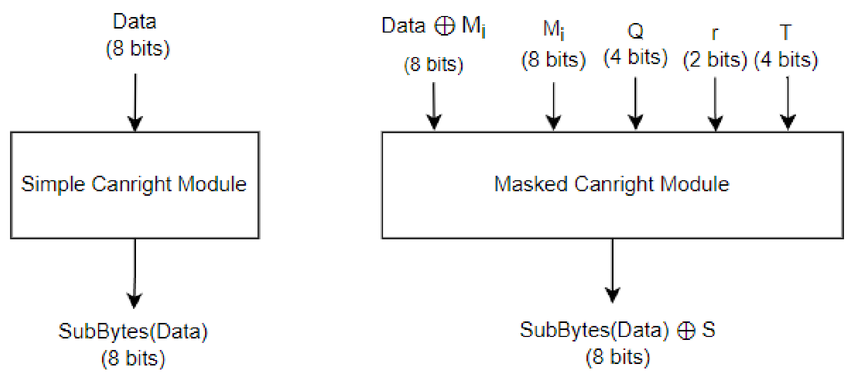


Figure 12. Masked Canright Module compared to Simple Canright.

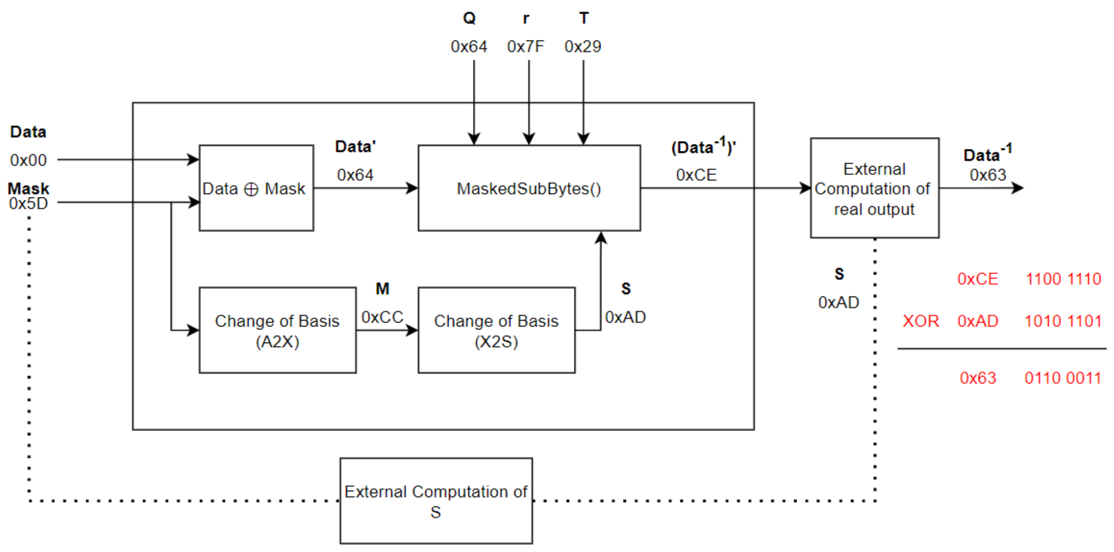


Figure 13. Use of Masked version of SubBytes. MaskedSubBytes() refers to the alternate version of Canright's algorithm that includes the masks. 'External computations' refer to computations that need to be performed outside the SubBytes module, so that no sensitive information can be derived.

Canright goes a step further and examines optimizations over the bit reusability of the masks, which in turn allows the optimization of the calculations, but those aren't examined in this thesis. The security and performance of this implementation are proved by both works mentioned.

3.4.3. CNG (First version)

The second countermeasure examined is based on the Correlation Noise Generation methodology. In this case, the same operations are performed in parallel, one using the intended data and other using data that were generated from those data. If more than one operations are performed concurrently, the resulting leakage will be an aggregation of the leakages that would result, had those operations been performed individually. This resembles the computational addition of noise, something an adversary can easily remove by means of using filters or with the extraction of a larger amount of power traces. In CNG though, the data used in the parallel computations bear a degree of correlation, which makes the extraction of sensitive data a harder task.

In this version of CNG (CNGv1), the data will use the same data path. In the C code, all values declared are of type integer (int). This means that 32 bits are bound by the design for the handling of each of those variables. 8 bits are used for the computation of SubBytes() and another 8 bits are used for its CNG.

SBox operations, as defined in simple Canright, are bitwise, meaning that if more bits are considered, they will be affected by the calculations, without themselves affecting the original result. For example, value 0x00 is extended with the correlated 0x01 value, forming the extended value 0x0100. The result of SubBytes(0x0100) then would be 0x7C63, which is a concatenation of the separate results of SubBytes(0x00) and SubBytes(0x01). The 32 bits of the integers used can accommodate a total of 1 SBOX and 3 CNGs, but the effect of a single CNG will be examined.

To apply CNG to the Canright SBOX we need perform the following changes:

1. Extend all values used in the code accordingly. If a single CNG is implemented, A2X = { 0x98, 0xF3, 0xF2, 0x48, 0x09, 0x81, 0xA9, 0xFF } should be altered to { 0x9898, 0xF3F3, 0xF2F2, 0x4848, 0x0909, 0x8181, 0xA9A9, 0xFFFF }. Similarly, X2S, the affine constant and all bit isolation masks used.
2. Given the above change, G256_newbasis() functionality should be extended to include the additional bits. For a single extension then, both the ith and the (8+i)th bits should be examined in the same iteration. This requires an additional, separate condition to be examined for the upper part of the integer value.
3. Configure the input data so they are correlated. In the example above, the upper 8 bits of the input carry a value that can be derived from the lower 8 bits of the input, if 1 is added (XORed). A common technique used is, for any given input, a true key and a fake key to be used for the SBOX and the CNG part respectively, as if the data resulted from two different AddRoundKeys operations. The resulting leakage will be an aggregation of the operations SubBytes(Data $\oplus$ TrueKey) and SubBytes(Data $\oplus$ FakeKey). CNGv1 interface is shown in Figure 14.

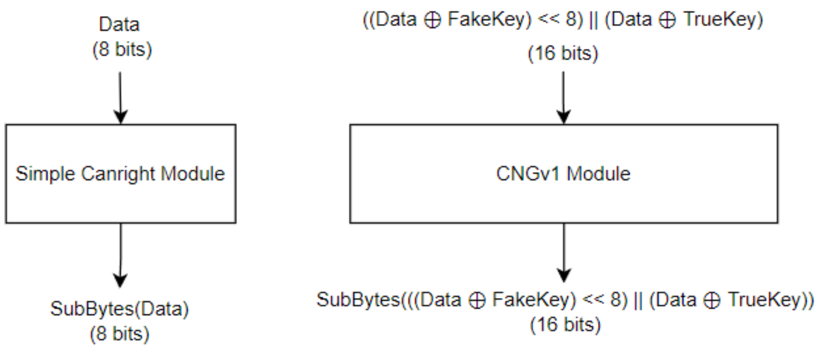

Data
(8 bits)
Simple Canright Module
SubBytes(Data)
(8 bits)
((Data ⊕ FakeKey) << 8) || (Data ⊕ TrueKey)
(16 bits)
CNGv1 Module
SubBytes(((Data ⊕ FakeKey) << 8) || (Data ⊕ TrueKey))
(16 bits)

Figure 14. CNGv1 (single extension) module compared to Simple Canright

3.4.4. CNG (Second version)

In the second version of CNG (CNGv2), the code remains the same as the one with the simple Canright. The parallel operation of SubBytes() is achieved by using the same module twice, something that can be configured from Vivado[7] environment. The alternative CNG implementation's interface is depicted in Figure 15.

[7] Vivado offers RTL-to-Bitstream transformation flow.

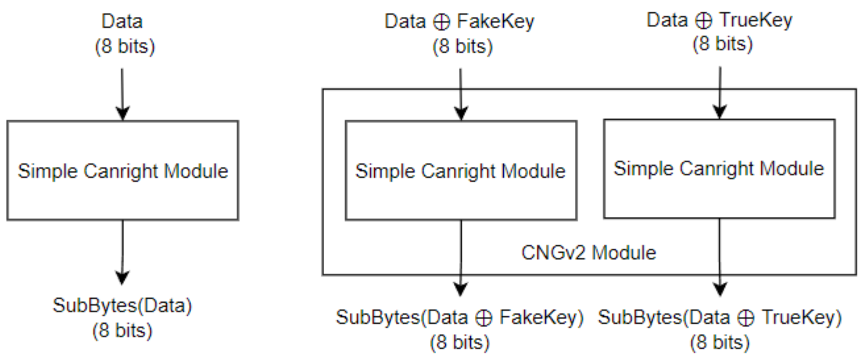


Figure 15. CNGv2 (single extension) module compared to Simple Canright

3.4.5. Combinational Countermeasure

In the final implementation the masked Canright is combined with the second version of CNG (CNGv2 + Masked Canright from now on). Two masked SubBytes() modules will be utilized instead of one, in an effort to increase the security of the implementation. The same plaintext and M_i (will result in the same M and S), Q, r, T masks will be used, but different keys. This way, the resulting leakage will correspond to the aggregation of the leakage of the masked operations SubBytes(Data ⊕ M_i ⊕ TrueKey, Mi Q, r, T) and SubBytes(Data ⊕ M_i ⊕ FakeKey, Mi Q, r, T). This design will be a good point of comparison, since it will be the most computationally complex implementation. The interface of the last implementation is presented in Figure 16.

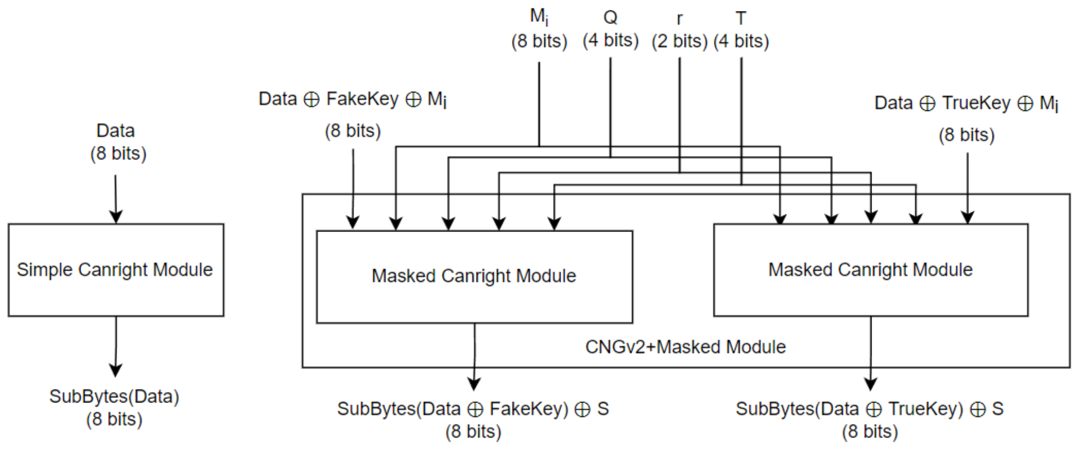


Figure 16. Combinational countermeasure (CNGv2 + Masked Canright) module compared to Simple Canright

4. Metrics

The metrics provided by Vivado HLS are estimates of the implementation's behavior, described by elements such as:

- **Timing** : The fastest clock frequency that can be achieved by the module. A threshold time value (target) can be defined in the projects settings, along with an accepted window of uncertainty. In the designs examined, those values were set to 20 ns and 2.5 ns respectively. If the estimated timing exceeds the target, the module will not be able to function at the set frequency and further optimizations shall be performed.

- **Latency** : The number of clock cycles passed since an input is given to the module until the output is produced. Minimum and maximum latency have a meaning if the code contains conditional loops containing logic that can affect the total cycles of the output generation. The metric **initialization interval (II)** may also appear in those reports. It represents the rate in clock cycles, of the module's ability to process new inputs (throughput), and is directly associated with their size. Latency can be examined in further detail:
    - For each sub-module, in order to determine each one's overhead.
    - For loops, where the value of "iteration latency" concerns the number of cycles a single iteration takes to complete. "Tripcount" depicts the number of loop iterations.
- **Resource Utilization** : Depicts the number of elements such as LUTs, Flip-Flops, BRAMs, etc, each module uses for memory, multiplexing, FIFO implementations, registers and other. Again, Resource Utilization can be viewed in further details for each sub-module. If the utilization exceeds 100%, a warning is generated.

Those metrics are derived after the Synthesis process. More accurate values can be extracted in later stages of the development, as for example, if "Vivado synthesis, place and route" evaluation option is selected when the RTL is to be exported. Yet this is a slow procedure that tackles the principle of early knowledge. Lastly, Vivado HLS only returns results about the module under examination. In the cases where two different modules are used, some assumptions were made about the Synthesis results.

5. Directives

Directives are instructions of how the compilation of the RTL is going to handle Synthesis. Their use requires a good comprehension of the code and the relationship among the elements involved, since they deal with architectural decisions. Their use though, can greatly impact the code's performance. The use of directives does not guarantee that the correctness of the functionality is retained after the changes they impose. The best practice is to review their effect manually, through C Simulation and C/RTL Co-simulation.

Directives are applied within the code in the form of *pragma definitions*, or, as Vivado HLS allows, through a specialized UI window, with each action logged in a separate .tcl file defined for each solution. Directives are applied either on data structs and code blocks such as loops or functions. The following section is dedicated to an extensive analysis of those directives, aggregating information from multiple guides [13][52][53][54][55] of the tool in use.

In Vivado HLS 2020.1, 21 directives can be applied:

- ALLOCATION : Allows the restriction of the number of hardware resources assigned for the implementation of a functionality. As a result, Synthesis imposes the sharing of the limited resources throughout the functionality, something that can benefit the area utilization at the expense of latency.
- ARRAY_MAP : Each defined array is synthesized into a separate memory element (by default, ROM for read-only arrays, RAM for read-write). This can obviously be offset in the case of small arrays. Arrays belonging in the same scope can be concatenated into a larger array, which can then be placed in a single memory element instead of separate ones, and reduce memory utilization significantly. There are two modes of mapping, with their use demonstrated at Figure 17:
    - Horizontal mapping : All given arrays are concatenated into one, with element size equal to the maximum size existing among the concatenated arrays. When offset is set, it indicates how the elements are positioned in the new array.
    - Vertical mapping : The data of the given arrays are concatenated, creating elements of larger size.

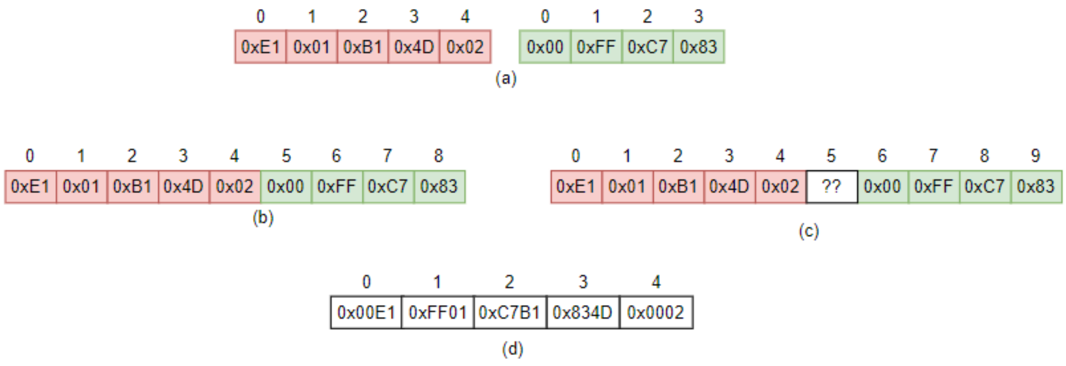

0 1 2 3 4
0xE1 0x01 0xB1 0x4D 0x02
0 1 2 3
0x00 0xFF 0xC7 0x83
(a)
0 1 2 3 4 5 6 7 8
0xE1 0x01 0xB1 0x4D 0x02 0x00 0xFF 0xC7 0x83
(b)
0 1 2 3 4 5 6 7 8 9
0xE1 0x01 0xB1 0x4D 0x02 ?? 0x00 0xFF 0xC7 0x83
(c)
0 1 2 3 4
0x00E1 0xFF01 0xC7B1 0x834D 0x0002
(d)

Figure 17. ARRAY_MAP modes example : (a) Arrays before the directive is used, (b) Horizontal ARRAY_MAP, no offset set, (c) Horizontal ARRAY_MAP with offset = 1 for the second array, (d) Vertical ARRAY_MAP

- ARRAY_PARTITION : In this case, an array is separated into more than one array, and so, more memory instances are required. While this seems counter-productive in terms of area, when more separated arrays are used, more access ports to them are created, and thus the data bandwidth increases. The number of the resulting arrays is defined by factor N. Multi-dimensional arrays can also be handled accordingly,given the dimension enumeration over which the partition is going to be performed. For example, given an array of the form A[rows][col], value 0 will partition over all dimensions, 1 will partition over the rows dimension, etc. Three modes are defined:
  - Block partitioning : The given array is split into N smaller arrays of equal size. Its use is shown at Figure 18.
  - Cyclic partitioning : The elements of the given array will alternately be placed into the partitioned arrays. Its behavior is demonstrated at Figure 19.
  - Complete partitioning : Each element of the given array is placed into a dedicated array. When multi-dimensional arrays are concerned, the register will contain the address to the array that the partitioning will create. Again, its use can be better understood through Figure 20.

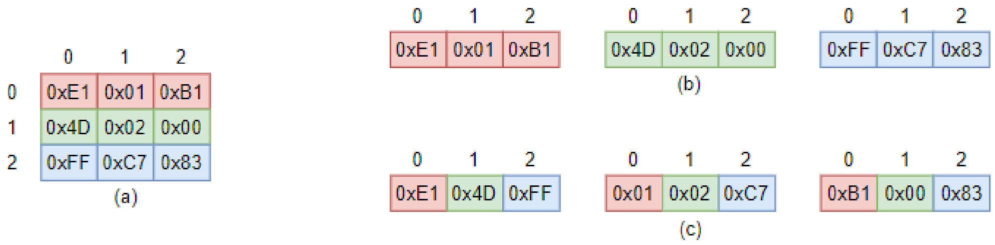


Figure 18. ARRAY_PARTITION Block example : (a) Two-dimensional array, (b) Partitioning for N=3, dim = 2, (c) Partitioning for N=3, dim = 1

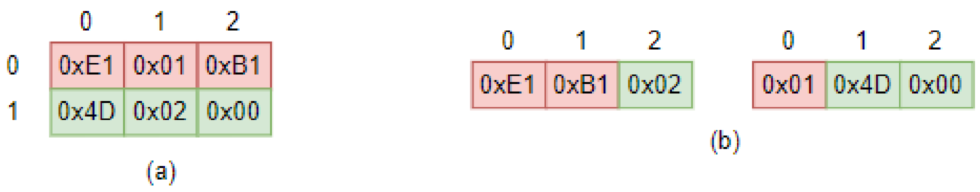


Figure 19. ARRAY_PARTITION Cyclic example : (a) Two-dimensional array, (b) Partitioning for N=2

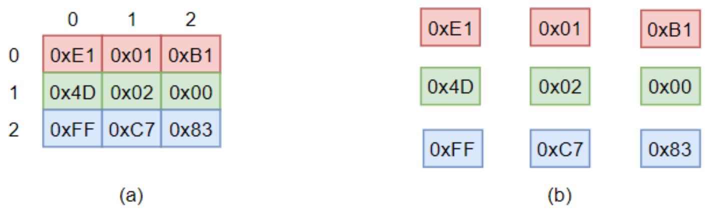


Figure 20. ARRAY_PARTITION Complete example (a) Two-dimensional array, (b) Partitioning for d=0

- ARRAY_RESHAPE : The specified array is first divided as per ARRAY_PARTITION directive and then merged vertically, creating an array "wider" elements, as in ARRAY_MAP, when vertical mapping is opted. It allows for the better utilization of the memory components of the design. Figure 21 depicts an example of the directive's use.

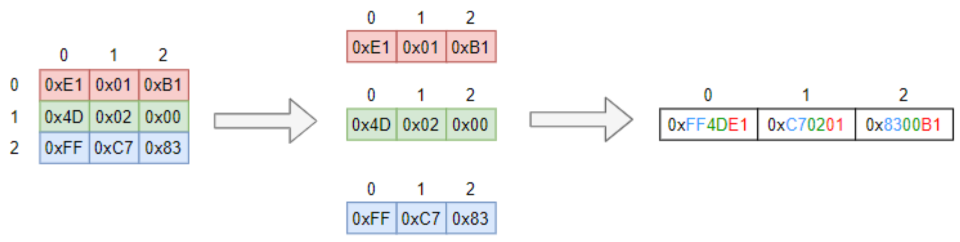
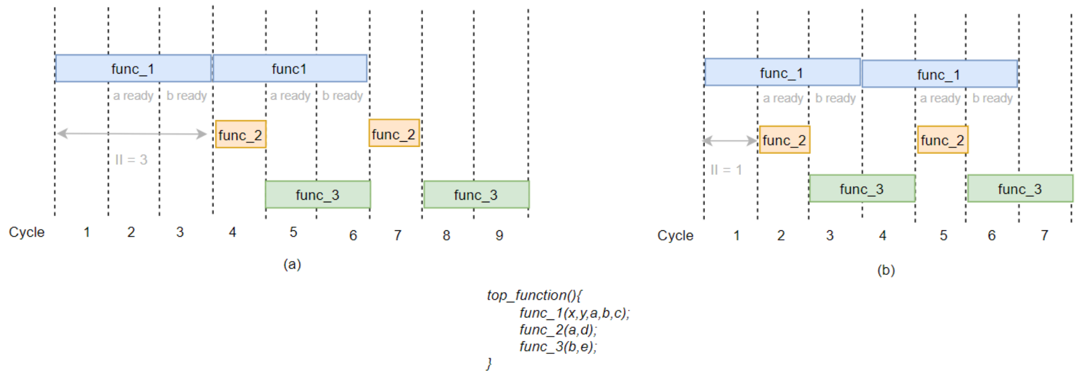


Figure 21. ARRAY_RESHAPE steps : The given array is first split into smaller arrays that can be rearranged according to the designer's needs. In this case, the directive options are : factor=3, type = block, dim=1.

- CLOCK (supported only in SystemC) : When applied to a function, it operates at a different clock frequency than the one applied to the module, thus producing/consuming data at a different rate than the rest of the design. C and C++ don't offer the corresponding functionality.
- DATAFLOW : This directive applies a form of parallelism among data-dependent functions or loops. In many cases, in order for a function to start executing, all operations over the required data need to have been finalized, even though the result of the operations may be available earlier. For example, func_1 performs a series of calculations, returning data a,b and c. Despite a and b being available before the completion of func_1, they cannot be used as inputs for func_2 and func_3 respectively. DATAFLOW allows for the use of those data at an earlier stage, saving the design some clock cycles. An example of its effect is demonstrated at Figure 22:



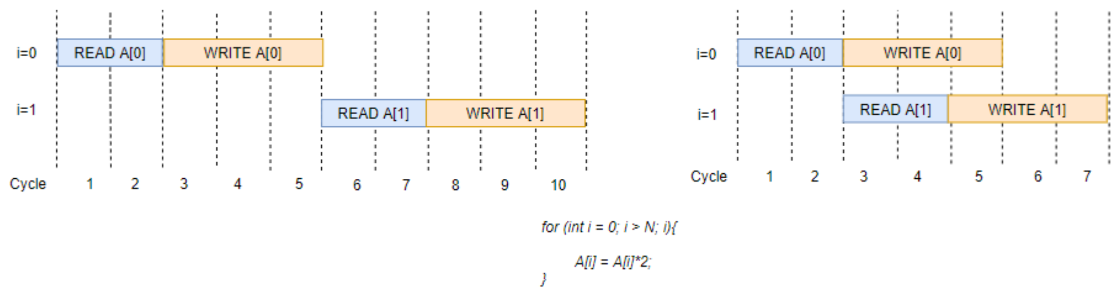
```
top_function(){
        func_1(x,y,a,b,c);
        func_2(a,d);
        func_3(b,e);
}
```

Figure 22. Dataflow example. (a) While a and b are available, they cannot be used until func_1 is finalized. (b) func2 can begin computations as soon as the data are available.

- DEPENDENCE : It is used to provide information to better handle dependencies. Vivado HLS lacks in terms of the abiltiy to correctlyidentify and resolve programming obstacles, especially when different *directions* of them exist in code, such as:
    - True dependencies : They occur when a memory location is accessed for a read operation, before a write operation is finalized (Read after Write – RAW).
    - Anti-dependencies : They occur when a memory location is accessed for a write operation, before a read operation is finalized (Write after Read – WAR).
    - Output dependencies : They occur when a memory location is accessed for a write operation, before another write operation is finalized (Write after Write– WAW).

  Specifically for loops, two additional dependencies occur:
    - Loop-independent dependence: One of the directions above occurs for a memory location within the same loop iteration. It is defined as “intra” in the directive “type” options.
    - Loop-carry dependence: One of the directions above occur for a memory location in different loop iterations. It is defined as “inter” in the directive “type” options.

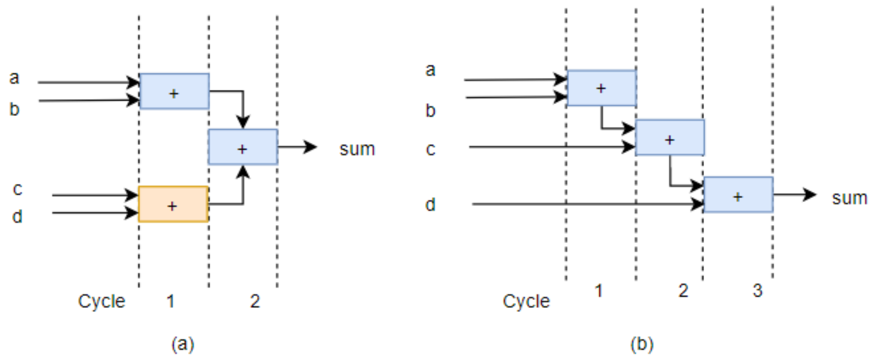
  In the case pipelining of the iterations is applied, the loop-carry dependence will affect its application. That is not the case for loop-independent dependencies though. Yet Vivado HLS treats them all the above as non-resolvable dependences. Hence, the use of DEPENDENCE directive assists the compiler to determine the nature of those dependences and ignore the “false” ones, allowing other directives to take full effect. A use case is depicted at Figure 23:



```
for (int i = 0; i > N; i){
        A[i] = A[i]*2;
}
```

Figure 23. DEPENDENCE case use in loop-independent dependency : (a) Vivado HLS assumes that WRITE A[0] and READ A[1] result in a WAR dependence, stalling the process by 3 clock cycles. (b) DEPENDENCE directive is set in this case, explicitly specifying that WAR will not occur

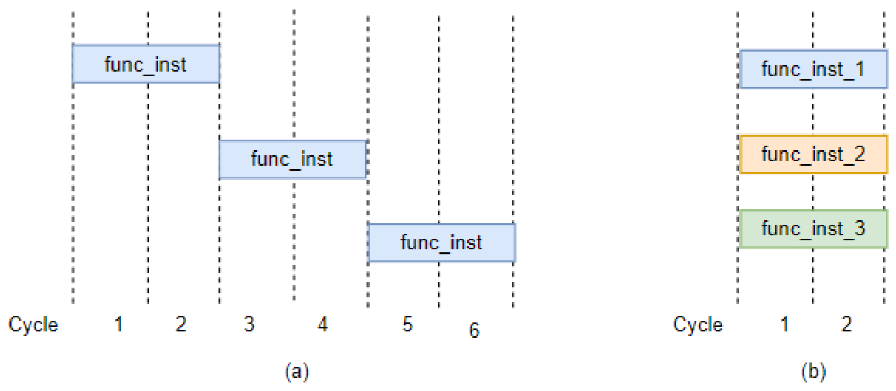
- EXPRESSION_BALANCE : By default, Vivado HLS “balances” the tree the execution creates, in order to minimize latency. The directive allows for the disabling of this behavior to minimize resource utilization. A simple example of EXPRESSION_BALANCE is provided at Figure 24:



```
sum = a + b
sum += c
sum += d
```

Figure 24. EXPRESSION_BALANCE use case : (a) represents the execution as described in the given code, as implemented by Vivado HLS (b) disabling EXPRESSION_BALANCE minimizes the required computational resources, sacrificing some time.

- FUNCTION_INSTANTIATE : This directive can be used when multiple occurrences of the same sub-function exist in a function. Synthesizing them in different instances allows them to be optimized for the different variables. Same copies of functions at the same hierarchy level are synthesized to a single -generic- block. This directive, better presented at Figure 25, assists parallelism, resulting in decreased latency and increased throughput at the expense of area. The area overhead could be very large, but combined with other area-limiting directives, performance balance can be achieved.



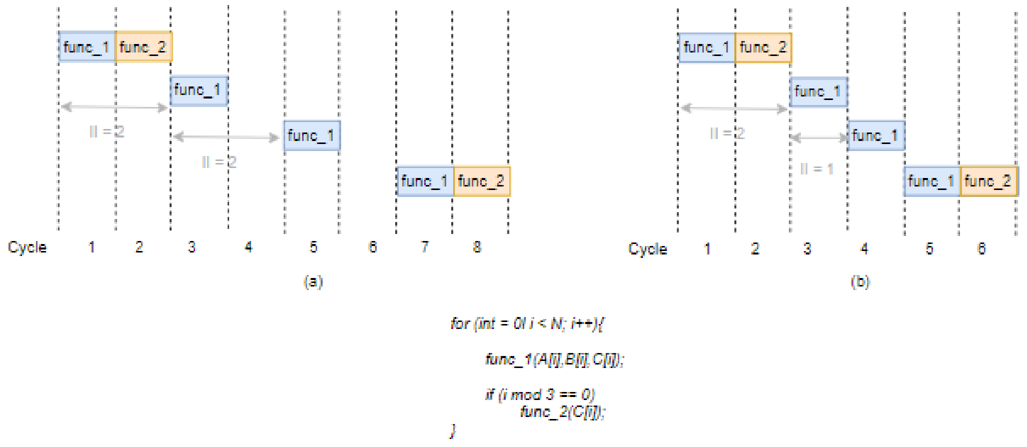
```
top_function(){
        func(a,b,return_1);
        func(return_1,c,return_2);
        func(return_1,return_2,d);
}
```

Figure 25. FUNCTION_INSTANTIATE effect : By default, the same function is synthesized to a single, generic instance. With the use of directive, three instances can be used, optimized per variables.

- INLINE : Used to remove a function from the block hierarchy. It's functionality passes to the caller function, where it may cooperate with the present functionality better. INLINE directive can also be applied recursively, inlining a series of sub-functions to their next one in the hierarchy.
- INTERFACE : Allows the specification of the way the block will communicate with other elements of the design, through the interface ports. The additional signals will define the behavior

of the synthesized block, as for example the rate it can accept new inputs. Briefly, there are protocols ideal for continuous operation, where control signals would add unnecessary delays and should not be defined (ap_none), handshake for simple (ap_ctrl_hs) and more complex (ap_ctrl_chain) implementations. More modes can be defined for ports, whether they remain stable during the computations (ap_stable), require validation (ap_ovld, for output only) or acknowledgement of reception (ap_ack), communication with memory components (ap_memory for RAMs, bram for BRAMs, ap_fifo for FIFO, ap_bus for bus interface), as well as AXI master (m_axi) and slave (s_axilite) interfaces.

- LATENCY : Sets the maximum and minimum latency constraints. If the latency of the synthesized design falls under the set minimum, the latency will be virtually increased through the use of registers. If the resulting latency exceeds the set maximum, the tool will attempt to verge towards it, applying latency reduction optimizations that do not contradict the effect of other directives.
- LOOP_MERGE : Allows the combination of the logic within the loops for which the directive is designated into a single loop. Clearly, the number of iterations of all the loops to-be-merged must be the same.
- OCCURRENCE : It is used when part of a code within a pipelined loop logic is executed conditionally. Such an *occurrence* will operate at a lower throughput rate (initialization interval**)** than the rest of the function/loop (equal to 1 when pipelined). Vivado HLS sets as the initialization interval of a code block the maximum initialization interval presented in the functionality, meaning such cases can lead to degradation of the design's performance, especially if they are rare. This directive allows the distinction of those conditional executions and the setting of a separate II for them. In Figure 26, the optimization of the direcitve is shown:



```
for (int = 0l i < N; i++){

      func_1(A[i],B[i],C[i]);

      if (i mod 3 == 0)
            func_2(C[i]);
}
```

Figure 26. OCCURENCE use in data-dependent operations : (a) Operation without the directive. Vivado HLS cannot predict when func_2 will be executed, so it extends II to accommodate its execution in all cases. (b) The directive instructs the compiler where to add the additional cycles.

- PIPELINE : Pipeline is a common parallelization technique that can dramatically increase the execution of a function/loop. Normally, in the case of hardware implementations, a code block such as a function or a loop's body binds the logic blocks involved for the whole duration of the execution. This is counter-productive, since they can be used for other computations, without affecting the running code's functionality. Its use can be affected by data dependencies, occurrences and inherently slow blocks, all of which can be optimized with the use of other directives. Its effect can be better understood, given the example at Figure 27:

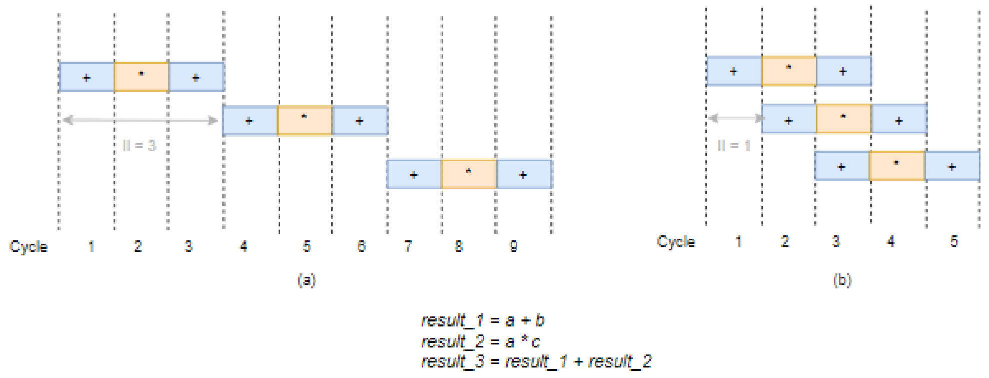
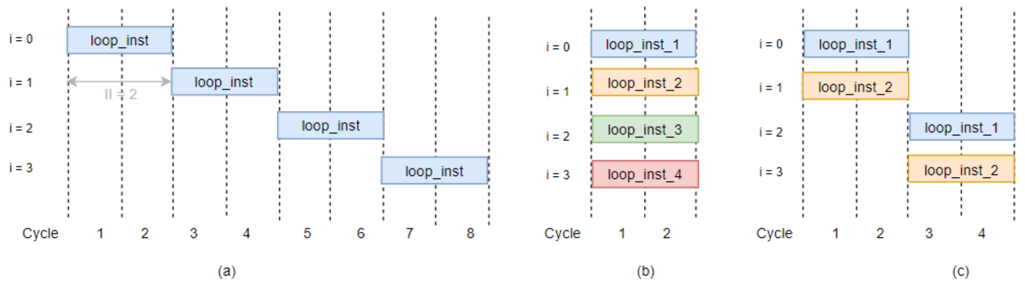


Figure 27. PIPELINE example. (a) 3 iterations of the given code executed without pipeling take 9 clock cycles, with input rate (II) of 3 cycles, while in (b) with PIPELINE set, it is executed in 5 clock cycles, with II equal to 1.

- PROTOCOL : It allows for the creation of a manual protocol for the defined code region, directing the way the resulting RTL signals should behave within its functionality. For example, commands may not carry any dependence among them, yet the order of execution shall be explicitly followed. The region can be executed in parallel with the rest of the functionality or separately, depending on the mode (floating, fixed respectively) opted. It is combined with the ap_utils.h C library, which offers helpful operations to manually define the region's behavior.
- RESOURCE : The directive is used to define the resource (core) that will be used for the synthesis of a variable or an arithmetic operation, especially when multiple definitions of cores exist in code. Examples of use are:
    - For Operators : Addition and subtraction core choice depends on the data type concerned (integer, half-precision, single-precision, double-precision), whether it is implemented with the use of pipeline technique, as well the use or not of a DSP type (DSP48, full DSP, medium DSP, no-DSP). Ten (10) different cores exist only for addition and subtraction. Other options exist for multiplication and division cores, logarithmic and exponential calculations, square root, inversion and multiplexing.
    - Storage : Three memory types are offered -FIFO, RAM and ROM-. The number of ports can be also defined, when RAMs and ROMs are concerned. The resource type can also be defined, whether it would be BRAM, Distributed RAM (LUTRAM), Ultra RAM (URAM) specifically for RAMs and Shift Registers specifically for FIFOs.
- STABLE : It can be defined for variables of any type that are input or outputs of a code region and specifies that do not need any synchronization control signals. It is applied best at read-only variable, where the data cannot be changed. When applied at data that can be written at any point of the execution, the behavior of the data should be examined thoroughly. In any case, it saves the implementation of some clock cycles that would be otherwise required for the validation of the input/output signal.
- STREAM : It implements a continuous feed of data, useful in certain design scenarios. When used in a design, it can have either of the following uses:
    - For Dataflow optimization : Allows the change of array type from the default RAM to FIFO.
    - For FIFO configuration : Allows the specification of size (depth) of the FIFO struct.
- TOP : Used to specify the top module of the design through the use of a directive. Alternatively, it can be specified through the solution's settings.
- UNROLL : Loop unrolling is another common programming technique that has become common practice because of parallelization. It allows for a loop's body to be implemented into different instances, that can run concurrently, thus decreasing considerably the latency of the design. An example of different use cases is provided in Figure 28. The level of unrolling defines:
    - Full unrolling : It is implemented when the directive instructs Synthesis to implement each iteration into a dedicated instantiation. The primary condition is that the number of

iterations is fixed. For example, while and do-while loops that examine conditions cannot be unrolled with the use of this directive. It is best applied when the iterations do not contain data dependencies among each other, but it can be very costly in terms of area.

- Partial unrolling : To balance out the area cost and latency, a specific number of copies (N) can be created, unrolling the loop by a factor N. N should be an integer, multiple of the total number of iterations required, else, optionally, the exit condition shall be checked. This allows for loops of an unknown number of iteration to be unrolled.



```
for (int = 0l i < N; i++){
        array[i] = i * 2;
}
```

Figure 28. UNROLL use : (a) loop operation without the directive (b) loop after unrolling by a factor of 4 - full unroll, (c) loop after unrolling by a factor of 2 - partial unroll

- RESET (only for data) : Allows the configuration of reset over static or global variables. It can be enabled (default) or disabled.

6. Solutions

The following segment overviews the directives examined, the way they can shape Synthesis and in extend, the resulting code. As an indication, the hierarchies of the HDL modules are presented. Before delving deeper into the solutions and their effect on the designs, a brief introduction is provided:

- **solution1** : Use of the default Vivado HLS settings for memory and functionality.
- **solution1_1_no_inline** : Use of the default Vivado HLS settings as before, with the exception of the disabling of functionality inlining.
- **solution2_loop_unroll_F8** : Application of full loop unrolling technique.
- **solution3_loop_unroll_F4** : Application of partial loop unrolling technique.
- **solution4_resource_constants** : Enforcement of BRAM use.
- **solution5 _lshr_1_all_functions** : Application of tight timing and operational resource constraints.

6.1. solution1

No directives are set in this solution, meaning Vivado HLS is synthesizing the given HLL code according to a **default strategy**. That strategy prioritizes the satisfaction of the timing constraints, then seeks the best possible throughput, by decreasing the II and the latency, and finally, attempts to minimize the resources given to the design. Consequently, the tool performs a small amount of optimizations automatically.

The first step is the formation of the top module's interface, from which the code's functionality communicates with external components. Firstly, two ports for the clock (ap_clock) and reset (ap_rst) signals are used. The arguments of the top function, as set in HLL code, as well as the return value (ap_return), if one is defined for the top function, will be synthesized into I/O ports. Input-only ports are synthesized as simple wires, while output-only require additional validation signals. By default, a handshake protocol (ap_ctrl_hs) is applied, used to allow communication with other modules when handling data, which adds the following four control signals to the interface:

- ap_start: Input signal that specifies when the block can start processing data.
- ap_done : Output signal that signifies when the block has finished its process. It serves as validation for the ap_return signal, if present.
- ap_idle : Output signal that indicates that the module does not – and will not, while it is enabled- perform any operation.
- ap_ready : Output signal that specifies when the block is ready to accept new inputs.

The examination of the code's functions follows. Generally, Vivado HLS will try to retain the hierarchy as expressed in the HLL code into the RTL, with each function synthesizing into a block. Sometimes, a function is automatically *inlined*, meaning its functionality is not synthesized into a separate block but is included in the one higher in the hierarchy. This is always the case, for example, with the function G4_sq(), which performs the bit swap in a given 2-bit value and returns the result, unless otherwise specified.

As for other code elements, loops are by default remain "rolled", and they are always executed serially. Again, small loops may be automatically unrolled. The compiler automatically opts to represent the read-only array data A2X and X2S into ROMs. Again, an automation step the default strategy takes is to use registers for the elements of small arrays, in a similar way ARRAY_PARTITION directive is used. The threshold size is set to 4, but can be configured from the settings of Vivado HLS. Structs are decomposed and at last, operators are assigned to appropriate hardcore elements.

The results of the Synthesis given any set of directives were derived from the generated VHDL code (or, if a graphic representation is preferred, from Vivado's RTL Analysis step). To better understand the effect of solution1's directives on the implementation, a paradigm is given in Figure 27. It is clear that inlining has been performed for the modules that perform simpler computations. Examining the code or simply the messages printed in the Vivado HLS console, it is understood that G256_inv module contains logic regarding G16_sq_sc and G16_inv, which in turn contain their own dedicated logic for G4_sq, G4_scl_N2 and G4_mul. Similarly, G16_mul, contains its own logic for separate G4_mul and G4_scl_N computations. It is noted that A2X_U and X2S_U are implementing the memory access to ROM components, used to store the arrays A2X and X2S, respectively. Figure 29 shows the solution's effect over Simple Canright's implementation:

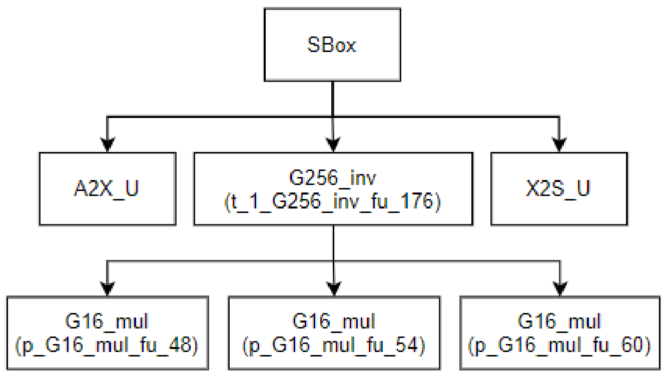


Figure 29. Hierarchy for simple Canright's implementation, given solution1. Note the naming of the modules, which are generated in a way to support debugging.

For the masked implementation the hierarchy slightly differs. The complexity of logic involved doesn't allow the level of inlining observed in simple Canright's implementation to be applied. While most functions are still inlined, both multiplications G16_mul and G4_mul (part of the inlined G16_inv and G16_sq_scl_N) are not. Moreover, the number of the required instances increases because of the correction terms calculations. Similarly, the G256_newbasis is not inlined because of its additional use for the input mask. Solution1's effect over the masked version of Sbox is depicted at Figure 30:

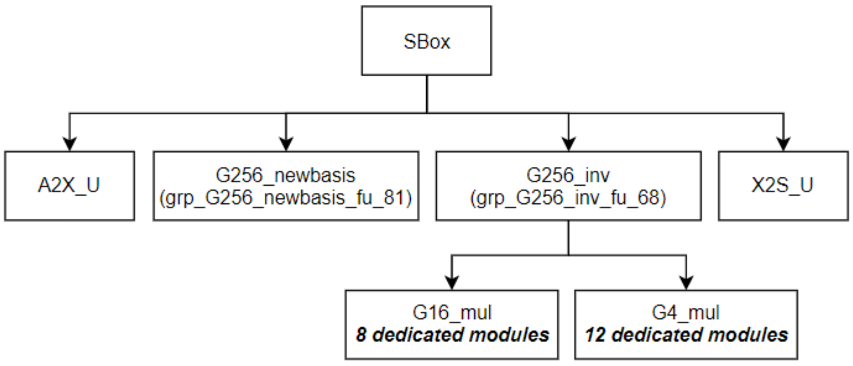

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_81)
G256_inv (grp_G256_inv_fu_68)
X2S_U
G16_mul *8 dedicated modules*
G4_mul *12 dedicated modules*

Figure 30. Hierarchy for masked Canright's implementation, given solution1

The hierarchy is altered again for CNGv1. This time two modules of G16_mul are created instead of the three for simple Canright or the eight for masked Canright. Inlining of the rest of the functionality is in effect. Lastly, solution1's result over CNGv1 can be viewed in Figure 31:

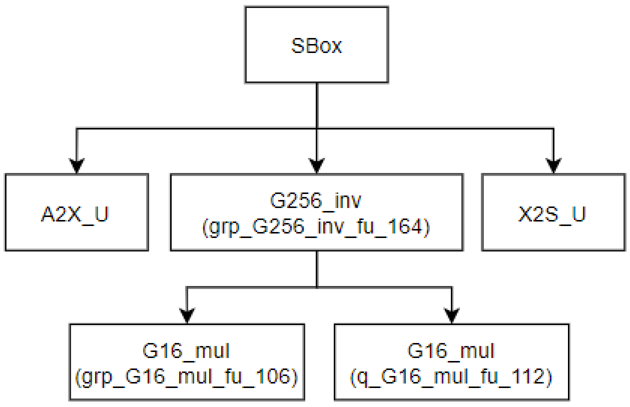

SBox
A2X_U
G256_inv (grp_G256_inv_fu_164)
X2S_U
G16_mul (grp_G16_mul_fu_106)
G16_mul (q_G16_mul_fu_112)

Figure 31. Hierarchy for CNGv1's implementation, given solution1

CNGv2 and the Masked-CNG Countermeasure use as their top modules simple Canright and masked Canright respectively, hence their hierarchies will be the same, duplicated for the two different instances they utilize. It is verified, through Vivado's "RTL Evaluation" that the elements of each module are not shared with the other.

6.2. solution1_1_no_inline

For the next solution, and as the name implies, it is strictly specified that no inlining is going to be performed for any of the design's modules. This can be achieved through the proper option of the directive INLINE. The functionality of each function is to be implemented in a dedicated module that can be shared, instead of implementing separately the required functionality on each upper-level function. Since no restrictions on the limit of modules/resources that can be used are imposed, multiple copies of those modules can be created.

Disabling inlining may result in stalls, as well as an overall bigger design. Its effect is depicted below, for simple_Canright's design: Vivado HLS "decides" to create dedicated sub_modules for G16_inv and G16_sq_scl, and share the sub_modules among the G16_mul modules. Figure 32 depicts the result of inlining disabling on simple Canright's implementation:

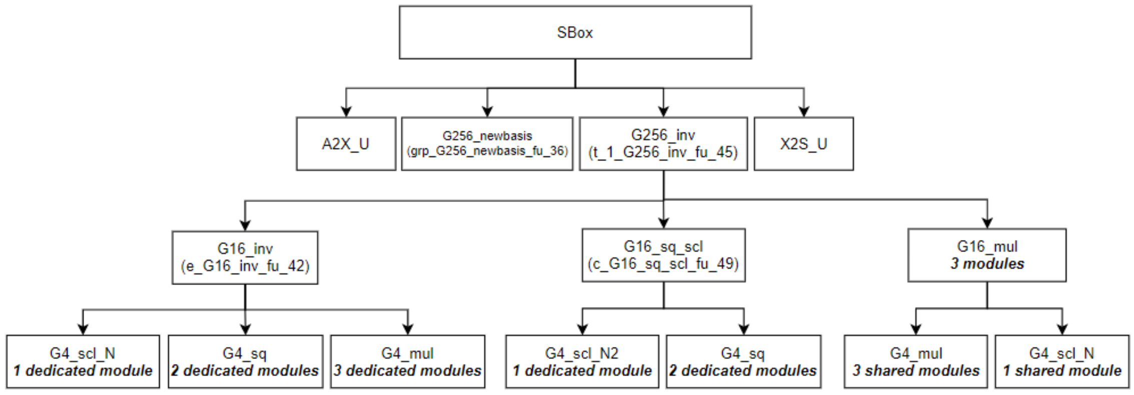

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_36)
G256_inv (t_1_G256_inv_fu_45)
X2S_U
G16_inv (e_G16_inv_fu_42)
G16_sq_scl (c_G16_sq_scl_fu_49)
G16_mul 3 modules
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 32. The resulting hierarchy for simple Canright's implementation, given solution1_1_no_inline.

For the masked implementation, while the hierarchy is the same, the modules used are increased according to the additional logic involved, as shown in Figure 33:

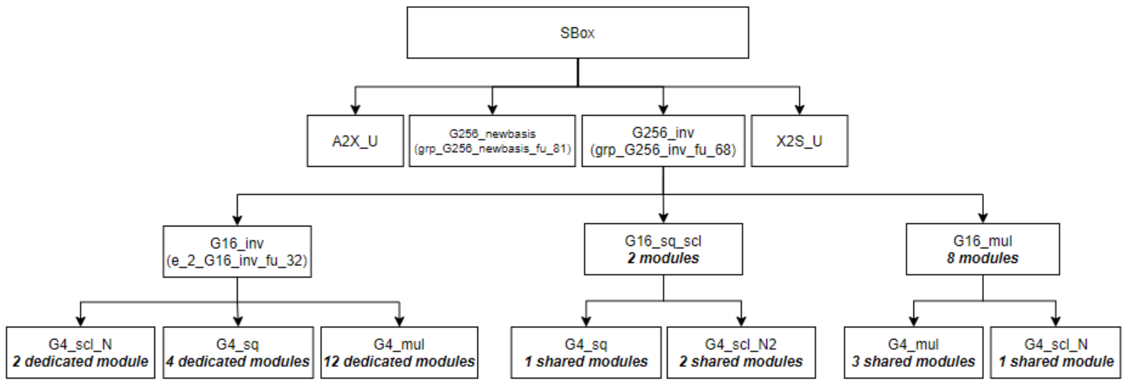

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_81)
G256_inv (grp_G256_inv_fu_68)
X2S_U
G16_inv (e_2_G16_inv_fu_32)
G16_sq_scl 2 modules
G16_mul 8 modules
G4_scl_N 2 dedicated module
G4_sq 4 dedicated modules
G4_mul 12 dedicated modules
G4_sq 1 shared modules
G4_scl_N2 2 shared modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 33. The resulting hierarchy for masked Canright's implementation, given solution1_1_no_inline.

CNGv1 hierarchy follows the same pattern, with small alterations in the number of the lower modules, as shown in Figure 34:

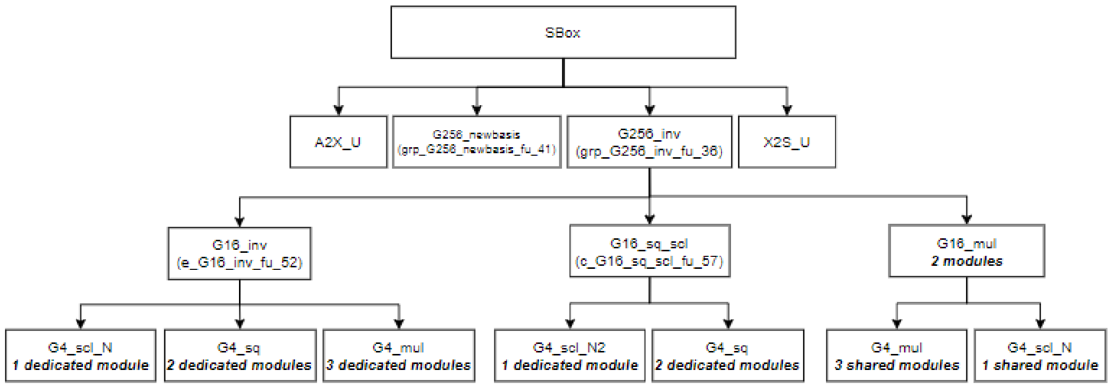

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_41)
G256_inv (grp_G256_inv_fu_36)
X2S_U
G16_inv (e_G16_inv_fu_52)
G16_sq_scl (c_G16_sq_scl_fu_57)
G16_mul 2 modules
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 34. The resulting hierarchy for CNGv1 implementation, given solution1_1_no_inline.

6.3. solution2_loop_unroll_F8

The third solution's directive concerns the G256_newbasis() function, which contains the only loop existing in the design. It consists of 8 iterations, with each one performing the needed calculations in every bit of the function's first input, in order to change the representation, either from GF(2^8) to GF(2^8)/GF(2^4)/GF(2^2) or from GF(2^8)/GF(2^4)/GF(2^2) to GF(2^8), according to the array given as the function's second input. The directive unrolls the loop by a factor 8, thus full unrolling is performed. As the hierarchy in the following figure indicates, the unrolled loop's logic is simple enough to be inlined with Sbox's logic. Generally, the resulting designs appear to be simpler compared to previous solutions. The effect of loop unrolling can be better understood from Figure 35:

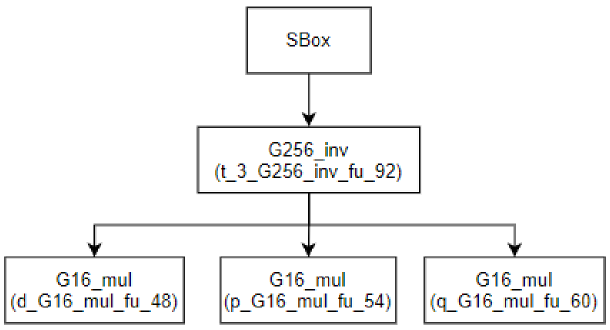


Figure 35. The resulting hierarchy for simple Canright, given solution2_loop_unroll_F8

For the masked Canright implementation, as shown in Figure 36, the multiplications over GF(2^2) are not inlined.

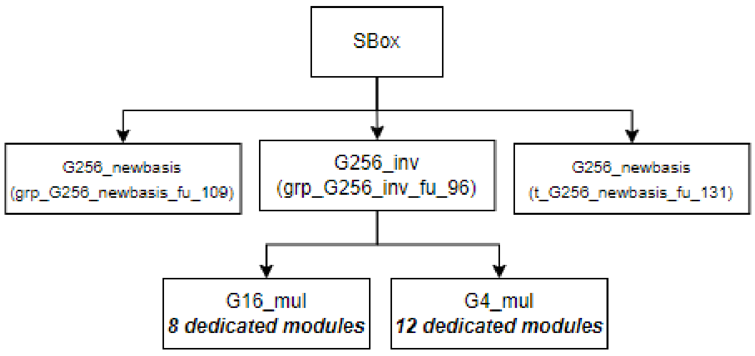


Figure 36.The resulting hierarchy for masked Canright, given solution2_loop_unroll_F8

A different hierarchy is also derived for CNGv1 with the use of loop unrolling, as shown in Figure 37:

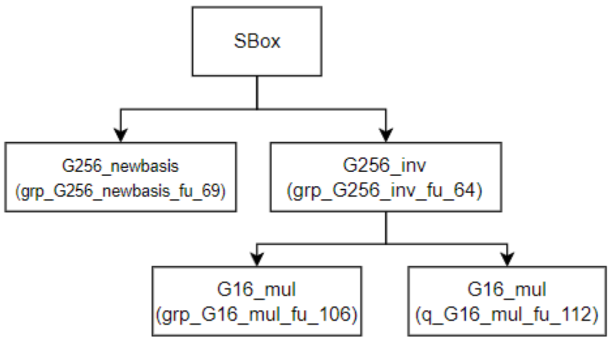


Figure 37.The resulting hierarchy for CNGv1, given solution2_loop_unroll_F8

6.4. solution3_loop_unroll_F4

In the fourth solution examined, the same directive as before is used, this time with the factor being set at 4 this time. This will cause partial unrolling of the loop involved, which may balance out the benefits and the trade-offs of full unrolling and no unrolling. The resulting hierarchy for simple Canright implementation is presented in Figure 38:

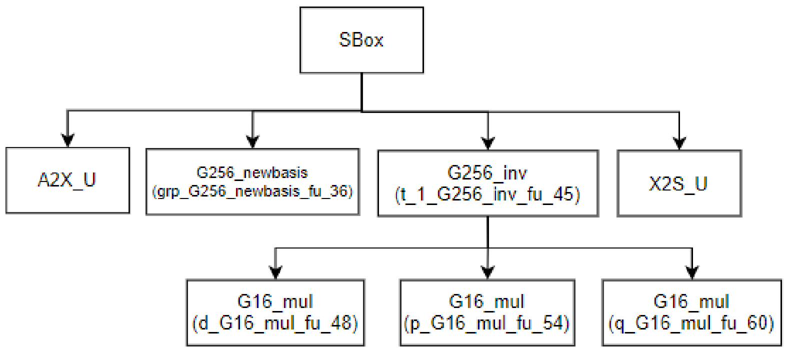


Figure 38. The resulting hierarchy for simple Canright, given solution3_loop_unroll_F4

In the case of masked Canright's implementation (Figure 30), the resulting hierarchy resembles the one presented for solution1's use, as depicted in Figure 39:

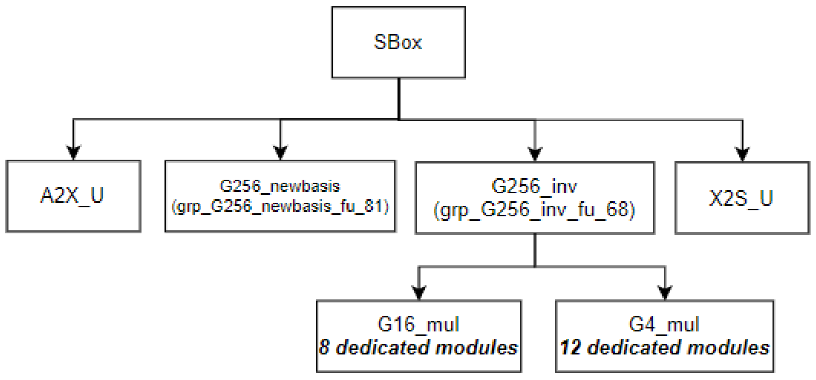


Figure 39. The resulting hierarchy for masked Canright, given solution3_loop_unroll_F4

CNGv1's hierarchy when solution3 is applied is presented in Figure 40:

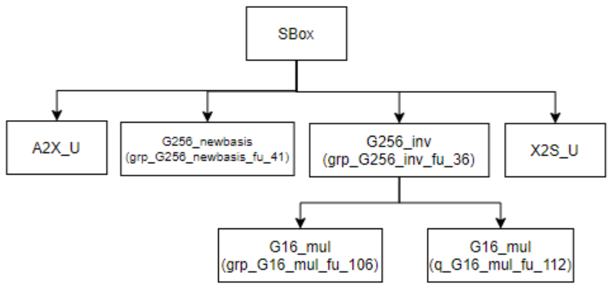

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_41)
G256_inv (grp_G256_inv_fu_36)
X2S_U
G16_mul (grp_G16_mul_fu_106)
G16_mul (q_G16_mul_fu_112)

Figure 40.The resulting hierarchy for CNGv1, given solution3_loop_unroll_F4

6.5. solution4_resource_constants

This solution combines the no inline directive for all functions, and the directive RESOURCE, which set for three arrays (A2X, X2S and y, an internal array used in G256_newbasis for the computations) the type of the core they are going to utilize. In this case, single-port BRAMs cores are defined, which are by nature limiting to the design, since simultaneous access becomes impossible. As implied by Figure 41, 3 Single-port BRAMs will appear in Synthesis reports.

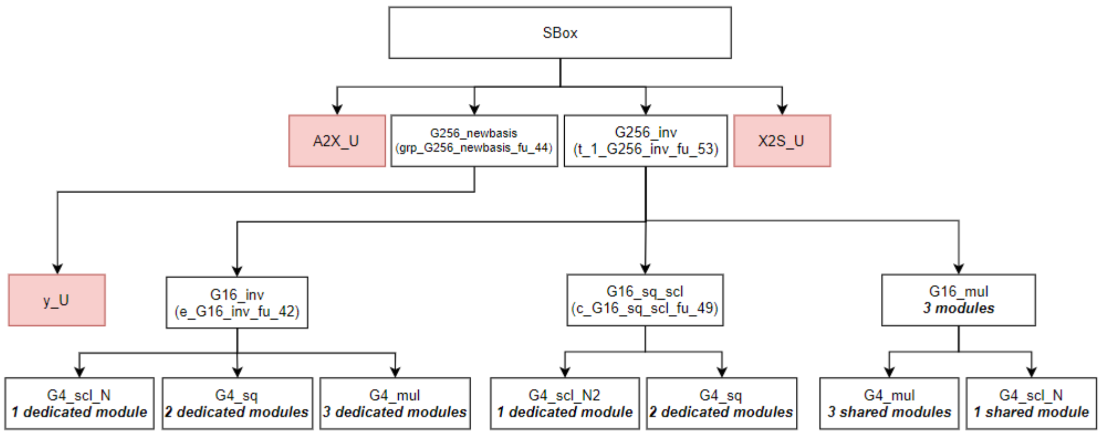

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_44)
G256_inv (t_1_G256_inv_fu_53)
X2S_U
y_U
G16_inv (e_G16_inv_fu_42)
G16_sq_scl (c_G16_sq_scl_fu_49)
G16_mul 3 modules
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 41. This design's hierarchy is similar to the one of solution1_1_no_inline's design. The red modules are the ones using the arrays A2X, X2S and y for which the directive was set.

Solution4's effect over the masked implementation of Sbox is depicted in Figure 42:

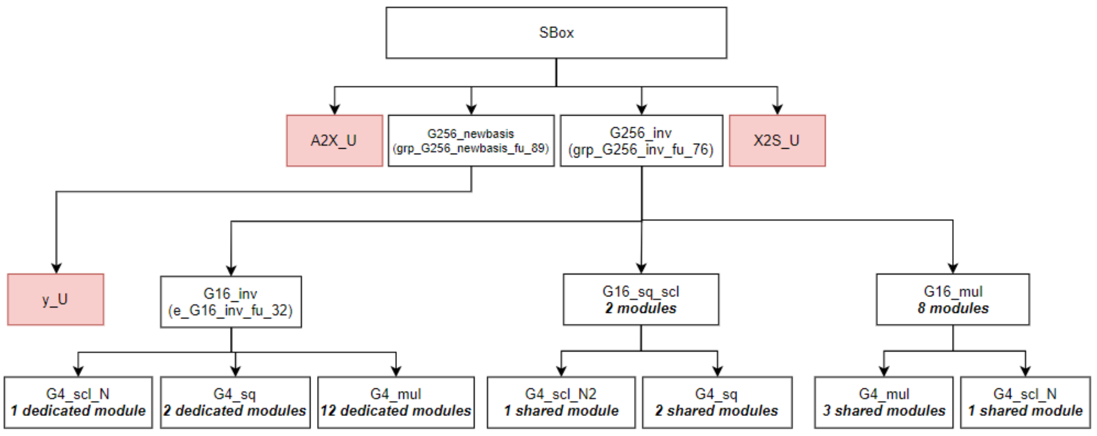

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_89)
G256_inv (grp_G256_inv_fu_76)
X2S_U
y_U
G16_inv (e_G16_inv_fu_32)
G16_sq_scl 2 modules
G16_mul 8 modules
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 12 dedicated modules
G4_scl_N2 1 shared module
G4_sq 2 shared modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 42. The resulting hierarchy for masked Canright, given solution4

Figure 43 presents the effect of the directives comprising solution4 over CNGv1 implementation:

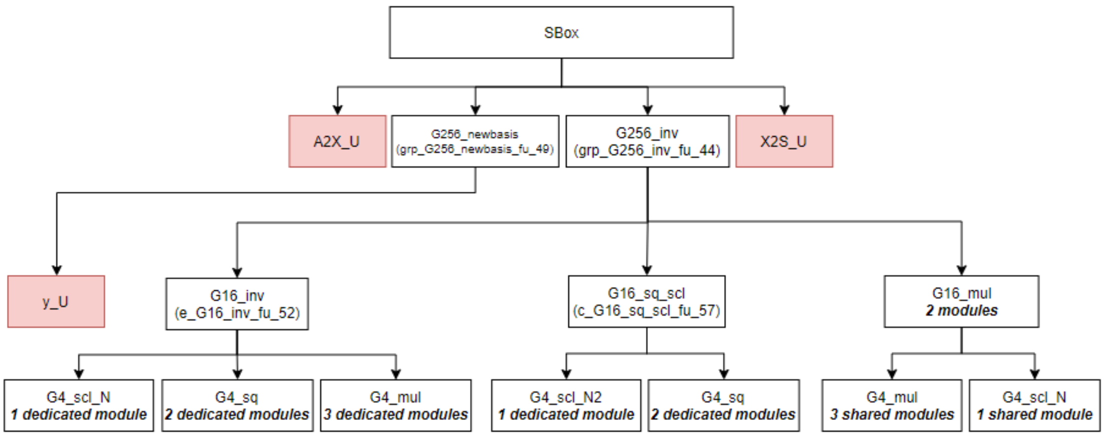

SBox
A2X_U
G256_newbasis (grp_G256_newbasis_fu_49)
G256_inv (grp_G256_inv_fu_44)
X2S_U
y_U
G16_inv (e_G16_inv_fu_52)
G16_sq_scl (c_G16_sq_scl_fu_57)
G16_mul 2 modules
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 shared modules
G4_scl_N 1 shared module

Figure 43. The resulting hierarchy for CNGv1, given solution4

6.6. solution5 _lshr_1_all_functions

In the final solution, four kinds of directives are applied to the design: a) firstly, the inlining is disabled, for all modules, b) the loop contained in G256_newbasis is fully unrolled, c) the operations of shift left, logical shift left and arithmetic shift left, as well as the use of some sub-modules, are limited to one per module. Specifically, one module of G256_newbasis and one module of G256_inv will be used for Sbox, one module of G16_mul will be used for G256_inv and one module of G4_mul will be used for G16_mul. d) Finally, the target clock for the resulting design is set to half the value it was set for the other solutions. The effect for all these directives results in the hierarchy shown in Figure 44, concerning simple Canrights' implementation:

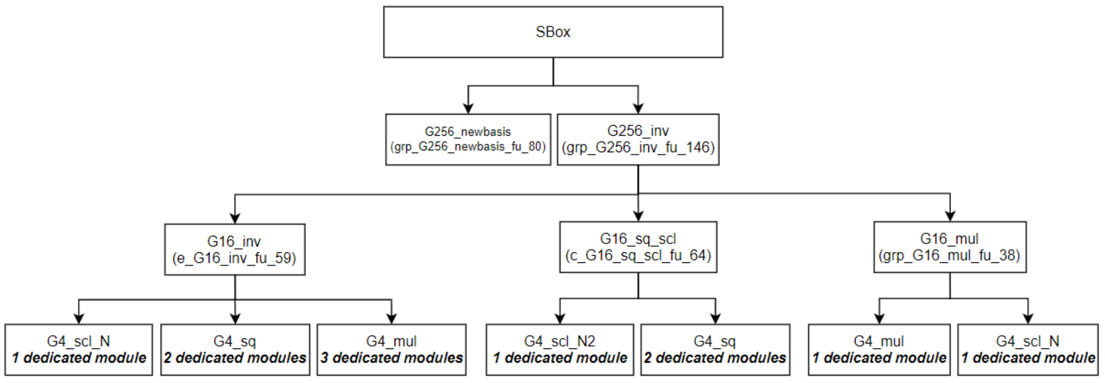

SBox
G256_newbasis (grp_G256_newbasis_fu_80)
G256_inv (grp_G256_inv_fu_146)
G16_inv (e_G16_inv_fu_59)
G16_sq_scl (c_G16_sq_scl_fu_64)
G16_mul (grp_G16_mul_fu_38)
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 1 dedicated module
G4_scl_N 1 dedicated module

Figure 44. Simple Canright design's hierarchy for solution5

Figure 45 shows the effect of solution5 over the masked implementation of Sbox:

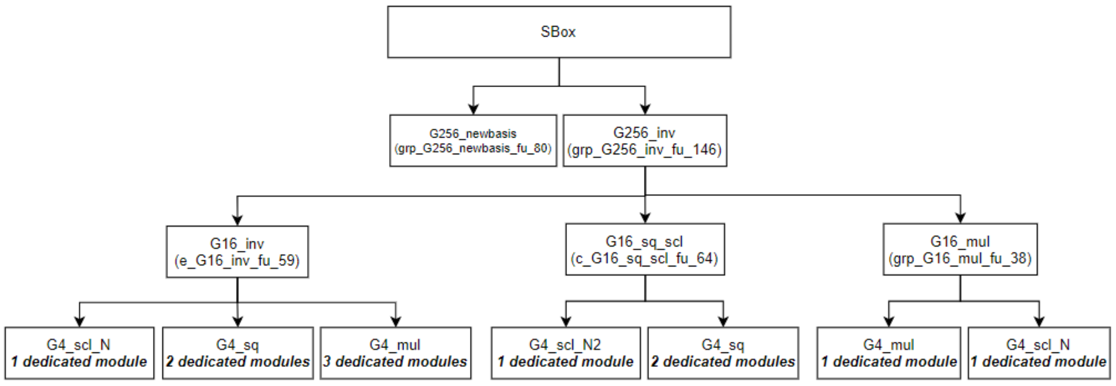

SBox
G256_newbasis (grp_G256_newbasis_fu_80)
G256_inv (grp_G256_inv_fu_146)
G16_inv (e_G16_inv_fu_59)
G16_sq_scl (c_G16_sq_scl_fu_64)
G16_mul (grp_G16_mul_fu_38)
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 3 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 1 dedicated module
G4_scl_N 1 dedicated module

Figure 45. Masked Canright design's hierarchy for solution5

The effect of the constraints enforced from solution5 over CNGv1's hierarchy is shown at Figure 46:

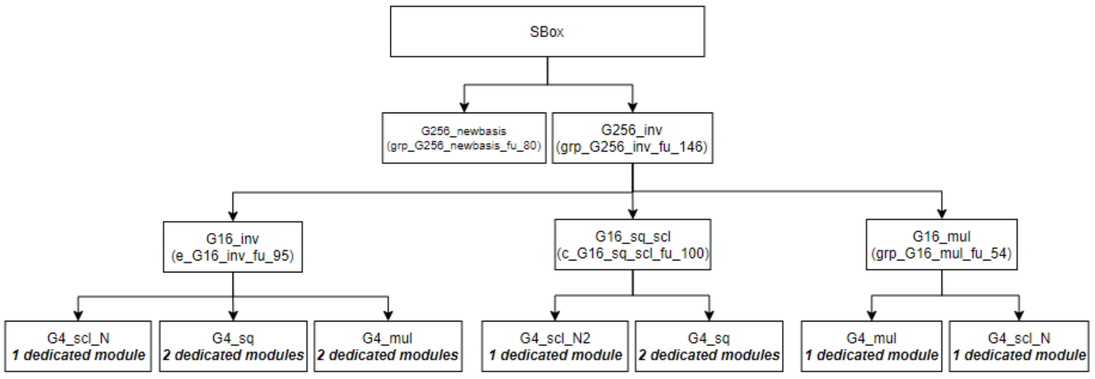

SBox
G256_newbasis (grp_G256_newbasis_fu_80)
G256_inv (grp_G256_inv_fu_146)
G16_inv (e_G16_inv_fu_95)
G16_sq_scl (c_G16_sq_scl_fu_100)
G16_mul (grp_G16_mul_fu_54)
G4_scl_N 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 2 dedicated modules
G4_scl_N2 1 dedicated module
G4_sq 2 dedicated modules
G4_mul 1 dedicated module
G4_scl_N 1 dedicated module

Figure 46. CNGv1 design's hierarchy for solution5

Lastly, tighter timing constraints have been applied for this solution. Unlike the other, which targeted a clock period of 20,00ns, this one targets a 10,00ns period. This aspect is not set as a directive, but as a solution setting.

Multiplication is a quite complex logic to implement, and is often traded for bit shifters[8]. Given that both options are limited because of the directives set, this design is expected to present greater latencies.

7. Performance analysis of Sbox Implementations

In the following sections, the results of the thirty implementations are going to be presented and compared in regards of the metrics discussed. Abbreviations will be used for the solutions presented previously.

7.1. Simple Canright

Firstly, the expected clock for the top module is examined. As shown in Table 1. Sol1 of the simple Canright's implementation serves as the baseline for the comparison among the implementations. Sol3 and Sol1 have equal minimum clock periods of 15,704 ns. Sol2 achieves a slightly lower clock period, at 15,938 ns, while sol1_1 and sol5_5 manage a minimum clock period 11% lower than the baseline's clock, with an estimated clock period of 17,500 ns. Sol11 operates at tighter timing constraints, thus it presents a smaller minimum clock period, by 46%, resulting at an estimated clock of 8,430 ns.

It appears that inlining, even when applied automatically as in the cases of sol1, sol2 and sol3 results in a faster design. Sol11 cannot be compared with the other designs, given the restriction imposed, but it could serve as a point of examining the effect of tighter timing constraints on other metrics.

Table 1. Timing estimates for simple Canright's solutions

Clock		sol1	sol1_1	sol2	sol3	sol4	sol5
ap_clk	Target	20,00 ns	20,00 ns	20,00 ns	20,00 ns	20,00 ns	10,00 ns
	Estimated	15,704 ns	17,500 ns	15,938 ns	15,704 ns	17,500 ns	8,430 ns

The next metric to be examined is latency. Table 2 depicts the values extracted by Vivado HLS tool. Min and max values have no meaning for the algorithm opted, since unbalanced conditional executing does not exist in code and can be skipped in the following latency tables. In this case, the effect of unrolling is obvious. The initial implementation requires 34 clock cycles to produce the result. Partial unrolling of sol3 by factor 4 cuts the amount of cycles in half, requiring 17 cycles to produce its result. Full unrolling taking place in sol2 results in a latency of 1 clock cycle, indicating that total parallelization has been achieved. The absolute latency then aligns with the target clock cycles, which means maximum throughput has been achieved. Again, disabling inlining, as in the cases of sol1_1, sol4 and sol5 adds delays at the designs. For example, while full unrolling is performed at sol5, its full potential does not take effect.

Table 2. Total latency estimates for simple Canright solutions

		sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	Min	34	37	1	17	37	14
	max	34	37	1	17	37	14
Latency (absolute)	Min	0,680 μs	0,740 μs	0,02 μs	0,340 μs	0,740 μs	0,140 μs
	max	0,680 μs	0,740 μs	0,02 μs	0,340 μs	0,740 μs	0,140 μs
Interval (cycles)	Min	34	37	1	17	37	14
	max	34	37	1	17	37	14

In order to better comprehend the effect of loop unrolling, the G256_newbasis loop, whether it is inlined or existing in its own module will be examined. As shown in Table 3, sol1's G256_newbasis loop presents a latency of 2 clock cycles per iteration and 16 in total, which is almost half of the total latency

[8] Shifting the bits of a number by n positions left results in the multiplication of that number by 2^n

pointed in Table 2. The inlined loop is used two times, one for converting the input value to the tower field representation and one to reverse this conversion for the output. Thus, the loops contribute to 32 out of the 34 cycles of the total latency. Any additional latency that occurs is a result of an overhead of the multiple modules being used, as depicted in Table 4. For example, in solutions sol1_1 and sol4, where inline is disabled, the loop latency is the same as sol1's since no directive affects it, yet it is enforced to be part of a dedicated module, hence the increased total latency, adding a few extra clock cycles to the resulting designs.

When the loop examined is fully unrolled, those characteristics do not exist since the loop itself doesn't exist, as observed in sol2 and sol5 cases. In sol3's case, where the loop does exist, in a different form than the original, tripcount is equal to 2 (8 iterations unrolled by a factor 4), with each of these iterations presenting a latency of 3 instead of 2, because the logic within each loop has increased, and a total of 6 instead of 16.

Table 3. Loop metrics estimates for simple Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	16	16	-	6	16	-
Iteration Latency	2	2	-	3	2	-
Tripcount	8	8	-	2	8	-

Table 4. G256_newbasis module metrics estimates for simple Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	inlined	17	Inlined	inlined	17	0

Finally, the resource utilization the synthesized designs of simple Canright's implementation will approximate is presented in Table 5. Flip-Flop (FF) utilization drops dramatically in the cases of sol2 and sol5, by 93% and 79% respectively, and for sol4 by 62%. At the same time, FFs use estimation increases for sol1_1 by 15% and for and sol3 by 32%, despite the fact that the latter applies the loop unrolling like sol2 and sol5. Sol3 also shows an increase of 54% in the number of LUTs, while the other solutions present decrease in that number, compared to sol1. There seems to be a relation to the low latency those two solutions present. A smaller amount of resources could result in a faster implementation. Lastly, it is noted the number of BRAMs used in solution sol4. As speculated previously, the directives used over the three arrays existing in code are taking effect, with the binding of 3 corresponding cores.

Table 5. Resource Utilization estimates for simple Canright solutions

	Available[9]	sol1	sol1_1	sol2	sol3	sol4	sol5
BRAM_18K	270	0	0	0	0	3	0
DSP48E	240	0	0	0	0	0	0
FF	126800	293	337	18	389	109	60
LUT	63400	599	506	446	928	446	496
URAM	0	0	0	0	0	0	0

7.2. Masked Canright

Compared to simple Canright's implementations, the majority of solutions seem to be able to perform the same (as in sol1_1 and sol4) or fewer (as in sol1, sol2, sol3) clock cycles, despite handling a more complex logic, as presented in Table 6. The case of sol2 is notable, since it displays a 9% reduction in the estimated clock. The only exception is the number of the sol5's clock cycles, which not only is 20% higher compared to simple Canright's estimated clock, but also exceeds the set threshold.

[9] For xc7a100t-ftg256-2 target device.

Comparing the solutions of the masked implementation, it is noteworthy to mention that the timing estimates follow the same pattern: Solutions that allow inlining result in a higher frequency clock. Full loop unrolling also seems to affect that number.

Table 6.Timing Estimates for Masked Canright's solutions

Clock		sol1	sol1_1	sol2	sol3	sol4	sol5
ap_clk	Target	20,00 ns	20,00 ns	20,00 ns	20,00 ns	20,00 ns	10,00 ns
	Estimated	15,371 ns	17,500 ns	14,457 ns	15,371 ns	17,500 ns	10,137 ns

The latencies of all resulting implementations of the mask application over Canright's algorithm, presented in Table 7, introduce additional logic to the design, which translates, as expected to more complex, slower designs. Comparing the solutions to their respective ones from the simple Canright version, a minimum of 42% of latency increase can be observed. Even loop unrolling cannot fully optimize the design, as noted in the case of sol2, where the optimal throughput is now 4 cycles instead of 1 due to the increased complexity of the cryptographic operation.

Table 7. Total latency estimates for masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	59	59	4	29	59	50
Latency (absolute)	1,180 μs	1,180 μs	0,08 μs	0,580 μs	1,180 μs	0,507 μs
Interval (cycles)	59	59	4	29	59	50

The loop metrics haven't been altered, since its logic hasn't been changed significantly, as presented in Table 8. Yet, the small addition in code causes Vivado HLS not to inline it automatically, as viewed in Table 9. In sol2's case, two G256_newbasis modules are created, but they do not present any latency because of unrolling

Table 8. Loop metrics estimates for masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	16	16	-	6	16	-
Iteration Latency	2	2	-	3	2	-
Tripcount	8	8	-	2	8	-

Table 9. G256_newbasis module metrics estimates for masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	18	18	0	8	18	-

As it appears in Table 10, the resource utilization for all the masked designs, compared to simple Canrights designs, has nearly doubled, with an average of 61% for FFs and 55% for LUTs. Specifically, the minimum increase is observed at the masked sol3 by 40% for FFs and 42% for LUTs, while the maximum is displayed at masked sol2, with an increase of 89% for FFs and 66% for LUTs. It is noted that both of these solutions solely perform loop unrolling, full and partial respectively. Again, BRAMs are used only for the design of sol4, as the directive instructs.

Table 10. Resource Utilization estimates for masked Canright solutions

	Available	sol1	sol1_1	sol2	sol3	sol4	sol5
BRAM_18K	270	0	0	0	0	3	0
DSP48E	240	0	0	0	0	0	0
FF	126800	583	601	169	651	373	300

LUT	63400	1206	1286	1312	1602	1226	990
URAM	0	0	0	0	0	0	0

7.3. CNG (First version)

The first version of the CNG's designs seems to be able to achieve slightly faster clocks than its simple Canright counterparts: sol1 and sol3 present a 4% fall to the clock cycles (were both 15,704 ns), sol1_1 and sol4 remain the same, sol2's clock cycles fall to 5% and sol5's fall to 10% (were 17,500 ns and 8,430 ns), while satisfying the time constraints, unlike the masked version.

Compared to the masked version, the CNGv1 implementations also result in faster clocks. This is because the logic of CNGv1 is not more complicated than any of the masked designs. On the contrary, it may be a case of better utilization of the resources, because of the parallel bitwise computation performed. Again, disabling inlining, as in cases of sol1_1 and sol4 (sol5 is not directly comparable) seems to result in a slower clock for the design. The results are depicted in Table 11:

Table 11.Timing Estimates for CNGv1's solutions

Clock		sol1	sol1_1	sol2	sol3	sol4	sol5
ap_clk	Target	20,00 ns	20,00 ns	20,00 ns	20,00 ns	20,00 ns	10,00 ns
	Estimated	15,055ns	17,500ns	15,055ns	15,055ns	17,500ns	7,565ns

The total latency of CNGv1 implementations shows a similar behavior as the designs examined previously. The inlined designs of sol1, sol2 and sol3 are better compared to functions that don't inline any functionality. Likewise, loop unrolling assists the latency reduction to a degree that is analogous to the factor of the unrolling. The resulting designs are all slightly slower than the designs of the simple Canright, by 1 to 4 clock cycles, but it would be an acceptable trade-off if the security that the countermeasure is supposed to offer was verified. The numbers are presented in Table 12.

The reason behind those extra clock cycles, especially in the cases of sol1, sol2 and sol3 may be a result of the level of inlining performed. The minor differences in G256_newbasis, viewed in Table 14, among the two algorithms (CNGv1 contains an additional condition that needs to be examined for every CNG used) results in different synthesis decisions from the tool. While the function is mostly inlined in simple Canright, its respective CNGv1 is not.

Table 12.Total latency estimates for CNGv1's solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	36	38	3	18	38	18
Latency (absolute)	0,720 µs	0,760 µs	0,06 µs	0,360 µs	0,760 µs	0,180 µs
Interval (cycles)	36	38	3	18	38	18

Table 13. Loop metrics estimates for CNGv1 designs

	sol1		sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	16	16	16	-	6	16	-
Iteration Latency	2	2	2	-	3	2	-
Tripcount	8	8	8	-	2	8	-

Table 14.G256_newbasis module metrics estimates for CNGv1 solutions

	sol1		sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	inlined	inlined	17	0	7	17	0

The resource utilization's behaviour presented in Table 15 for CNGv1 differs a lot compared to simple Canright's designs. Specifically, FFs use has risen 27% for sol1, 64% for sol2, 47% for sol4 and 70% for sl11, with a slight drop for sol1_1 (0,5%) and sol3 (13%). As for LUTs, their use resembles the numbers depicted for the masked implementation of the SBox, as they have generally doubled. Sol1's and sol1_1's LUTs have both increased by 62%, sol2's by 76%, sol3's by 49% sol4's by 67% and lastly, sol5's by 61%, compared to simple Canright.

Table 15. Resource Utilization estimates fo CNGv1 solutions

	Available	sol1	sol1_1	sol2	sol3	sol4	sol5
BRAM_18K	270	0	0	0	0	3	0
DSP48E	240	0	0	0	0	0	0
FF	126800	212	335	50	336	207	204
LUT	63400	1607	1364	1930	1848	1386	1284
URAM	0	0	0	0	0	0	0

7.4. CNG (Second version)

CNGv2, as described before, implements the CNG logic by duplicating the SBox module. Since this logic is not implemented through Vivado HLS, the estimates the tool produces are referring solely to the top module's operation, which is essentially a simple Canright's module. Therefore, the metrics are examined through the following assumptions, given that the hardware elements of each Sbox are not shared:

- The clock period of the "super-module" is the period of the slowest Sbox module. Given that Sbox is duplicated, the clock period is the same as simple Canright's one.
- Since the input is extended twice, while the clock period remains the same (presented in Table 16), the throughput will double.
- The latency of the "super-module" is the greatest of the latencies of the Sbox modules, as presented in Table 17 for the total latency. Given that there is no sharing that could lead to stalls, the latency of each design is the same one as its respective simple Canright's solution. Loop latency (Table 18) and G256_newbasis's latency (Table 19) remain the same as simple Canright's (Table 3 and 4 respectively)
- Each module is independent of one another's operation, hence the resource utilization will be doubled, as shown in Table 20.

Table 16. Timing estimates for CNGv2's solutions

Clock		sol1	sol1_1	sol2	sol3	sol4	sol5
ap_clk	Target	20,00 ns	20,00 ns	20,00 ns	20,00 ns	20,00 ns	10,00 ns
	Estimated	15,704 ns	17,500 ns	15,938 ns	15,704 ns	17,500 ns	8,430 ns

Table 17. Total latency estimates for CNGv2 solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	34	37	1	17	37	14
Latency (absolute)	0,680 μs	0,740 μs	0,02 μs	0,340 μs	0,740 μs	0,140 μs
Interval (cycles)	34	37	1	17	37	14

Table 18. Loop metrics estimates for CNGv2 solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	16	16	-	6	16	-

Iteration Latency	2	2	-	3	2	-
Tripcount	8	8	-	2	8	-

Table 19. G256_newbasis module metrics estimates for CNGv2 solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	inlined	17	Inlined	inlined	17	0
		17			17	0

Table 20. Resource Utilization estimates fo CNGv2 solutions

	Available	sol1	sol1_1	sol2	sol3	sol4	sol5
BRAM_18K	270	0	0	0	0	6	0
DSP48E	240	0	0	0	0	0	0
FF	126800	586	674	36	778	218	120
LUT	63400	1198	1012	892	1856	892	992
URAM	0	0	0	0	0	0	0

7.5. Combinational Countermeasure

Combinational countermeasure uses the idea behind CNGv2, this time employing the masked Canright module instead of the simple one. The same assumptions as before are applied once again, resulting in the respective timing estimations (Table 21), total latency (Table 22), loop and G256_newbasis latencies (Tables 23 and 24 respectively) and resource utilization (Table 25).

Table 21. Timing Estimates for CNGv2 + Masked Canright's solutions

Clock		sol1	sol1_1	sol2	sol3	sol4	sol5
ap_clk	Target	20,00 ns	20,00 ns	20,00 ns	20,00 ns	20,00 ns	10,00 ns
	Estimated	15,371 ns	17,500 ns	14,457 ns	15,371 ns	17,500 ns	10,137 ns

Table 22. Total latency estimates for CNGv2 + masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	59	59	4	29	59	50
Latency (absolute)	1,180 μs	1,180 μs	0,08 μs	0,580 μs	1,180 μs	0,507 μs
Interval (cycles)	59	59	4	29	59	50

Table 23. Loop metrics estimates for CNGv2 + masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	16	16	-	6	16	-
Iteration Latency	2	2	-	3	2	-
Tripcount	8	8	-	2	8	-

Table 24. G256_newbasis module metrics estimates for CNGv2 + masked Canright solutions

	sol1	sol1_1	sol2	sol3	sol4	sol5
Latency (cycles)	18	18	0	8	18	-

	18	18	0	8	18	-

Table 25. Resource Utilization estimates fo masked Canright solutions

	Available	sol1	sol1_1	sol2	sol3	sol4	sol5
BRAM_18K	270	0	0	0	0	3	0
DSP48E	240	0	0	0	0	0	0
FF	126800	583	601	169	651	373	300
LUT	63400	1206	1286	1312	1602	1226	990
URAM	0	0	0	0	0	0	0

7.6. Result Summary Analysis

The results analyzed previously are displayed graphically below. Figure 47 presents the aggregated results in regards of timing estimates. The horizontal axis represents the different solutions used, while the vertical axis represents the resulting values. CNGv2 and the combinational countermeasure implementations are aggregated with the results of simple Canright and masked Canright respectively, given the assumption made in the previous chapter. Figure 48 and 49 present the summarized results in regards of latency. Again, the horizontal axis represents the different solutions used, while the vertical axis represent the resulting values for clock cycles and time respectively. Again, CNGv2 and the combinational countermeasure implementations are represented through simple Canright's and masked Canright's results. Resource utilization is shown in Figure 50 for flip-flops and 51 for LUTs. The number of elements used is depited in the vertical axis. This time CNGv2 and the combinational countermeasure is represented separately, given the assumption that they operate with double the amount of elements their respective simple Canright and masked Canright implementations use.

Generally, masked Canright implementation is proved to be the most computational intensive countermeasure. CNGv1 is a better option, offering theoretically better security than the simple Canright version, for the trade-off of a greater number of LUTs used. In regards to the solutions used, sol1 (the default use of Vivado HLS) results in better results compared to some of the other solutions used, pointing out that even with little understanding of an implementation or generally the hardware's behavior, Vivado HLS can result in high-quality results.

Despite the different implementations and their requirements in area and time domains, the trends of the metrics tend to remain the same for the solutions applied. Full unrolling undoubtedly benefits the designs' efficiency as noted in the cases of sol2 and sol5, yet the factor opted should be carefully considered. Partial unrolling taking place in sol3 seems to have a negative impact in resource utilization, as shown in Figure 50, in a degree that it possibly cannot justify any advantages gained. This indicates a possible problem of directive misuse. Lastly, resource handling may offer an advantage, as displayed in sol4, but it should be combined with more directives in order to result in a better implementation.

In any case, the planning of directives is a useful tool that allows the examination of better alternatives from the very first steps of design development.

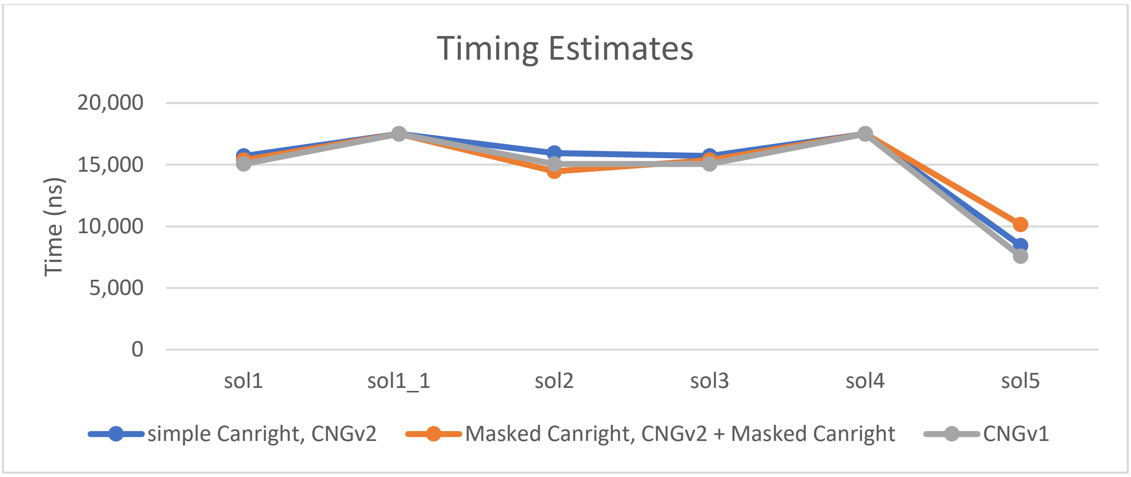


Figure 47. Timing Estimates Summary

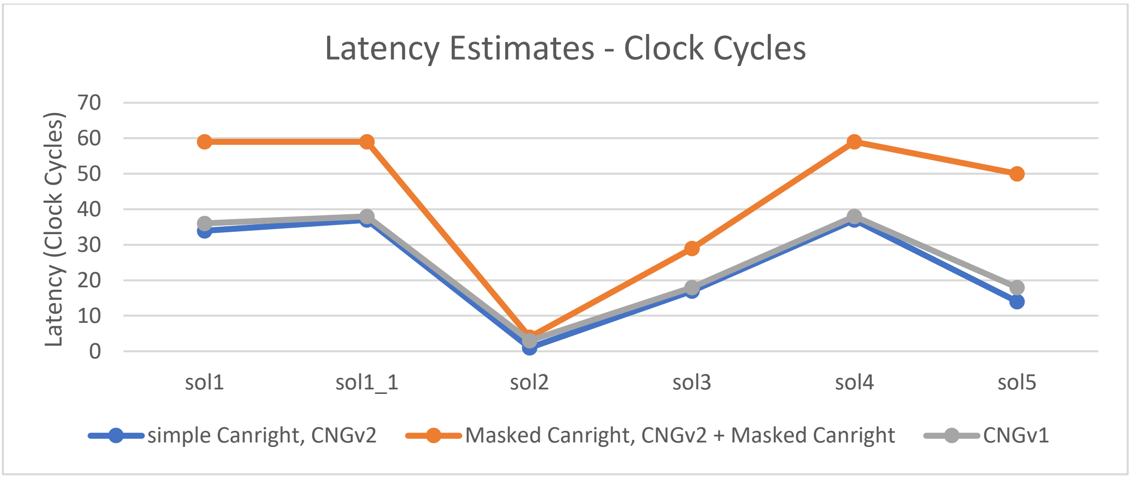


Figure 48. Latency Estimates Summary - Clock Cycles

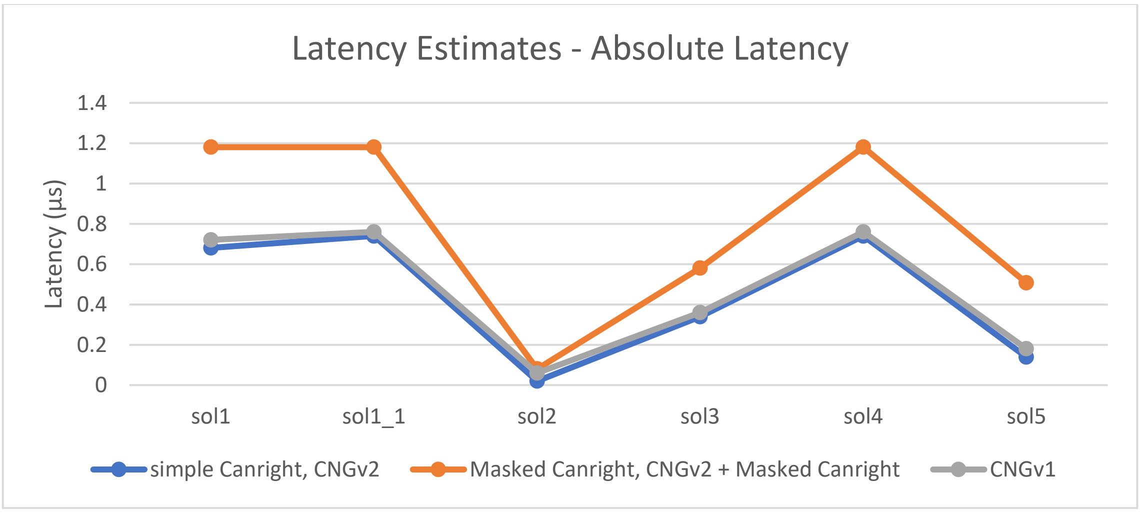


Figure 49. Latency Estimates Summary - Absolute Latency

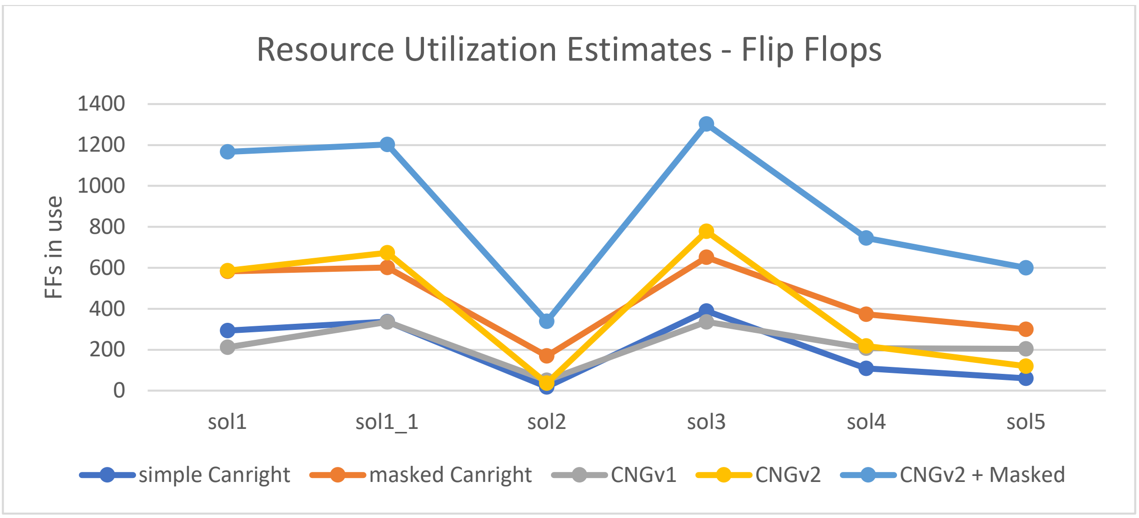


Figure 50. Resource Utilization Estimates – Flip Flops

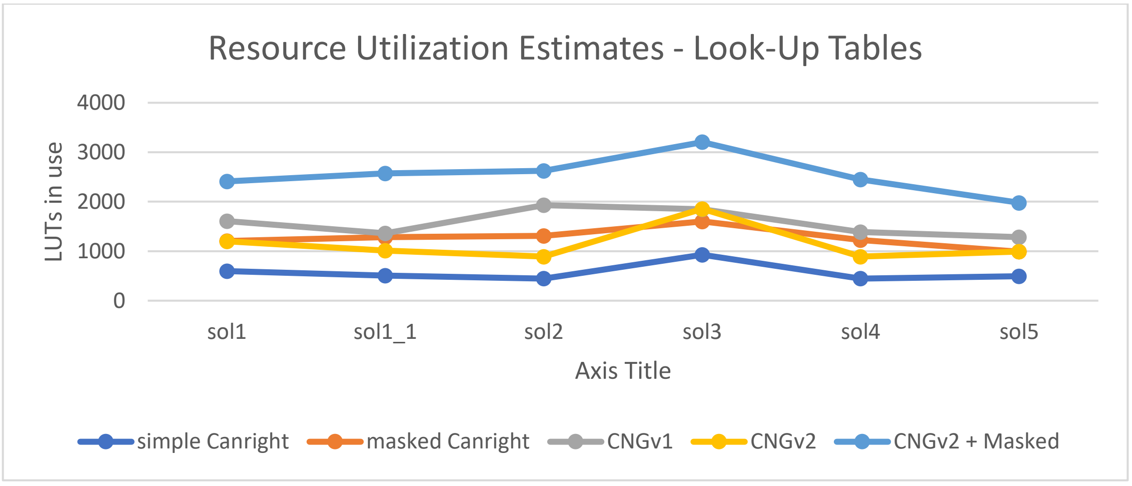


Figure 51. Resource Utilization Estimates – LUTs

8. Conclusion and Further Research

This thesis tackles the subject of examining the optimality of the generated secure designs for hardware platforms when using an HLS tool. Firstly, a brief overview of today's environment was given. Applications based on hardware devices, such as Internet of Things have seen a great rise in demand. Embedded systems can be highly capable and offer an edge on computational acceleration, yet they come with strict limitations in regards of energy consumption, response time, size and more. A "good" design for such systems has to meet those specific requirements. Therefore, the development of applications over such platforms comes with versatile needs on area and time domain, in a limited time window to meet market demands. This impacts the production process of hardware applications, calling for a change in the workflow.

Given that, the importance of HLS tools was then established. Tools such as Vivado HLS used in this thesis, have allowed the reduction of implementation times and the essential design exploration derived from those needs. Its ease of use derives from the fact that developers only need to develop the desired algorithm in a high level code to be given as an input to the HLS tool. Instead of dwelling with the complex architectural characteristics of a hardware platform, the synthesis process can be configured, using directives to control the behaviour of the generated output. Finally, the result can be assessed and verified through a number of reports at this early stage of the development process.

Next, a brief history and current status of Cryptography was given to highlight the security requirements of modern applications. Most significantly, a detailed view of an emerging family of attacks on hardware was offered. Side-channel attacks undoubtedly pose a severe threat for all those devices used in the modern digitalized environment. It is not random then that AES, the most widely used cryptographic algorithm, is extensively studied and is as well opted in this case.

An alternate version of AES, as detailed by David Canright is examined, one that fits better a hardware platform capabilities. It specifically targets SubBytes() functionality, which in software is traditionally performed through the use of costly in terms of area substitution tables. In hardware, on the other hand, it can be easily computed through an optimized form of the polynomial inversion required. Based on that implementation, the countermeasures are presented. Those involve masking, one of the most known techniques used against side-channel attacks, developed especially for the non-linear nature of the inversion involved. They moreover concern the correlated noise generation (CNG) technique, either when applied within the datapath or through multiple instantiations of the base implementation. The last countermeasure combines the two techniques, using multiple instantiations of the masked design for CNG.

A detailed documentation of the directives contained in Vivado HLS follows, in order to make the level of architectural manipulation comprehensive. Thorough examples are used to present their effect in multiple cases. Finally, six sets of directives that form the solutions opted are overviewed, along with their expected effect. Thirty implementations are examined in total, in terms of estimations given by Vivado HLS in regards of timing, latency and area utilization.

It appears that the loop unrolling technique can greatly benefit the design, while inlining, when used individually, does not. The use of BRAMs reduces the designs' utilization metrics, but their general effect on the design requires further study. It is important to mention that the default Synthesis strategy presents, in all cases examined, good results, meaning that misuse of directives can possibly impact the design's quality and should be opted after a well-thought examination. On a side note, the research solely focuses on a small yet valuable part of the workflow from concept to realization. While HLS greatly assists that process, it only offers a first look at the output design. HDL-to-bitstream steps and their possible configurations can lead to a significantly diverged design.

The next step would be to further examine the effects of more combinations of directives. Already, several papers such as [23],[56],[57] shape the direction of study over that topic, yet the research can be expanded to accommodate more diverse cases. Furthermore, HLS tool outputs need to be assessed in terms of implementation quality. Already efforts have been made to quantify the pros and cons of HLS use against HDL development, as in [58] and [59]. Especially when it comes to embedded devices, the research can be expanded in order to determine whether state-of-the-art tools can efficiently target their strict requirements. Moreover, the resulting level of security can be examined, by comparing the resistance against side-channel attacks of cryptographic algorithms generated with HLS and developed directly in HDL.

Lastly, and as mentioned before, HLS tools offer a level of abstraction that doesn't allow architectural manipulation. Yet it is known that certain hardware phenomena [60] can lead to compromisable designs against side-channel attacks. Hence it is imperative to study the security level of the final designs, whether those phenomena occur more often when HLS tools are used and in such a case, seek out a way to prevent their occurrence early in the development process.

9. Bibliography

[1] Henzinger, T.A. and Sifakis, J., 2006, August. The embedded systems design challenge. In *International Symposium on Formal Methods* (pp. 1-15). Springer, Berlin, Heidelberg.

[2] Mehta, R., Kale, S. and Utage, A.S., 2017. The internet of things (IOT) intelligence computing Technology for Home Automation. *International Journal of Current Engineering and Technology*.

[3] Lueth, K., 2020. Top 10 IoT applications in 2020 - Which are the hottest areas right now? Accessed: September 23, 2021. [online] https://iot-analytics.com/top-10-iot-applications-in-2020/.

[4] Alioto, M. ed., 2017. *Enabling the Internet of Things: From Integrated Circuits to Integrated Systems*. Springer.

[5] Aluru, S. and Jammula, N., 2013. A review of hardware acceleration for computational genomics. *IEEE Design & Test*, *31*(1), pp.19-30.

[6] Transforma Insights. ,2020. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030 (in billions). *Statista*. Statista Inc. Accessed: September 23, 2021. [online] https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[7] Poschmann, A., 2009. Lightweight Cryptography. *Ruhr-University Bochum, Bochum*.

[8] Renauld, M., Standaert, F.X. and Veyrat-Charvillon, N., 2009, September. Algebraic side-channel attacks on the AES: Why time also matters in DPA. In *International Workshop on Cryptographic Hardware and Embedded Systems* (pp. 97-111). Springer, Berlin, Heidelberg.

[9] Rupnow, K., Liang, Y., Li, Y. and Chen, D., 2011, October. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC* (pp. 1102-1105). IEEE.

[10] Elliott, J.P., 1999. *Understanding behavioral synthesis: a practical guide to high-level design*. Springer Science & Business Media.

[11] Martin, G. and Smith, G., 2009. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, *26*(4), pp.18-25.

[12] Nane, R., Sima, V.M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F. and Anderson, J., 2015. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(10), pp.1591-1604.

[13] Xilinx, V.H., 2021. Vivado Design Suite User Guide : High-Level Synthesis (v2020.1).

[14] Menezes, A.J., Van Oorschot, P.C. and Vanstone, S.A., 2018. *Handbook of applied cryptography*. CRC press.

[15] Shannon, C.E., 1945. A mathematical theory of cryptography. Originally published in: Shannon. *CE (1949)," Communication theory of secrecy systems". Bell System Technical Journal*, *28*, pp.656-715.

[16] Shannon, C.E., 1949. Communication theory of secrecy systems. *The Bell system technical journal*, *28*(4), pp.656-715.

[17] Meurer, R.S., Mück, T.R. and Fröhlich, A.A., 2013, December. An Implementation of the AES cipher using HLS. In *2013 III Brazilian Symposium on Computing Systems Engineering* (pp. 113-118). IEEE.

[18] Silitonga, A., Schade, F., Jiang, G. and Becker, J., 2018, December. Hls-based performance and resource optimization of cryptographic modules. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)* (pp. 1009-1016). IEEE.

[19] Homsirikamol, E. and Gaj, K., 2014, December. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)* (pp. 1-8). IEEE.

[20] Socha, P., Miškovský, V. and Novotný, M., 2021. High-level synthesis, cryptography, and side-channel countermeasures: A comprehensive evaluation. *Microprocessors and Microsystems*, *85*, p.104311.

[21] Konigsmark, S.C., Chen, D. and Wong, M.D., 2017, July. High-Level Synthesis for side-channel defense. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (pp. 37-44). IEEE.

[22] Zhang, L., Mu, D., Hu, W. and Tai, Y., 2020. Machine-learning-based side-channel leakage detection in electronic system-level synthesis. *IEEE Network*, *34*(3), pp.44-49.

[23] Zhang, L., Mu, D., Hu, W., Tai, Y., Blackstone, J. and Kastner, R., 2019. Memory-based high-level synthesis optimizations security exploration on the power side-channel. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *39*(10), pp.2124-2137.

[24] Basu, K., Soni, D., Nabeel, M. and Karri, R., 2019. NIST Post-Quantum Cryptography-A Hardware Evaluation Study. *IACR Cryptol. ePrint Arch.*, *2019*, p.47.

[25] Standaert, F.X., 2010. Introduction to side-channel attacks. In *Secure integrated circuits and systems* (pp. 27-42). Springer, Boston, MA.

[26] Kevorkian, C. and Tanenbaum, J.B., 2011. *Advanced Cryptographic Power Analysis* (Doctoral dissertation, Worcester Polytechnic Institute).

[27] Randolph, M. and Diehl, W., 2020. Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. *Cryptography*, *4*(2), p.15.

[28] Mangard, S., Oswald, E. and Popp, T., 2008. *Power analysis attacks: Revealing the secrets of smart cards* (Vol. 31). Springer Science & Business Media.

[29] Kocher, P., Jaffe, J. and Jun, B., 1999, August. Differential power analysis. In *Annual international cryptology conference* (pp. 388-397). Springer, Berlin, Heidelberg.

[30] Gierlichs, B., Batina, L., Preneel, B. and Verbauwhede, I., 2010, March. Revisiting higher-order DPA attacks. In *Cryptographers' Track at the RSA Conference* (pp. 221-234). Springer, Berlin, Heidelberg.

[31] Daemen, J. and Rijmen, V., 1999. AES proposal: Rijndael.

[32] Blazhevski, D., Bozhinovski, A., Stojchevska, B. and Pachovski, V., 2013. Modes of operation of the AES algorithm.

[33] Almuhammadi, S. and Al-Hejri, I., 2017, April. A comparative analysis of AES common modes of operation. In *2017 IEEE 30th Canadian conference on electrical and computer engineering (CCECE)* (pp. 1-4). IEEE.

[34] Stallings, W., 2006. *Cryptography and network security, 4/E*. Pearson Education, Ltd., London.

[35] www.samiam.org, Sam Trenholme's webpage. Accessed: September 24, 2021 [online] https://www.samiam.org/key-schedule.html

[36] Commonlounge.com. ,2014. The Advanced Encryption Standard (AES) Algorithm | CommonLounge. Accessed: September 24, 2021 [online] https://www.commonlounge.com/discussion/e32fdd267aaa4240a4464723bc74d0a5

[37] El-Haii, M., Chamoun, M., Fadlallah, A. and Serhrouchni, A., 2018, October. Analysis of cryptographic algorithms on iot hardware platforms. In *2018 2nd Cyber Security in Networking Conference (CSNet)* (pp. 1-5). IEEE.

[38] Chari, S., Jutla, C., Rao, J.R. and Rohatgi, P., 1999, March. A cautionary note regarding evaluation of AES candidates on smart-cards. In *Second Advanced Encryption Standard Candidate Conference* (pp. 133-147).

[39] Bogdanov, A., Khovratovich, D. and Rechberger, C., 2011, December. Biclique cryptanalysis of the full AES. In *International conference on the theory and application of cryptology and information security* (pp. 344-371). Springer, Berlin, Heidelberg.

[40] Bonnetain, X., Naya-Plasencia, M. and Schrottenloher, A., 2019. Quantum security analysis of AES. *IACR Transactions on Symmetric Cryptology*, *2019*(2), pp.55-93.

[41] Balasch, J., Gierlichs, B., Reparaz, O. and Verbauwhede, I., 2015, September. DPA, bitslicing and masking at 1 GHz. In *International Workshop on Cryptographic Hardware and Embedded Systems* (pp. 599-619). Springer, Berlin, Heidelberg.

[42] Kocher, P., Jaffe, J., Jun, B. and Rohatgi, P., 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, *1*(1), pp.5-27.

[43] Ors, S.B., Gurkaynak, F., Oswald, E. and Preneel, B., 2004, April. Power-analysis attack on an ASIC AES implementation. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.* (Vol. 2, pp. 546-552). IEEE.

[44] Caltagirone, C. and Anantha, K., 2003, June. High throughput, parallelized 128-bit AES encryption in a resource-limited FPGA. *In Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures* (pp. 240-241).

[45] Prouff, E., 2005, February. DPA attacks and S-boxes. In *International Workshop on Fast Software Encryption* (pp. 424-441). Springer, Berlin, Heidelberg.

[46] Dogan, A., Ors, S.B. and Saldamli, G., 2014. Analyzing and comparing the AES architectures for their power consumption. *Journal of Intelligent Manufacturing*, 25(2), pp.263-271.

[47] Canright, D., 2005. *A very compact Rijndael S-box," Naval Postgraduate School Technical Report: NPS-MA-05-001*. Tech. Rep.

[48] Paar, C., 1994. Efficient VLSI architectures for bit-parallel computation in Galois fields. *PhD Thesis, Inst. for Experimental Math., Univ. of Essen*.

[49] Oswald, E., Mangard, S., Pramstaller, N. and Rijmen, V., 2005, February. A side-channel analysis resistant description of the AES S-box. In *International workshop on fast software encryption* (pp. 413-423). Springer, Berlin, Heidelberg.

[50] Kamoun, N., Bossuet, L. and Ghazel, A., 2009, November. Correlated power noise generator as a low cost DPA countermeasures to secure hardware AES cipher. *In 2009 3rd International Conference on Signals, Circuits and Systems (SCS)* (pp. 1-6). IEEE.

[51] Canright, D. and Batina, L., 2008, June. A very compact "perfectly masked" S-box for AES. In *International Conference on Applied Cryptography and Network Security* (pp. 446-459). Springer, Berlin, Heidelberg.

[52] Xilinx, V.H., 2014. Vivado design suite user guide-high-level synthesis (v2014.1).

[53] Xilinx, V.H., 2017. Vivado HLS Optimization Methodology Guide (v2017.4).

[54] Xilinx, V.H., 2018. Vivado design suite user guide-high-level synthesis (v2017.4).

[55] Xilinx, V.H., 2019. SDx Pragma Reference Guide (v2019.1).

[56] Georgopoulos, K., Chrysos, G., Malakonakis, P., Nikitakis, A., Tampouratzis, N., Dollas, A., Pnevmatikatos, D. and Papaefstathiou, Y., 2016, September. An evaluation of vivado HLS for efficient system design. In *2016 International Symposium ELMAR* (pp. 195-199). IEEE.

[57] Choi, Y.K. and Cong, J., 2018, November. HLS-based optimization and design space exploration for applications with variable loop bounds. *In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (pp. 1-8). IEEE.

[58] Huang, Q., Lian, R., Canis, A., Choi, J., Xi, R., Brown, S. and Anderson, J., 2013, April. The effect of compiler optimizations on high-level synthesis for FPGAs. *In 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines* (pp. 89-96). IEEE.

[59] Dai, S., Zhou, Y., Zhang, H., Ustun, E., Young, E.F. and Zhang, Z., 2018, April. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. *In 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 129-132). IEEE.

[60] Faust, S., Grosso, V., Pozo, S.M.D., Paglialonga, C. and Standaert, F.X., 2018. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018, 3*(pp. 89-120).