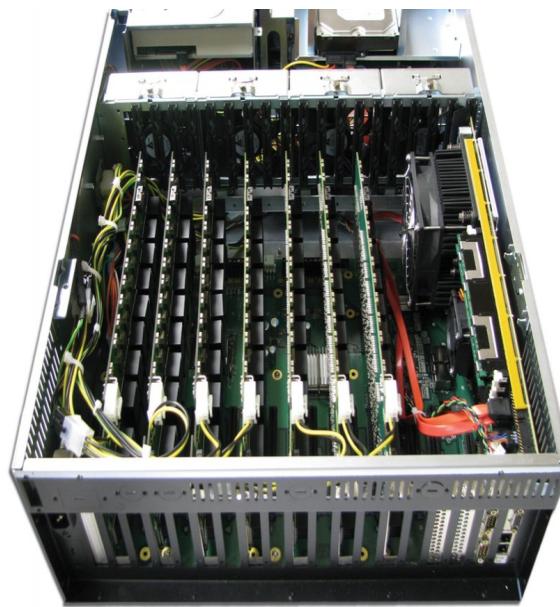


Bruteforce password attack on FPGAs



Thèse de Bachelor présentée par

Abivarman KANDIAH

pour l'obtention du titre Bachelor of Science HES-SO en

**Informatique et systèmes de communication avec orientation
Systèmes informatiques embarqués**

Septembre 2024

Professeur-e HES responsable

Andres UPEGUI POSADA

Mandant

ELCA Security

Légende et source de l'illustration de couverture :

<https://militaryembedded.com/cyber/encryption/accelerating-cryptography-fpga-clusters>

TABLE DES MATIÈRES

Remerciements	v
Résumé	vii
Liste de acronymes	viii
Liste des illustrations	x
Liste des tableaux	xi
Liste des annexes	xii
Introduction	1
1 Chapitre 0 : Base Technique	3
1.1 FPGA	3
1.2 Fonction de hachage	5
a Salt	5
b Attaque par bruteforce	6
2 Chapitre 1 : Analyse	7
2.1 Description du projet	7
2.2 Méthodes de communication	8
a UART	8
b PCIe	9
2.3 Bcrypt	10
a Algorithme	10
b Format du Hash	11
c Implémentations Existantes	12
3 Chapitre 2 : Bcrypt sur FPGA	13
3.1 Bcrypt sur FPGA	13
a Bcrypt Core	13
b Password Generator	18
c Bcrypt Quadcore	19
d Bcrypt Cracker	20
4 Chapitre 3 : Implémentation Des Solutions	22
4.1 Description Solution UART	22
a Encodage Cobs	23
b CRC	24
c Format et Type de Paquet	24
c.1 Paquet pour le FPGA	24
c.2 Paquet pour l'ordinateur	25
4.2 Implémentation Solution UART	27

a	Architecture Logique	27
b	Implémentation - MOSI	27
	b.1 Module - Packet Receiver	28
	b.2 Module - RX Packet Process	28
	b.3 Module - RX Packet Pipeline	29
c	Implémentation - MISO	29
	c.1 Module - TX Packet Pipeline	30
	c.2 Module - Packet Transmitter	30
d	Modifications Bcrypt Cracker	30
e	Vérification Hardware	31
	e.1 Interfacage avec PC	31
	e.2 Test réception sur FPGA	31
	e.3 Test système au complet	32
f	Améliorations	32
4.3	Description Solution PCIe	34
4.4	Implémentation Solution PCIe	34
	a Architecture Logique	34
	b Implémentation sur FPGA	35
	b.1 Implémentation Cracker Regs	36
	b.2 Modifications Bcrypt Cracker	36
	c Vérification Hardware	37
5	Chapitre 4 : Mesures et Performances	39
5.1	Mesures CPU	39
5.2	Mesures GPU	39
5.3	Mesures FPGA	40
	a Solution UART	40
	b Solution PCIe	42
	c Comparaisons	42
	c.1 Comparaison du Hashrate	43
	c.2 Comparaison de la Consommation Électrique	43
	c.3 Efficacité Énergétique	44
	c.4 Consommation Énergétique sur 24 Heures	45
	c.5 Conclusion des Comparaisons	45
Conclusion	46	
Annexes	47	
Références documentaires	50	

REMERCIEMENTS

Je suis reconnaissant envers M. Upegui, qui m'a suivi tout au long de ce projet. Je tiens également à remercier mes parents, dont le soutien moral et les encouragements m'ont beaucoup aidé.

ATTAQUE BRUTE-FORCE SUR FPGA POUR BCRYPT**ORIENTATION : SYSTÈMES INFORMATIQUES EMBARQUÉS****Descriptif :**

Le projet vise à développer un accélérateur de calcul basé sur des circuits FPGA pour attaquer les mots de passe protégés par l'algorithme bcrypt, surpassant les performances des approches traditionnelles basées sur CPUs et GPUs. L'étudiant implémentera un cœur de calcul compact afin d'instancier plusieurs cœurs de calcul dans un seul FPGA, avec des interfaces de communication intégrées pour interagir avec l'hôte. Le système inclura une interface UART pour une communication série simple et économique, ainsi qu'une interface PCIe offrant des vitesses de transfert élevées pour les besoins de performances intensives. Un protocole adapté sera mis en place pour gérer les différents types de messages entre le PC et le FPGA.

Deux stratégies seront explorées pour la génération des mots de passe. La première consiste à utiliser un compteur interne au FPGA pour optimiser la vitesse et l'efficacité. La seconde permettra d'implémenter des attaques par dictionnaire, gérées par un PC qui enverra les mots de passe via l'interface PCIe. Cette configuration tirera pleinement parti des capacités de parallélisation du FPGA, offrant une solution extensible et performante pour le cassage de mots de passe.

Travail demandé : Dans le cadre de ce projet l'étudiant devra :

- Implémenter un cœur de calcul compact pour bcrypt en utilisant VHDL ou réutiliser des cours déjà existants.
- Instancier et paralléliser plusieurs cœurs de calcul sur le FPGA.
- Analyser des performances et les comparer avec des résultats sur CPU et/ou GPU.
- Proposer et implémenter des solutions pour la génération de mots de passe (compteur et dictionnaire)
- Mise en place des communications et le protocole nécessaire pour garantir la fiabilité.
- Effectuer des tests unitaires et des tests d'intégration pour vérifier le bon fonctionnement du système.
- Optimiser les performances.

Candidat-e :

KANDIAH ABIVARMAN

Filière d'études : ISC

Professeur-e(s) responsable(s) :

UPEGUI ANDRES

En collaboration avec : ELCA Security

Travail de bachelor soumis à une convention de stage en entreprise : non

Travail de bachelor soumis à un contrat de confidentialité : non

RÉSUMÉ

Lorsqu'un utilisateur doit s'authentifier auprès d'un service, il doit fournir un mot de passe préalablement défini. Pour des raisons de sécurité, le système stocke ce mot de passe en le passant par une fonction de hachage. Ainsi, lors de l'authentification, le système compare le hash du mot de passe entré par l'utilisateur avec le hash stocké pour vérifier son identité. Une fonction de hachage qui est souvent utilisé pour le stockage de mots de passe est le Bcrypt, qui a comme particularité d'être assez lente, rendant les mots de passe assez résistante aux attaques par bruteforce. Ce rapport a pour but de détailler la mise en oeuvre d'un système visant à attaquer les mots de passe protégés par l'algorithme bcrypt en utilisant un Field-Programmable Gate Array (FPGA). L'objectif principal est de créer une solution extensible et plus performante que les approches traditionnelles basées sur Graphics Processing Unit (GPU). Le système repose sur plusieurs cœurs de calcul parallèles sur le FPGA pour générer les hashs bcrypt. Ce projet présente deux solutions distinctes pour l'implémentation d'une attaque par bruteforce. La première solution génère les mots de passe directement sur le FPGA, accompagnée d'une interface Universal Asynchronous Receiver Transmitter (UART) qui permet à l'utilisateur d'interagir avec le système. La seconde solution, encore en développement, repose sur l'utilisation de l'interface Peripheral Component Interconnect Express (PCIe) pour permettre des transferts de données rapides, rendant ainsi possible une attaque par dictionnaire. Après l'implémentation, des mesures ont été effectuées pour évaluer les performances et l'utilisation des ressources sur le FPGA. Bien que la solution basée sur l'interface PCIe ne soit pas encore entièrement opérationnelle, les résultats obtenus jusqu'à présent ouvrent la voie à des optimisations et des améliorations futures.



Candidat-e :

ABIVARMAN KANDIAH

Filière d'études : ISC

Professeur-e(s) responsable(s) :

ANDRES UPEGUI POSADA

En collaboration avec : ELCA Security

Travail de bachelor soumis à une convention de stage
en entreprise : non

Travail soumis à un contrat de confidentialité : non

LISTE DE ACRONYMES

BRAM Block RAM. 14, 15, 19, 20, 34, 35, 36

COBS Consistent Overhead Byte Stuffing. 23, 30, 31, 46

CPU Central Processing Unit. 39, 40, 42, 43, 44, 45

CRC Cyclic Redundancy Check. 24, 27, 28, 29, 30, 31, 46

FPGA Field-Programmable Gate Array. vii, 1, 2, 3, 4, 7, 8, 9, 12, 22, 24, 25, 26, 31, 34, 40, 42, 43, 44, 45, 46, 47

GPIO General Purpose Input/Output. 37

GPU Graphics Processing Unit. vii, 1, 7, 39, 40, 42, 43, 44, 45, 46

I/O Input / Output. 3, 14

IC Integrated Circuit. 3

IP Intellectual Property. 9, 34, 35, 37

LUT Look Up Table. 3

PC Personal Computer. 7, 8, 9, 31, 34, 36

PCIe Peripheral Component Interconnect Express. vii, 2, 8, 9, 34, 37, 47

SBOX Substitution boxes. 10, 17

UART Universal Asynchronous Receiver Transmitter. vii, 8, 22, 27, 28, 29, 30, 31, 42, 47

VHDL Very High Speed Integrated Circuit Hardware Description Language. 1, 3, 4, 8, 12, 13, 24, 27, 32, 46

LISTE DES ILLUSTRATIONS

1.1	Schéma interne d'un FPGA	3
1.2	Fonction de hachage	5
1.3	Salt	6
2.1	Diagramme général	7
2.2	Diagramme général avec UART	8
2.3	Diagramme général avec PCIe	9
2.4	Algorithme Bcrypt	11
2.5	Format du hash Bcrypt	11
2.6	Différence Base 64	12
3.1	Architecture Bcrypt sur FPGA	13
3.2	Interface du Bcrypt core	14
3.3	Schéma Bcrypt core simplifié	15
3.4	Machine d'état Bcrypt core simplifié	16
3.5	Timing du bcrypt	17
3.6	Table de conversion	18
3.7	Machine d'état Bcrypt quadcore simplifié	20
3.8	Timing du bcrypt cracker	21
4.1	Carte de développement Nexys Video	22
4.2	Schéma Encodage COBS	23
4.3	Format de paquet - UART	24
4.4	Format de paquet - MOSI	25
4.5	Format de paquet - Retour	25
4.6	Format de paquet - MISO	26
4.7	Schéma système UART - FPGA	27
4.8	Schéma Packet Receiver	28
4.9	Schéma RX Packet Process	29
4.10	Carte de développement KCU116	34

4.11 Schéma Système PCIe - FPGA	35
4.12 Schéma logique AXI4 Attack Ctrl - FPGA	35
4.13 BRAM Adresses - FPGA	36
4.14 Block Design Test PCIe - FPGA	38
5.1 Mesures Bcrypt CPU	39
5.2 Mesures Bcrypt GPU	40
5.3 Ressources utilisés Nexys Video	41
5.4 Mesures Puissance Max Nexys Video	41
5.5 Mesures Puissance Max KCU116	42
5.6 Évolution de la consommation énergétique sur 24 heures	45

Références des URL

- URL01 Introduction to FPGA and Its Architecture
- URL02 Nexys Video Reference
- URL03 Consistent Overhead Byte Stuffing (COBS)
- URL04 Xilinx KCU116 Board

LISTE DES TABLEAUX

4.1	Tableau de code d'erreur - Paquet UART	25
5.1	Mesures de hashrate sur KCU116	42
5.2	Comparaison des hashrates pour différentes architectures.	43
5.3	Comparaison des consommations électriques pour différentes architectures. . .	43
5.4	Comparaison de l'efficacité énergétique (Hashes par joule).	44

LISTE DES ANNEXES

Annexe 1	49
-----------------	-------	-----------

INTRODUCTION

Ce travail s'inscrit dans le cadre de mon travail de bachelor, réalisé en réponse à une demande d'ELCA Security, une entreprise spécialisée dans la cyber-sécurité. Le projet a pris comme source d'inspiration un article hackaday¹, qui parle de l'utilisation de **FPGA** pour une attaque par bruteforce de mots de passe. Le projet présenté ici vise à explorer une approche peu commune pour attaquer des mots de passe protégés par l'algorithme bcrypt en utilisant un **FPGA**. Lorsque l'on souhaite faire une attaque pour casser des mots de passe protégés par un hash, la solution habituelle est l'utilisation d'un **GPU**. Le problème est que généralement un **GPU** est une solution plutôt énergivores et ce genre d'attaque peut prendre énormément de temps. L'intérêt dans l'utilisation d'un **FPGA** plutôt qu'un **GPU** dans ce cas précis est la possibilité d'avoir une solution beaucoup moins énergivores. Ce projet a pour objectif final de fournir une solution extensible et performante qui puisse être utilisable lors d'une attaque par bruteforce.

Ce projet a été entamé avec le travail de semestre. Après une recherche approfondie des implémentations existantes du bcrypt sur FPGA, le projet de semestre a débuté par une analyse du papier² concernant une implémentation déjà existante. En parallèle, une page Wikipédia détaillant le fonctionnement de bcrypt³ a été utilisée comme ressource principale pour comprendre les spécificités de cet algorithme. De plus, un papier sur une attaque MD5⁴ sur FPGA a été consulté pour enrichir la compréhension des techniques d'attaque sur des dispositifs matériels. Ces ressources documentaires ont été cruciales pour comprendre les différents concepts, orienter les choix de conception et résoudre les problèmes techniques rencontrés.

Suite à ces recherches, j'ai pu reprendre l'implémentation existante en **Very High Speed Integrated Circuit Hardware Description Language (VHDL)** d'un programme d'attaque par bru-

1. *BY. All Your Passwords Are Belong To FPGA.* en-US. Mai 2020. URL : <https://hackaday.com/2020/05/15/all-your-passwords-are-belong-to-fpga/> (visité le 11/08/2024).

2. Friedrich WIEMER et Ralf ZIMMERMANN. "High-speed implementation of bcrypt password search using special-purpose hardware". In : *2014 International Conference on ReConfigurable Computing and FPGAs (ReConFig14)*. ISSN: 2325-6532. Déc. 2014, p. 1-6. DOI : 10.1109/ReConfig.2014.7032529. URL : <https://ieeexplore.ieee.org/document/7032529> (visité le 10/03/2024).

3. *bcrypt.* en. Page Version ID: 1210874707. Fév. 2024. URL : <https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1210874707> (visité le 10/03/2024).

4. Maruthi GILLELA, Vaclav PRENOSIL et Venkat Reddy GINJALA. "Parallelization of Brute-Force Attack on MD5 Hash Algorithm on FPGA". In : *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. ISSN: 2380-6923. Jan. 2019, p. 88-93. DOI : 10.1109/VLSID.2019.00034. URL : <https://ieeexplore.ieee.org/document/8710753> (visité le 10/03/2024).

teforce de mot de passe bcrypt. Après avoir étudié le papier associé et constaté des incohérences dans les testbenches, j'ai refait ces derniers pour valider le programme. Par la suite, j'ai identifié et corrigé des erreurs dans le code afin d'obtenir un programme fonctionnel. En parallèle, j'ai brièvement étudié le fonctionnement du PCIe afin d'y mettre en place une simple interface entre une carte **FPGA** et un ordinateur.

Poursuivant ce travail dans le cadre de mon projet de bachelor, j'ai développé une interface UART afin de pouvoir initialiser les différents cœurs de calcul bcrypt. Cette solution a nécessité la mise en place d'un système de paquets utilisant un système d'encodage COBS. Ce système permet non seulement de recevoir des confirmations ou des erreurs en cas de problème avec les paquets reçus, mais également de renvoyer le mot de passe une fois celui-ci trouvé.

Parallèlement, j'avais prévu de mettre en place une solution PCIe pour améliorer les performances de communication entre la carte FPGA et l'ordinateur, ce qui aurait permis d'exploiter pleinement la puissance de calcul de la FPGA. Cependant, en raison des contraintes de temps, je n'ai pas pu finaliser cette partie du projet. Malgré les efforts déployés, la solution PCIe n'a pas pu être complétée dans les délais impartis, ce qui a conduit à se concentrer sur la solution UART pour atteindre les objectifs principaux du développement.

Dans ce rapport, je vais commencer par brièvement expliquer les différents notions clés de ce projet, afin de poser une base technique. Je vais par la suite vous présenter une analyse du projet, afin d'y décrire le projet de manière détaillé, expliquer le fonctionnement du Bcrypt et les différentes solutions mise en place durant ce projet. Puis, je vais détailler la méthodologie de travail, en exposant les différentes étapes de mise en œuvre du projet, les simulations et les tests qui ont été faites et les différentes difficultés rencontrées. Enfin, je vais décrire les résultats obtenus lors des différents tests et mesures qui ont été faits.

CHAPITRE 0 : BASE TECHNIQUE

Ce chapitre a pour but d'introduire et expliquer les différents aspects techniques clés de ce projet de Bachelor. Je vais notamment expliquer brièvement ce qu'est un **FPGA** et le principe d'une fonction de hachage.

1.1. FPGA

Un **FPGA** est un **Integrated Circuit (IC)** qui va contenir un certain nombre de blocs logiques configurable, de fils d'interconnection, de mémoire et d'**Input / Output (I/O)**. Un bloc logique est composé de plusieurs **Look Up Table (LUT)**s et de registres.

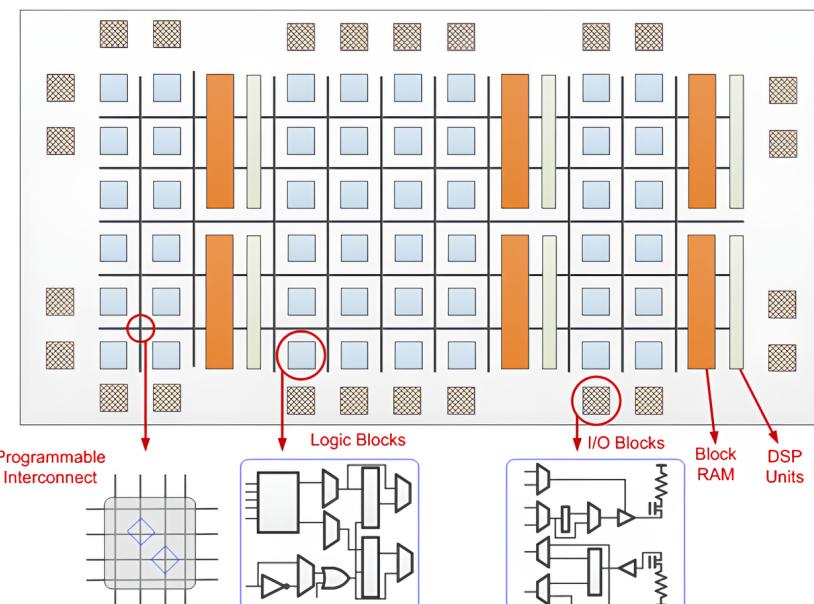


ILLUSTRATION 1.1 – Schéma interne d'un FPGA. Source : [towardsdatascience.com](https://towardsdatascience.com/introduction-to-fpgas-what-is-a-field-programmable-gate-array-1a2a0f3a2a0c) ref. URL01

Contrairement à un processeur, qui est limité par un certain nombre d'instructions et exécute les instructions de manière séquentielle, un **FPGA** permet d'exécuter de nombreux circuits logiques en parallèle.

Pour programmer un **FPGA**, on utilise généralement des langages de description matériel tels que le **VHDL** et le Verilog. Dans ce projet, j'ai personnellement travaillé avec le **VHDL**.

Le processus de programmation d'un **FPGA** est séparé en deux étapes : la synthèse et l'implémentation. La synthèse consiste à traduire le code **VHDL** en registres et portes logiques. L'implémentation lui consiste à placer les différents composants logiques sur le **FPGA** et à les

interconnecter. Ces différents processus vont prendre généralement beaucoup de temps.

Lorsque l'on souhaite tester un programme **VHDL**, il est possible d'utiliser des outils de simulation afin de vérifier le fonctionnement souhaité. Il est aussi possible d'automatiser la phase de simulation à l'aide de fichier que l'on appelle testbench. La simulation va permettre de valider une première fois le programme afin d'éviter de perdre du temps à reprogrammer le **FPGA**.

Durant ce travail, j'ai utilisé Vivado qui est le logiciel qui m'a permis la simulation et la programmation des **FPGA** qui ont été utilisés.

1.2. FONCTION DE HACHAGE

Une fonction de hachage est une fonction qui va prendre en entrée une donnée à taille variable et va ressortir une donnée de taille fixe.

Une des propriétés fondamentales d'une fonction de hachage est qu'il n'existe pas de fonction mathématique permettant de retrouver la donnée originale à partir d'un hash généré. Même une petite modification apportée à la donnée en entrée conduira à un hash totalement différent en sortie. Cette particularité est essentielle pour sécuriser le stockage des mots de passe, car même si des hash venait à être compromis, il est extrêmement difficile de retrouver les mots de passe originaux à partir de leurs hachages.

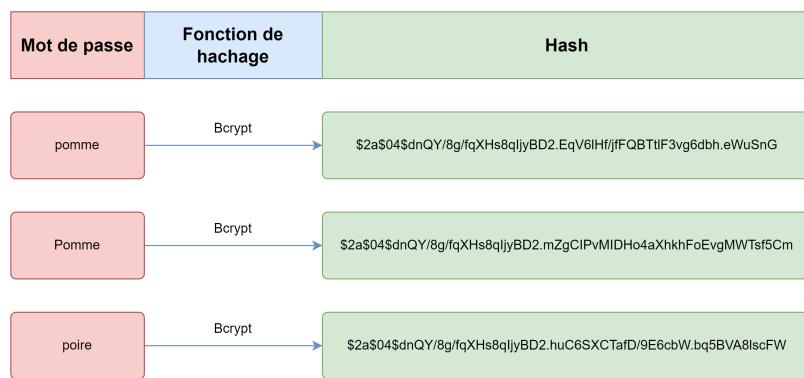


ILLUSTRATION 1.2 – Fonction de hachage. Source : réalisé par Kandiah Abivarman

a. Salt

Certaines fonctions de hachage tel que le bcrypt utilisent ce qu'on appelle un salt (sel en français), qui est une valeur générée aléatoirement qu'on va donner avec notre mot de passe. Le salt va permettre d'avoir un hash différent, même si deux personnes utilisent le même mot de passe, ajoutant ainsi une couche supplémentaire de sécurité.

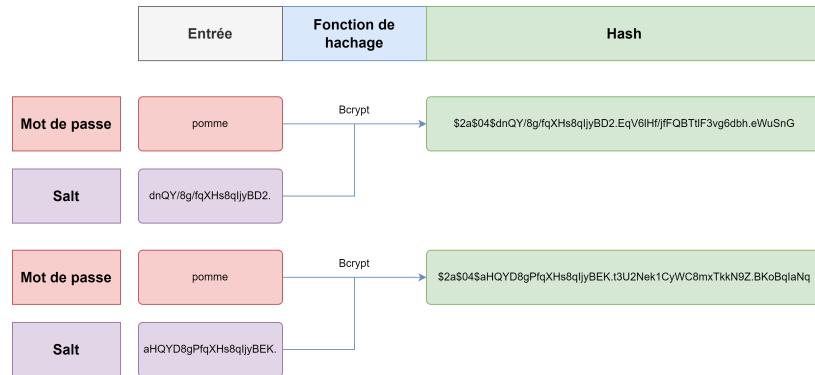


ILLUSTRATION 1.3 – Salt. Source : réalisé par Kandiah Abivarman

b. Attaque par bruteforce

L'attaque par bruteforce consiste à essayer toutes les combinaisons possibles de mots de passe afin de retrouver celui qui correspond au hash compromis. Cette méthode repose sur le fait qu'il est impossible de retrouver directement le mot de passe à partir du hash, obligeant ainsi l'attaquant à tester différentes entrées jusqu'à ce qu'il trouve celle qui génère le hash recherché.

Toutefois, cette méthode peut prendre beaucoup de temps, notamment lorsque les fonctions de hachage utilisées sont conçues pour être lentes à calculer.

CHAPITRE 1 : ANALYSE

2.1. DESCRIPTION DU PROJET

L'objectif principal de ce projet est d'exploiter le parallélisme offert par les **FPGA**, afin de calculer les fonctions de hachage nécessitant beaucoup de temps de calculs. Le but étant d'avoir au final un système plus efficient que les solutions actuelles sur **GPU** et de la rendre la plus scalable possible. Il est aussi nécessaire d'avoir une certaine communication entre le **Personal Computer (PC)** de l'attaquant et le **FPGA**, afin que l'attaquant puisse fournir les informations nécessaires au cassage du mot de passe.

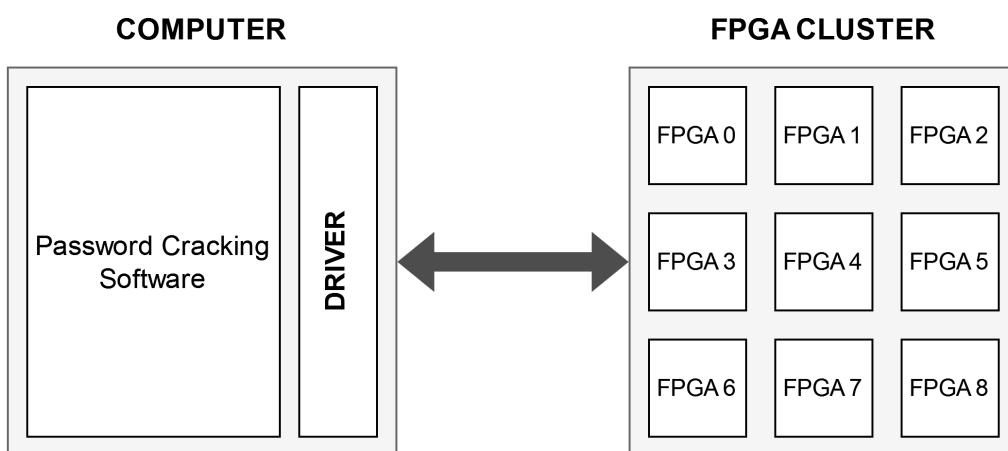


ILLUSTRATION 2.1 – Diagramme général. Source : réalisé par Kandiah Abivarman

L'idéal serait d'avoir deux solutions, une première qui consiste à simplifier le système en générant directement les mots de passe dans le **FPGA** à l'aide de système de compteur. Puis la deuxième solution, serait de générer les mots de passe sur le **PC** et d'ensuite les transférer au **FPGA** pour qu'il puisse faire les calculs. La première solution sera plus simple à mettre en place, car ne nécessite pas de communication rapide entre le **PC** et le **FPGA** et la deuxième solution sera plus complexe à mettre en place, mais nous apportera une plus grande flexibilité au niveau de la génération de mots de passe, nous permettant notamment une attaque par dictionnaire⁵.

5. *Attaque par dictionnaire.* fr. Page Version ID: 188231625. Nov. 2021. URL : https://fr.wikipedia.org/w/index.php?title=Attaque_par_dictionnaire&oldid=188231625 (visité le 21/03/2024).

2.2. MÉTHODES DE COMMUNICATION

Pour la communication entre le **PC** et la carte **FPGA**, j'ai pu identifier trois méthodes :

1. L'**UART** qui est la méthode la plus simple à mettre en place
2. Le **PCIe** qui est la méthode de communication la plus rapide
3. Le **Ethernet** qui est la méthode la plus scalable.

Dans ce projet, j'ai décidé de travailler avec l'**UART** et le **PCIe**. Toutefois, même si je n'ai pas utilisé l'**Ethernet** durant ce projet, cette méthode reste intéressante pour une solution nécessitant un grand nombre de **FPGA**.

a. **UART**

L'**UART** est une méthode de communication avec laquelle je suis déjà familier. Il est assez simple de la mettre en place côté **PC**, notamment en Python. Pour ce qui est de l'**UART** côté **FPGA**, j'ai déjà en ma possession du code **VHDL** qui nous a été donné durant nos cours qui me permet aisément d'en faire. Cette méthode est idéale pour le premier système dont la génération des mots de passe est faite directement dans le **FPGA**. En effet, la communication servira seulement à configurer le système au début de l'attaque puis recevoir le résultat lorsque le bon mot de passe a été trouvé.

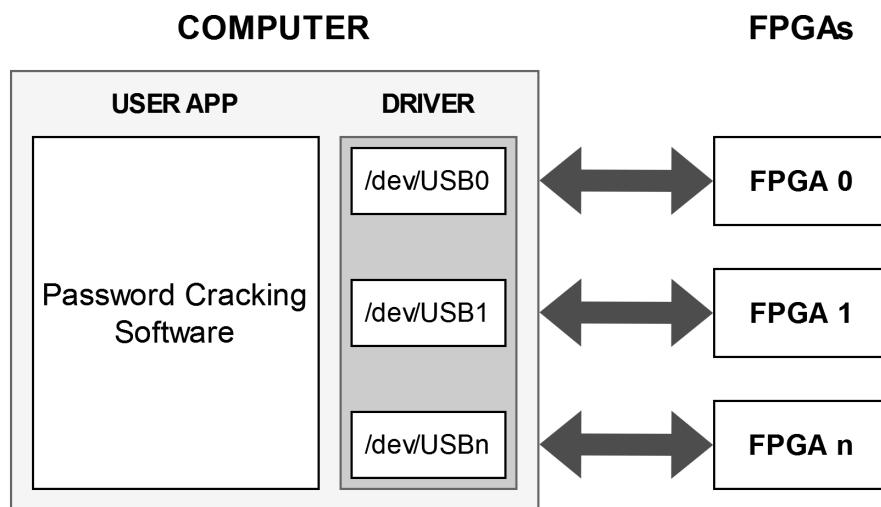


ILLUSTRATION 2.2 – Diagramme général avec **UART**. Source : réalisé par Kandiah Abivarman

Toutefois, afin d'avoir une communication robuste, il est nécessaire de mettre en place un

système de paquet. Le système de paquet devrait avoir un moyen de synchronisation et une vérification d'erreurs.

b. PCIe

Le PCIe est une méthode qui permet de transférer beaucoup de données à très haute vitesse. Cette méthode de communication va être utilisé pour le deuxième système, celui-ci va nous permettre de transmettre les mots de passe depuis le PC au FPGA. En effet, il nous faut une communication rapide, car il faudrait pas que la communication soit le goulot d'étranglement de notre système.

Pour ce faire, du côté de l’FPGA il y a un bloc Intellectual Property (IP) qui est fourni qui permet d’interfacer avec le PCIe. La difficulté provient majoritairement du PC, car il va falloir mettre en place le driver PCIe nous-même. Evidemment, il n'y a pas besoin de partir de zéro non plus, on peut retrouver des exemples de driver linux PCIe sur internet.

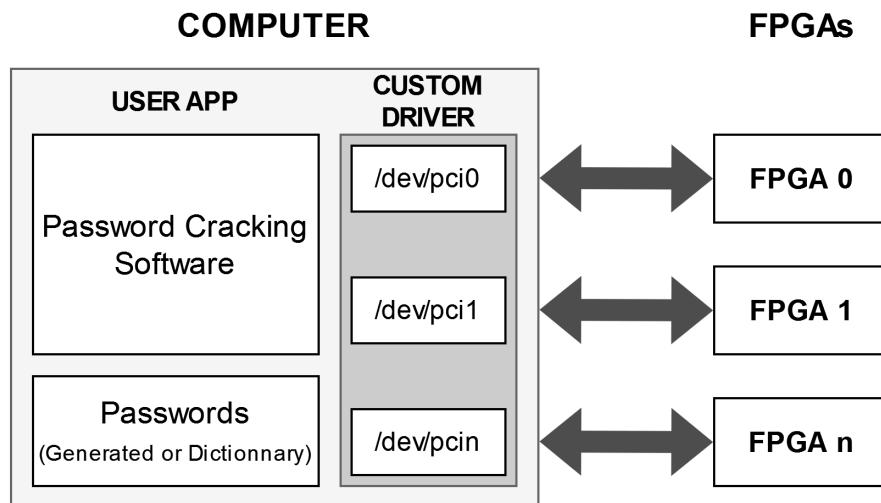


ILLUSTRATION 2.3 – Diagramme général avec PCIe. Source : réalisé par Kandiah Abivarman

2.3. BCRYPT

Pour ce projet, nous avons décidé de cibler le Bcrypt, car c'est une fonction de hachage qui prend du temps à être calculé.

Le Bcrypt est une fonction de hachage avec comme particularité, un paramètre supplémentaire qui est le cost (coût en français). Ce paramètre va définir le nombre d'itérations que va prendre la fonction de hachage, de ce fait plus le cost est élevé, plus le calcul va prendre du temps.

a. Algorithme

L'algorithme du Bcrypt se base sur l'algorithme de chiffrement Blowfish⁶ qui est une fonction de chiffrement à clef symétrique, c'est-à-dire que la même clef est utilisée pour le chiffrement et le déchiffrement. L'algorithme du Bcrypt peut être divisé en deux grandes étapes.

On a une première étape qui est une phase de mise en place des clés symétriques. Dans cette étape, on va créer les clés de chiffrements à partir des paramètres d'entrée de la fonction de hachage (mot de passe, salt, cost). Cette première étape est la partie la plus coûteuse de la fonction, car la mise en place de la clé va prendre plus ou moins de temps en fonction du cost. Les clés de chiffrement sont composées de Subkeys qui est un tableau de 18 entiers de 32 bits et quatre Substitution boxes (SBOX) qui sont chacun des tableaux de 256 entiers de 32 bits. Avant de calculer ces clés de chiffrements, ils sont tout d'abord initialisés avec les décimales de PI.

Puis il y a la deuxième étape, où l'on va utiliser les clés de chiffrement qui ont été calculées plus tôt afin de chiffrer la phrase magique "OrpheanBeholderScryDoubt", le chiffrement va être fait 64 fois.

6. *Blowfish Algorithm with Examples.* en-US. Section: Algorithms. Oct. 2019. URL : <https://www.geeksforgeeks.org/blowfish-algorithm-with-examples/> (visité le 21/03/2024).

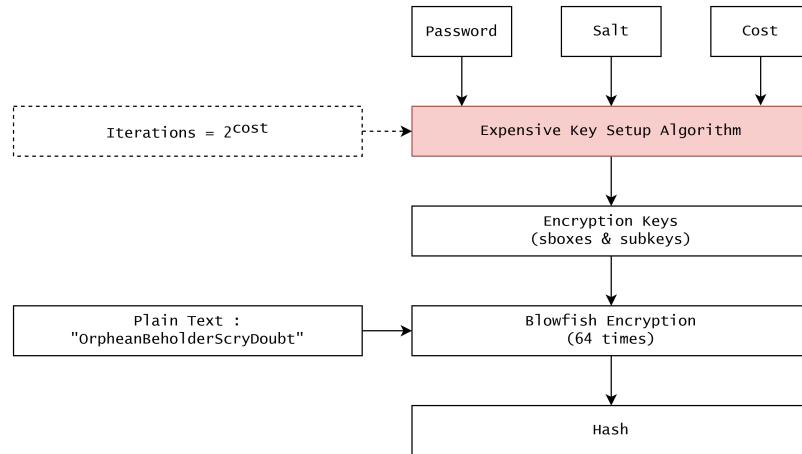


ILLUSTRATION 2.4 – Algorithme Bcrypt. Source : réalisé par Kandiah Abivarman

b. Format du Hash

Le hash généré par la fonction Bcrypt est généralement stocker sous une forme particulière.

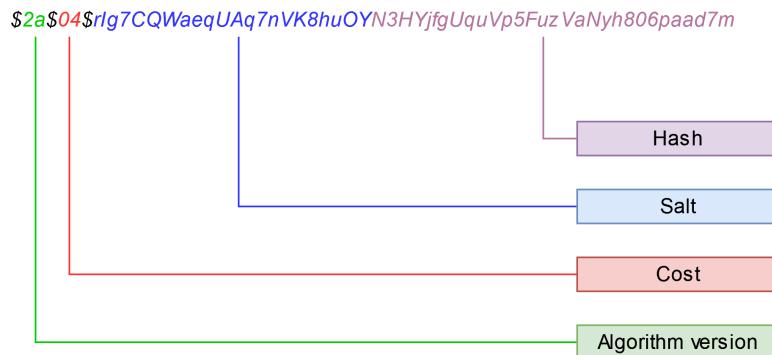


ILLUSTRATION 2.5 – Format du hash Bcrypt. Source : réalisé par Kandiah Abivarman

On va avoir un premier champ qui contient la version de l'algorithme, un deuxième qui contient le cost de la fonction, un troisième avec le salt et le quatrième avec le hash généré. Le salt et le hash sont en base 64, mais il faut faire attention, car c'est une base 64 différente de la norme RFC 4648⁷ qui est couramment utilisé.

7. Simon JOSEFSSON. *The Base16, Base32, and Base64 Data Encodings*. Request for Comments RFC 4648. Num Pages: 18. Internet Engineering Task Force, oct. 2006. DOI : 10 . 17487 / RFC4648. URL : <https://datatracker.ietf.org/doc/rfc4648> (visité le 21/03/2024).

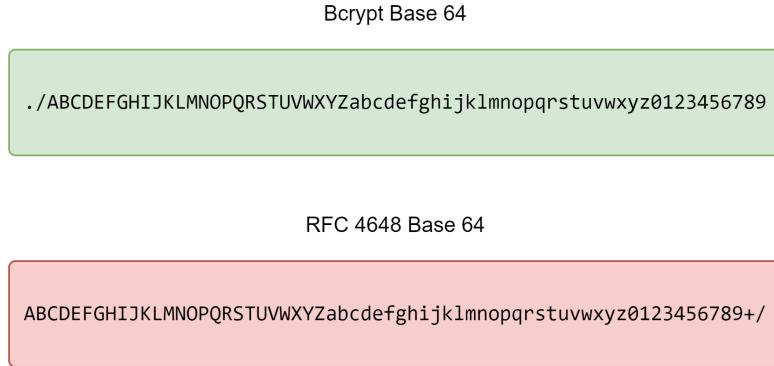


ILLUSTRATION 2.6 – Différence Base 64. Source : réalisé par Kandiah Abivarman

c. Implémentations Existantes

Afin d'éviter de réinventer la roue, la première tâche que j'ai entrepris est de chercher afin de voir s'il n'existe pas déjà des implémentations existantes sur **FPGA**.

D'après mes recherches, j'ai retrouvé seulement deux implémentations du Bcrypt sur **FPGA**. La première se situe dans le répertoire [github](#) de [JohnTheRipper](#)⁸ qui est un logiciel de cyber-sécurité destiné au craquage de mot de passe, l'implémentation a été faite en Verilog et spécifiquement pour la Ztex 1.15y qui est une carte **FPGA** assez ancienne et difficilement retrouvable. La deuxième est une implémentation faite en **VHDL** que j'ai aussi retrouvé dans un répertoire [github](#)⁹, accompagné d'un papier¹⁰ décrivant un travail de recherche effectué sur l'attaque de mot de passe sur **FPGA**. Pour ma part, connaissant seulement le **VHDL** et ne comprenant pas réellement la structure de code du premier et par manque d'informations, j'ai préféré reprendre le code du deuxième.

Le papier a été très instructif, j'ai pu notamment comprendre les différents choix qui ont été pris dans le code source. Malheureusement, tout n'a pas été documenté et le répertoire n'a pas été mis en place correctement. En effet, certains parties du code contenait des erreurs, les fichiers de tests étaient incomplets et des fichiers source semble avoir été retravaillé en aval.

8. [openwall/john](#). original-date: 2011-12-16T19:43:47Z. Mars 2024. URL : <https://github.com/openwall/john> (visité le 21/03/2024).

9. [rub-hgi/high-speed_bcrypt](#): VHDL implementation and LaTeX source of "High-Speed Implementation of bcrypt Password Search using Special-Purpose Hardware", published at ReConfig'14. URL : https://github.com/rub-hgi/high-speed_bcrypt (visité le 21/03/2024).

10. WIEMER et ZIMMERMANN, "High-speed implementation of bcrypt password search using special-purpose hardware".

CHAPITRE 2 : BCRYPT SUR FPGA

Tout le travail, dont je vais parler dans ce chapitre a été fait en grande partie durant le projet de semestre. Comme expliqué dans le chapitre précédent, pour le bcrypt, je suis reparti d'une implémentation déjà existante en [VHDL](#).

3.1. BCRYPT SUR FPGA

Après lecture du code source, j'ai pu déduire l'architecture suivante :

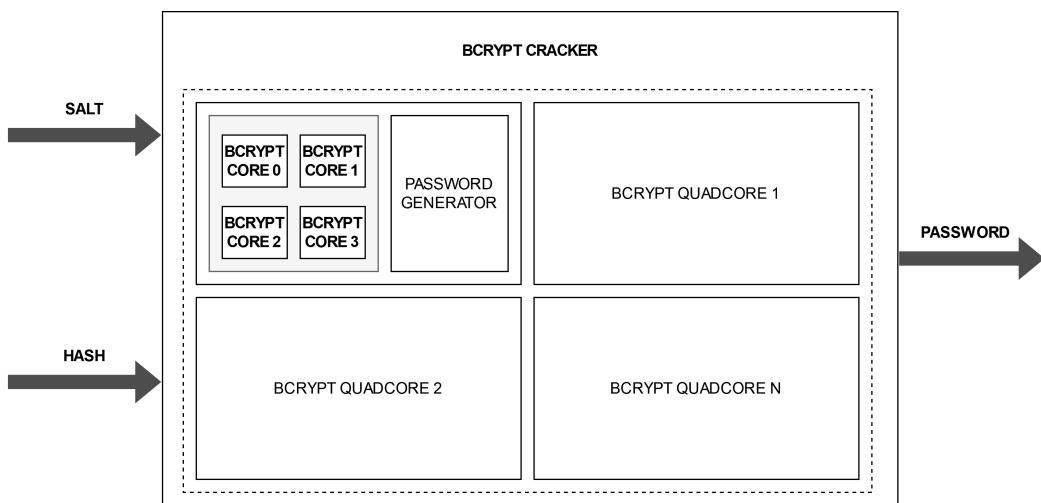


ILLUSTRATION 3.1 – Architecture Bcrypt sur FPGA. Source : réalisé par Kandiah Abivarman

L'architecture contient 4 modules clé, il y a d'abord le bcrypt core qui est le cœur de calcul qui va s'occuper de faire la fonction de hachage bcrypt. Ensuite, le password generator qui va s'occuper de générer les différents mots de passe pour l'attaque par bruteforce. Puis, le bcrypt quadcore qui va s'occuper d'instancier quatre bcrypt core et un générateur de mots de passe pour alimenter les cœurs en mot de passe. Enfin, le bcrypt cracker va instancier le nombre souhaité de quadcore, s'occuper de la gestion des différents coeurs et retransmettre le mot de passe lorsque il est retrouvé.

a. Bcrypt Core

Le bcrypt core était la partie qui m'intéressait le plus dans ce code source, car c'est le module qui s'occupe de faire le hachage et c'était ce que je cherchais initialement parmi les implémentations déjà existantes.

J'ai donc entamé le projet en testant le module avec de la simulation en utilisant le testbench fourni, mais je me suis vite rendu compte que le testbench fourni ne fonctionnait pas. En effet, le testbench fourni semble avoir été fait pour une ancienne version du bcrypt core avec une interface totalement différente, rendant le fichier de test obsolète.

Afin de pouvoir implémenter moi-même le testbench de ce module, je suis passé par une première phase où j'ai analysé le code afin de comprendre l'interface du bcrypt core.

J'ai pu notamment identifier les **I/O** qui permettent le contrôle du module, les **I/O** de la fonction de hachage (mot de passe, salt et hash) et les **I/O** qui vont permettre l'initialisation de la mémoire pour les clés de chiffrement. Le coût de la fonction de hachage est lui fixé par une constante située dans un fichier à part regroupant d'autres constantes du système.

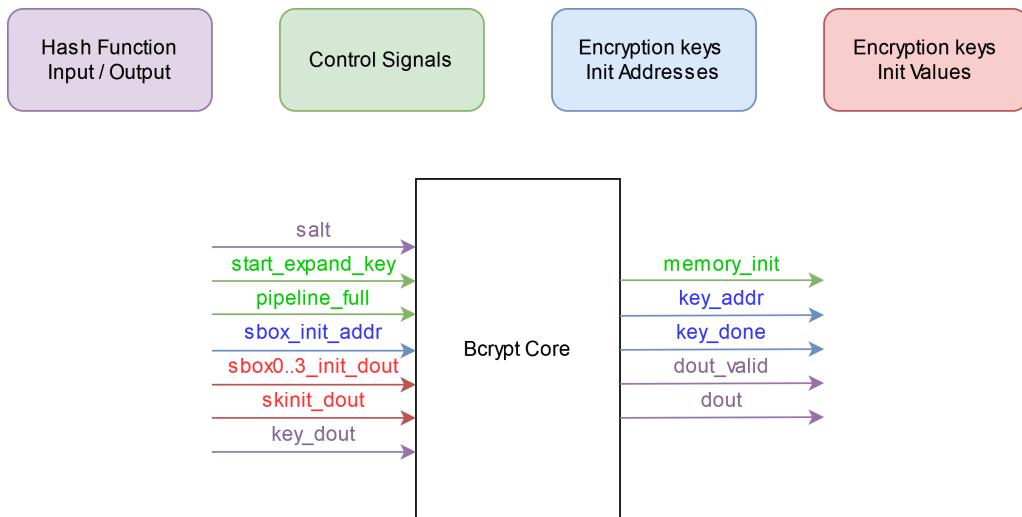


ILLUSTRATION 3.2 – Interface du Bcrypt core. Source : réalisé par Kandiah Abivarman

Après une identification des **I/O**, j'ai examiné les différents processus et instantiations qui ont lieu dans le module bcrypt core. Il y a tout d'abord plusieurs **Block RAM (BRAM)** qui sont utilisés pour le stockage des clés de chiffrement, un module qui s'occupe du chiffrement blowfish, une machine d'état pour gérer les différentes étapes de la fonction de hachage et différents compteurs nécessaires à l'adressage mémoire et à la machine d'état.

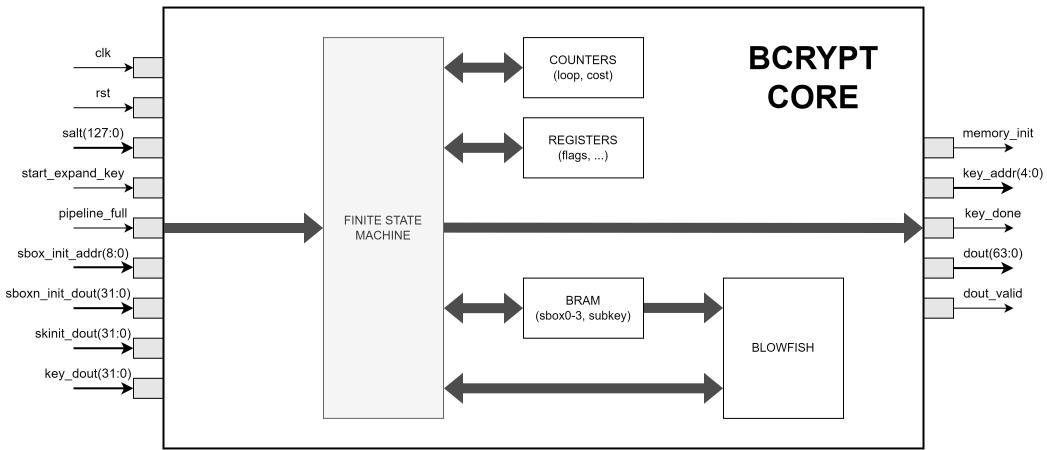


ILLUSTRATION 3.3 – Schéma du Bcrypt core simplifié. Source : réalisé par Kandiah Abivarman

La machine d'état contient 18 états, mais je vais la simplifier pour l'explication. Tout d'abord, le module va attendre un signal du module parent afin de démarrer, après réception du signal, la phase d'initialisation de la mémoire est lancée. Cette étape consiste à initialiser les clés de chiffrement, pour se faire, il faut fournir au module l'adresse mémoire où l'on souhaite écrire dans les **BRAM** et les données que l'on souhaite écrire dans notre cas les différents décimaux de PI. Après la phase d'initialisation de mémoire, le module va de nouveau attendre un signal en entrée afin de procéder au calcul des clés de chiffrement. Cette étape consiste en 7 états dans la machine d'état et va reboucler un certain nombre de fois en fonction du cost. Après les calculs des clés de chiffrement, vient le chiffrement du mot magique qui va nous donner notre hash. Le port de sortie pour le hash fait une taille de 64 bits, mais un hash fait 192 bits, de ce fait le module ressort le hash en trois morceaux. Le processus de chiffrement est donc séparé en trois étapes pour chaque morceau du hash, chaque étape consiste en réalité à 2 états, un premier état de préparation et ensuite un état de calcul.

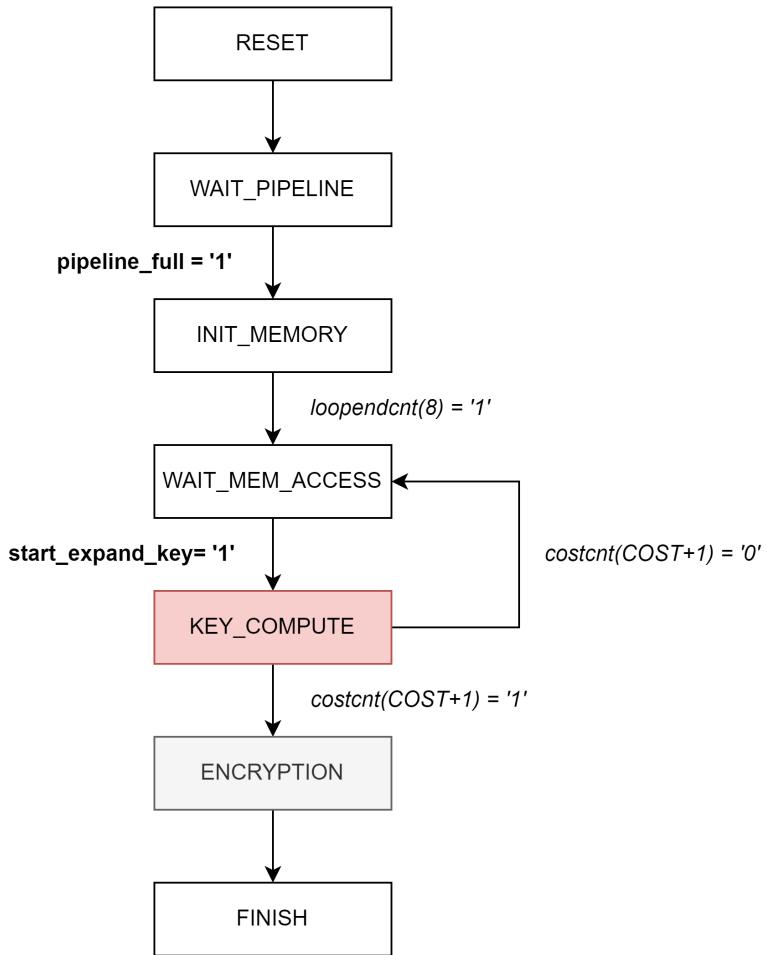


ILLUSTRATION 3.4 – Machine d'état du Bcrypt core simplifié. Source : réalisé par Kandiah Abivarman

Par la suite, j'ai fait des simulations en testant des valeurs dans les différentes entrées afin d'essayer de comprendre les timings attendus par le module. Après ces analyses, j'ai pu comprendre ce que le module attendait en entrée et les différents timings attendus.

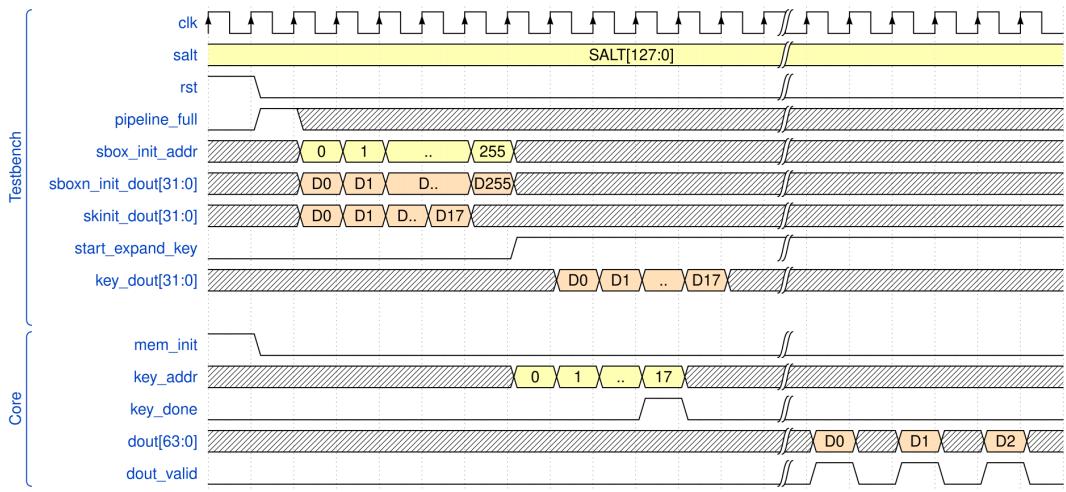


ILLUSTRATION 3.5 – Timing du bcrypt. Source : réalisé par Kandiah Abivarman

Le port *pipeline_full* va permettre de démarrer l’initialisation de la mémoire dans le module. Lors de l’initialisation de la mémoire, il faut fournir au module les adresses mémoires où l’on souhaite écrire avec *sboxn_init_addr* et on va fournir les valeurs initiales des **SBOX** sur *sboxn_init_dout* et des subkeys sur *skinit_dout*. Suite à cela, on va utiliser le port *start_expand_key* afin de démarrer le calcul des clés de chiffrement. Pour le calcul des clés, on va donner à *key_dout* la bonne partie du mot de passe en fonction de l’adresse mémoire fourni par *key_addr*, puis le port *key_done* va signaler la fin des calculs des clés. Pour finir, les différents morceaux du hash sont données par *dout* et le port *dout_valid* va avertir lorsque les différents morceaux sont prêts.

Après analyses des différents timings attendus, j’ai pu refaire le testbench du module bcrypt core est valider son bon fonctionnement.

b. Password Generator

Le password generator à un rôle assez important, car c'est le module qui va s'occuper de générer les différents de mots de passe à tester pour l'attaque.

J'ai donc repris la même démarche que pour le module précédent afin de comprendre le fonctionnement du bloc. C'est à partir de ce module que j'ai commencé à rencontrer des difficultés notamment dû à des constantes qui ont été fixées avec des valeurs sans aucun sens et sans explications. Après quelque temps passé avec le simulateur, j'ai réussi à comprendre le comment marche le module et comment utiliser les constantes et les fixer pour avoir le fonctionnement souhaité.

Dans ce module, les mots de passe sont générés à l'aide de compteur, il y a un compteur par caractère que l'on souhaite générer pour le mot de passe. Chaque valeur de compteur va être convertie en valeur ASCII afin de retrouver les caractères que l'on utilise dans nos mots de passe. La conversion est assez simple, nous avons dans l'ordre l'alphabet en minuscule, l'alphabet en majuscule et les chiffres.

Table de conversion	
0x00	NULL
0x01	'a'
0x02	'b'
0x1B	'A'
0x1C	'B'
0x35	'0'
0x36	'1'

ILLUSTRATION 3.6 – Table de conversion. Source : réalisé par Kandiah Abivarman

Enfin, les différents caractères générés sont concaténés afin d'avoir au final un mot de passe. La fonction de hachage Bcrypt a besoin en entrée un mot de passe de 72 bytes, donc lorsque un mot de passe plus petit est utilisé, on va répéter en boucle le mot de passe en boucle jusqu'à atteindre les 72 bytes. Il est aussi nécessaire de délimiter chaque répétition par un caractère null. Ce module prend en compte ce détail est va s'occuper de remplir les 72 bytes comme il se doit

lorsque le mot de passe généré est plus petit.

Un point à retenir est que lorsque l'on instancie le module, on peut définir le compteur initial et la taille du mot de passe initial. Par exemple, si on initialise le compteur à zéro et la taille du mot de passe à un, nous aurons comme premier mot de passe le caractère 'a' puis 'b' et ainsi de suite. Donc, lors de l'instanciation, il est nécessaire d'initialiser les compteurs intelligemment afin d'avoir l'attaque la plus optimale.

c. Bcrypt Quadcore

Le bcrypt quadcore est le bloc qui va s'occuper d'instancier les deux modules dont j'ai parlé précédemment, c'est aussi dans ce module que j'ai rencontré de nombreux erreurs que j'ai dû corriger afin d'avoir un programme fonctionnel.

Ce module contient le générateur de mot de passe, une **BRAM** pour stocker les mots de passe générés, quatre bcrypt core, une machine d'état et des compteurs.

Le système va tout d'abord avoir un premier état d'initialisation dans laquelle quatre mots de passe vont être générés pour chaque bcrypt core. Après la génération, chaque bcrypt core va calculer le hash en fonction de son mot de passe. Lorsque les calculs sont finis, les hash vont être comparé à l'hash que l'on souhaite casser, lorsque un hash correspond, le module va ressortir le mot de passe correspondant. Toutefois, si les hash ne correspondent pas, alors le système va retourner au premier état d'initialisation. Enfin, après un certain nombre d'essais fixé lors de l'instanciation du module, le système s'arrête.

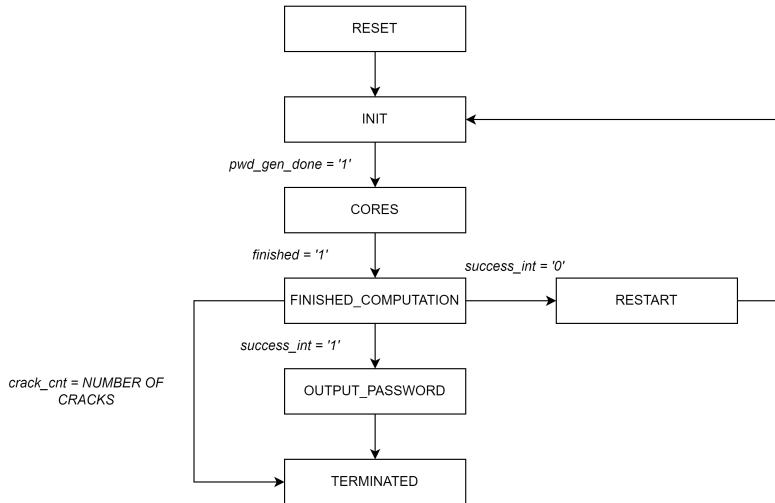


ILLUSTRATION 3.7 – Machine d'état du Bcrypt quadcore simplifié. Source : réalisé par Kandiah Abivarman

d. Bcrypt Cracker

Ce module va s'occuper d'instancier deux BRAM contenant l'état initial des clés de chiffrement qui vont, elles même permettre d'initialiser les BRAM présentes dans les bcrypt core. C'est aussi dans le bcrypt cracker qu'il y a l'instanciation des quadcore, le nombre souhaité de quadcore et le nombre d'essais est réglable dans le code.

Ce module est au final la partie qui va s'occuper du craquage de mot de passe, elle va prendre en entrée le salt et le hash du mot de passe que l'on souhaite retrouver et va ressortir le mot de passe lorsque il est retrouvé.

Afin de bien vérifier le fonctionnement de ce module et du bcrypt quadcore, j'ai utilisé le testbench fourni et j'ai ajouté la vérification de la sortie qui manquait au fichier de test.

Pour ce faire, j'ai regardé au niveau de la simulation afin d'étudier le comportement des sorties du module en fonction des entrées.

Le module à deux entrées qui sont le salt et le hash que l'on souhaite casser et quatre ports de sortie. Il y a d'abord le port *done* qui va permettre d'avertir lorsque le système est dans son état final, c'est-à-dire lorsque il a trouvé le mot de passe ou qu'il a atteint le nombre d'essais maximaux. Ensuite, il y a *success* qui est mis à un lorsque le mot de passe est trouvé. Puis, il y a *dout* qui va nous fournir le mot de passe lorsque il est trouvé et *dout_we* qui est un signal de permission si l'on souhaite écrire le mot de passe dans une mémoire par exemple.

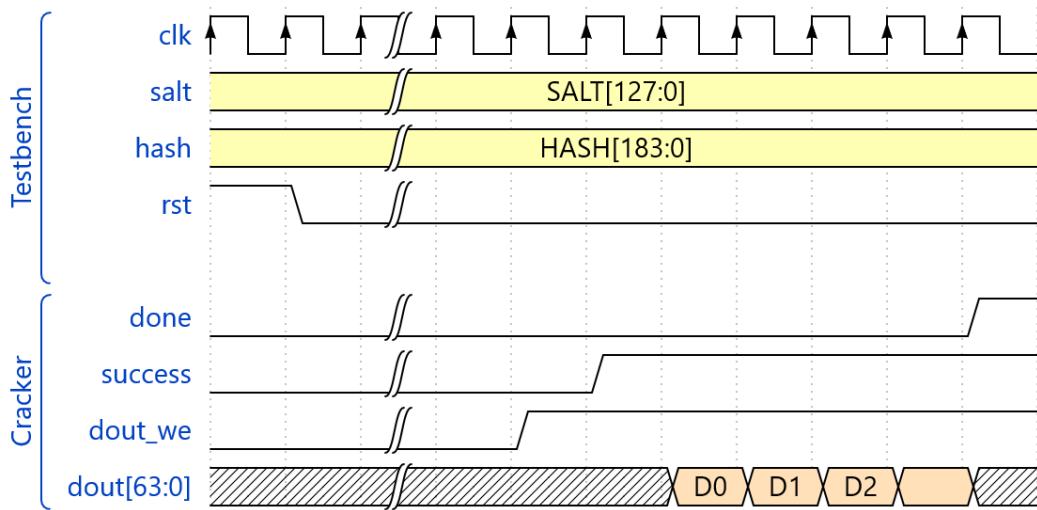


ILLUSTRATION 3.8 – Timing du bcrypt cracker. Source : réalisé par Kandiah Abivarman

Initialement, lors de mes premiers tests, le système semblait fonctionnel. Pour tester le système, je mettais des hash de mot de passe assez simple à casser comme 'a' ou 'b'. J'ai initialisé le compteur du générateur de mot de passe à 0, de ce fait les premiers mots de passe que le module va essayer sont justement 'a', 'b', 'c' et 'd'.

Toutefois, lorsque j'ai testé avec un mot de passe tel que 'z' qui va obliger plusieurs essais au système avant de trouver, le système s'arrête après le premier essai. Le système avait considéré avoir trouvé le mot de passe alors que ce n'était pas le cas. J'ai pu régler ce cas, après avoir identifié une condition incorrecte dans le quadcore. Par la suite d'autres problèmes sont apparus, par exemple des problèmes de réinitialisation de compteur lorsque le système va essayer des nouveaux mots de passe. Tous les problèmes à ce niveau-là provenaient du bcrypt quadcore et ont pu être corrigé grâce à la simulation.

CHAPITRE 3 : IMPLÉMENTATION DES SOLUTIONS

Dans ce chapitre, je vais décrire le fonctionnement et l'implémentation des deux solutions dont j'ai parlé dans le chapitre deux.

4.1. DESCRIPTION SOLUTION UART

Cette première solution va générer les mots de passe directement dans l'**FPGA**, ainsi, on aura besoin du communication **UART** afin de paramétriser l'attaque depuis notre ordinateur. On va notamment pouvoir communiquer le hash et le salt que l'on souhaite casser, ainsi que l'état initial du générateur de mot de passe et le nombre d'essais avant d'arrêter l'attaque.

Cette communication nécessite un encodage permettant de délimiter les paquets, ainsi qu'un moyen de vérification d'erreurs afin de savoir si le paquet n'a pas subi des pertes en chemin.

Afin de tester cette solution, j'ai utilisé une Nexys Vidéo qui est la carte **FPGA** que l'on a utilisé durant nos cours.

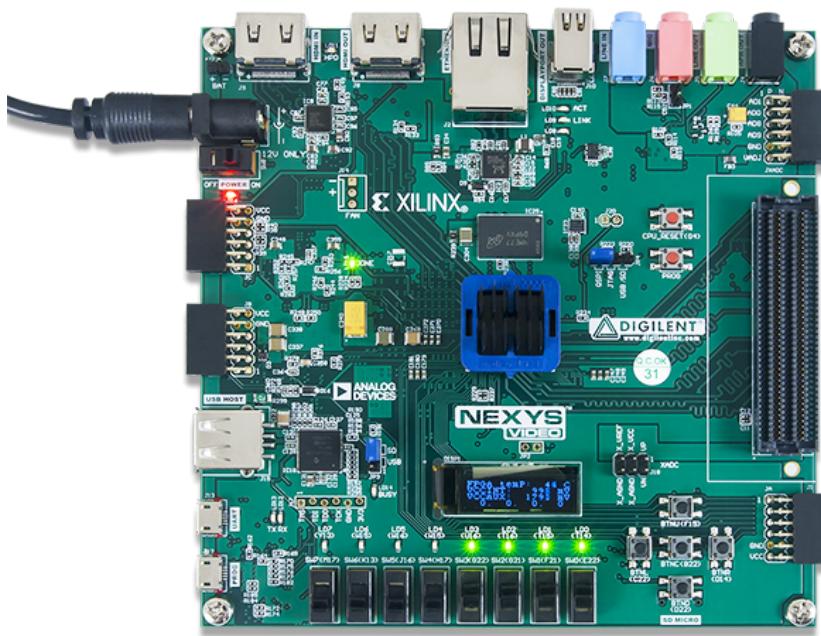


ILLUSTRATION 4.1 – Carte de développement Nexys Video. Source : digilent.com ref. URL02

a. Encodage Cobs

Une manière assez simple d'encoder un paquet et de le délimiter à l'aide de caractères spéciaux. C'est une méthode qui marche assez bien lorsque les données que l'on envoie sont limitées comme du texte par exemple. Toutefois, dans notre cas, nous souhaitons envoyer des données brutes qui peuvent contenir tout type de valeurs.

Il me fallait donc une méthode un peu plus poussée, c'est pour cela que je suis parti sur le **Consistent Overhead Byte Stuffing (COBS)** qui est une méthode d'encodage assez légère et simple d'implémentation.

L'idée du **COBS** est d'utiliser la valeur zéro comme indicatif de fin de paquet. Puis de remplacer tous les zéros présents dans le paquet d'origine par une valeur qui indique à combien d'octets, se trouve le prochain zéro. Un octet est ensuite ajouté au début du paquet pour indiquer la position du premier zéro dans les données originales. Ainsi, le paquet encodé ne contiendra qu'un seul zéro, situé à la fin, qui servira de marqueur pour indiquer la fin du paquet au destinataire. Au final, cette méthode est plutôt simple à mettre en place et rajoute seulement deux octets par rapport au paquet d'origine.

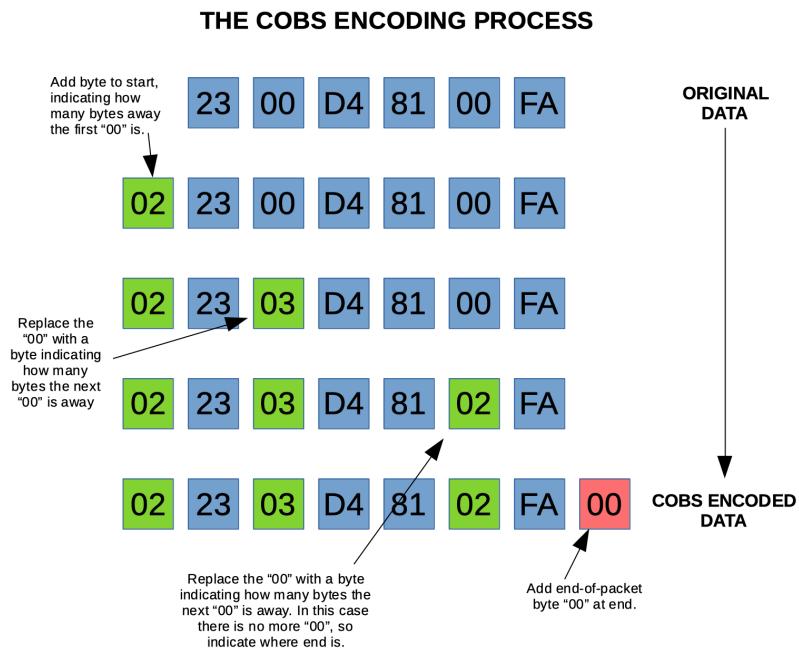


ILLUSTRATION 4.2 – Schéma Encodage COBS. Source : blog.mbedded.ninja ref. URL03

b. CRC

Pour ce qui est de la vérification d'erreurs, l'algorithme utilisé de manière général est le **Cyclic Redundancy Check (CRC)**. Après quelques recherches, je suis tombé sur un site internet¹¹ qui permet de générer du code dont du **VHDL** en fonction du type de **CRC** souhaité. Je suis parti sur le **CRC** le plus simple qui est le 8 bit, mais le code **VHDL** nous est fourni sous forme de module, il est donc assez facile de remplacer le **CRC** par un autre plus rigide.

c. Format et Type de Paquet

Au final, tous les paquets doivent être structurer dans le même format en prenant compte de l'encodage et du **CRC**. J'ai aussi pris l'initiative pour être sûr d'ajouter un champ supplémentaire dans le paquet qui est la taille du paquet d'origine, car le paquet provenant de l'ordinateur devrait toujours être de la même taille.

PACKET FORMAT

1 Byte	1 Byte	Variable	1 Byte	1 Byte
COBS HEAD	PAYLOAD LENGTH	PAYLOAD	CRC	COBS END

ILLUSTRATION 4.3 – Format de paquet - UART. Source : réalisé par Kandiah Abivarman

c.1. Paquet pour le FPGA

Lorsque l'on souhaite faire une attaque, on doit envoyer un paquet pour configurer chaque quadcore. De ce fait, on a qu'un seul type de paquet à envoyer et il faut en envoyer en fonction du nombre de quadcore présent dans le **FPGA**.

11. *Generator for CRC HDL code.* URL : <https://bues.ch/cms/hacking/crcgen> (visité le 18/08/2024).

PAYLOAD FORMAT - BCRYPT QUADCORE INIT

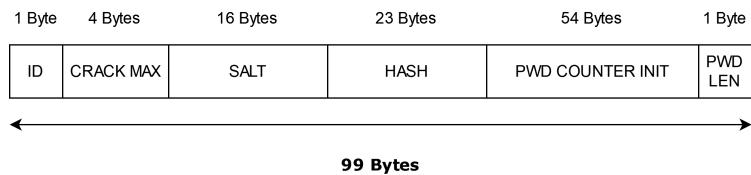


ILLUSTRATION 4.4 – Format de paquet - MOSI. Source : réalisé par Kandiah Abivarman

c.2. Paquet pour l'ordinateur

Lorsque le **FPGA** recevra un paquet, il devrait envoyer un paquet de réponse afin d'avertir l'ordinateur de la bonne réception du paquet.

PAYLOAD FORMAT - RETURN

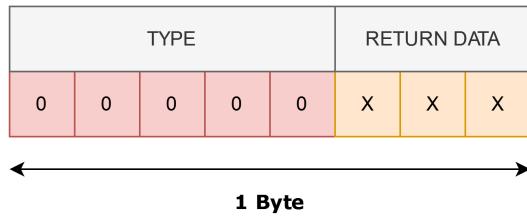


ILLUSTRATION 4.5 – Format de paquet - Retour. Source : réalisé par Kandiah Abivarman

Lorsque le paquet est une réponse, le champ type vaudra zéro et les trois bits de point faibles vont nous indiquer l'erreur si il y en a une.

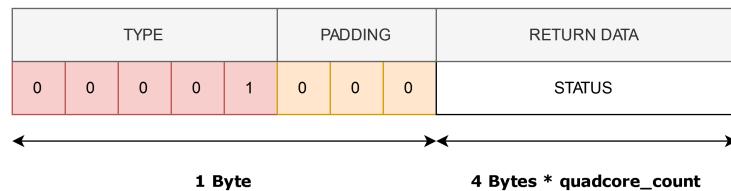
Return Code	Return
000	OK
001	Packet size greater than expected
010	Packet size smaller than expected
011	Quadcore ID not valid
100	CRC Error

TABLEAU 4.1 – Tableau de code d'erreur - Paquet UART

Quand le paquet n'est pas une réponse alors les trois bits vaudront zéros et pourront être ignoré.

En dehors d'une réponse, le **FPGA** envoyera aussi un paquet toutes les secondes pour donner l'état d'avancement de l'attaque. Puis, bien évidemment un paquet sera aussi envoyé lorsque le mot de passe cherché sera trouvé.

PAYLOAD FORMAT - STATUS REPORT



PAYLOAD FORMAT - PASSWORD FOUND

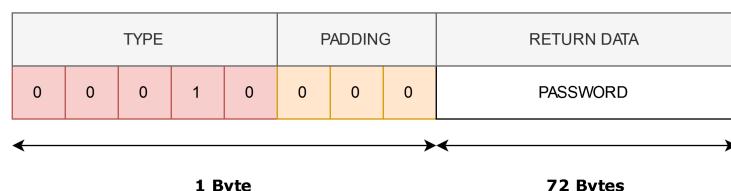


ILLUSTRATION 4.6 – Format de paquet - MISO. Source : réalisé par Kandiah Abivarman

4.2. IMPLÉMENTATION SOLUTION UART

a. Architecture Logique

Pour la partie implémentation, j'avais déjà un module **UART** en **VHDL** qui marchait et le **bcrypt** cracker fonctionnel lorsque réglage était codé en dur. Il fallait donc que je mette en place tout ce qui allait avoir entre l'interface **UART** et le système d'attaque **bcrypt**, c'est-à-dire toute la partie gestion de paquet. J'ai aussi dû adapter le module **bcrypt** cracker et quadcore afin qu'il puisse marcher correctement avec l'architecture souhaitée.

Pour le module qui allait s'occuper des paquets, j'ai pensé à diviser cette partie en deux. Une première partie qui allait s'occuper de la réception des paquets, donc le décodage, la vérification et ressortir les données pour les quadcores. Puis une deuxième partie qui doit gérer toute la partie transmission des paquets de retours, de statut et du mot de passe trouvé. Les deux modules seront reliés de ce fait, lorsque il y aura un souci ou non avec un paquet reçu, le module de réception pourra avertir le module de transmission afin qu'il puisse envoyer le paquet de retours.

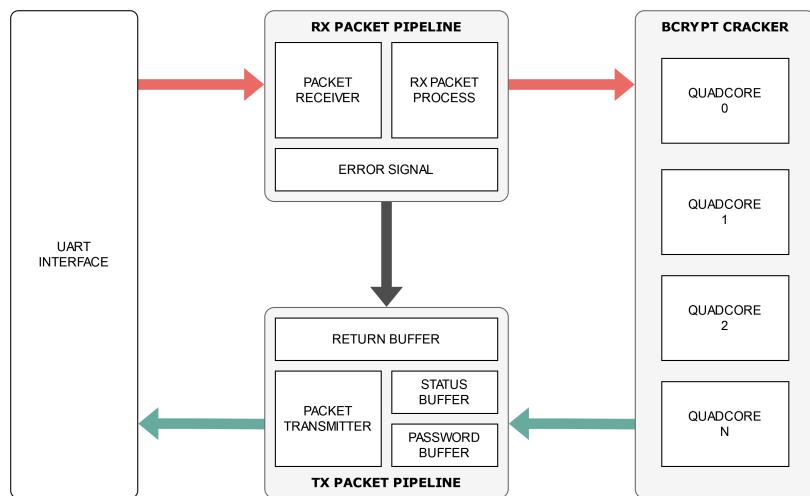


ILLUSTRATION 4.7 – Schéma système UART. Source : réalisé par Kandiah Abivarman

b. Implémentation - MOSI

La gestion de la réception de paquet va être composée tout d'abord d'un module **Packet Receiver** qui va s'occuper de décoder et de calculer le **CRC** et vérifier le **CRC** du paquet reçu. Ensuite, il y a le **RX Packet Process** qui va s'occuper de récupérer les données qui ont été décodées et de ressortir les données dans le bon format pour le **bcrypt** cracker. Pour finir, il y a

le RX Packet Pipeline qui va s'occuper d'instancier les deux modules et de récupérer les erreurs provenant de ceux-ci.

b.1. Module - Packet Receiver

Ce module s'occupe de récupérer byte par byte les données reçus par **UART**, puis de les décoder immédiatement afin de les exposer en sortie. Suite au décodage, le **CRC** sera calculé afin qu'il puisse être comparé avec le **CRC** réceptionné à la fin du paquet. Après vérification du **CRC**, une sortie sera mise à un, afin de signaler le module qui va récupérer les données que ceux-ci sont bien valides.

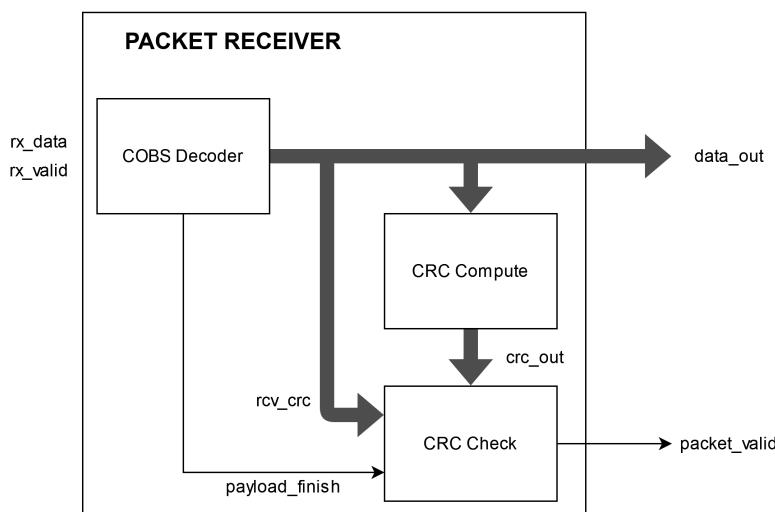


ILLUSTRATION 4.8 – Schéma Packet Receiver. Source : réalisé par Kandiah Abivarman

b.2. Module - RX Packet Process

Ce module va récupérer les données décodées du Packet Receiver et les stockés. Lorsque le module a réceptionné le paquet entier, il va vérifier à l'aide d'un signal émis par le module précédent si le paquet est valide. Si le paquet n'est pas valide, alors les données reçues seront ignorées et le module attendra un nouveau paquet. Toutefois si le paquet est bien valide, d'autres facteurs vont être vérifiés, tels que la taille du paquet ou encore l'adresage du quadcore. En effet, si le paquet ne fait pas la taille attendue ou encore le quadcore ciblé n'est pas valide, alors le module va lui aussi signaler dans une sortie le type d'erreur. Au final, si tout est bon alors, le module va ressortir les données dans le bon format pour le bcrypt cracker et le bcrypt quadcore.

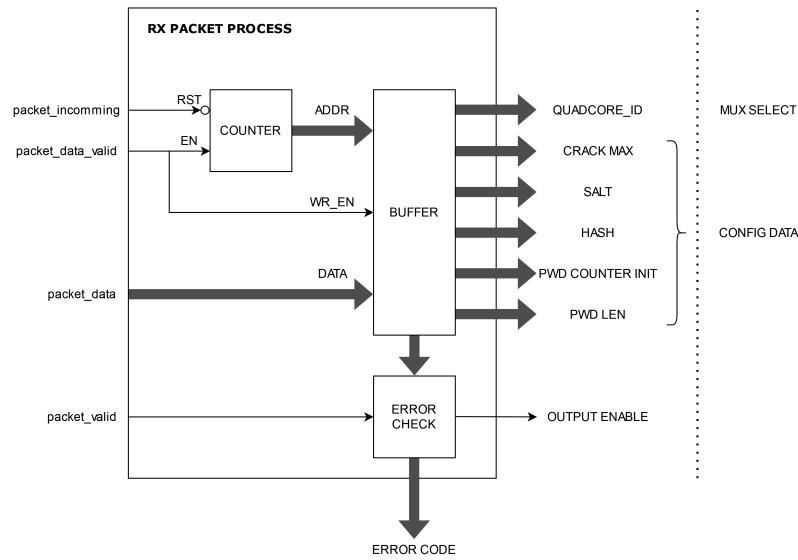


ILLUSTRATION 4.9 – Schéma RX Packet Process. Source : réalisé par Kandiah Abivarman

b.3. Module - RX Packet Pipeline

Le RX Packet Pipeline est un module qui sert principalement à instancier les deux modules précédemment vus. Le module va aussi s'occuper de récupérer les signaux d'erreurs provenant des deux autres modules et les exposer afin qu'elles puissent être utilisées pour le paquet de retour.

c. Implémentation - MISO

La gestion de la transmission des paquets va être scindée en deux modules. On a tout d'abord le TX Packet Pipeline qui est le module qui va décider quel type de paquets va être envoyé. Puis il y a le Packet Transmitter qui à l'inverse du Packet Receiver va s'occuper d'ajouter un **CRC** et d'encoder le paquet afin qu'il puisse être envoyé par **UART**. Le Packet Transmitter est instancié à l'intérieur du TX Packet Pipeline.

Cette partie du travail, a été celui avec lequel j'ai eu le plus de difficultés dans le système au complet.

c.1. Module - TX Packet Pipeline

Ce module va permettre de former les différents type de paquets qui seront envoyés à l'ordinateur.

La logique du module peut être séparée en quatre processus bien distincts :

1. On a un processus qui s'occupe de récupérer le retour du RX Packet Pipeline afin de mettre en place un paquet de retours.
2. L'autre processus va s'occuper de récupérer chaque seconde le nombre d'attaques effectué par chaque quadcore pour en faire un paquet.
3. Ce processus, qui est le plus important, va attendre que le mot de passe soit trouvé puis récupérer le mot de passe afin d'en faire un paquet.
4. Le dernier processus est là pour sélectionner quel paquet va être envoyé, lorsque un ou plusieurs processus sont prêts à envoyer leur paquet.

J'ai aussi mis en place un système de priorité, entre les différents type de paquet. Le paquet de réussite a donc la plus grande priorité, suivi du paquet de retours puis pour finir le paquet de statuts.

c.2. Module - Packet Transmitter

Lorsque le paquet est prêt à être envoyé, ce module va récupérer octet par octet le paquet et le stocker. Durant le stockage du paquet, le **CRC** sera calculé en même temps.

Suite au stockage de l'intégralité du paquet, le **CRC** calculé y sera ajouté et l'encodage **COBS** sera effectué.

Puis lorsque le paquet sera entièrement traité, il sera envoyé à l'interface **UART**.

d. Modifications Bcrypt Cracker

Afin que le bcrypt cracker puisse s'interfacer correctement avec le système de paquets, j'ai dû le modifier lui et le bcrypt quadcore.

À la base, l'état initial du générateur de mots de passe et le nombre d'attaques limite étaient tout deux écrit à la dure dans le code pour chaque quadcore. De ce fait, au démarrage tous les quadcores démarraient immédiatement l'attaque en même temps.

J'ai donc ajouté des ports supplémentaire au bcrypt quadcore afin que l'on puisse configurer tous les éléments importants lors d'une attaque. J'ai aussi ajouté un état de départ qui va attendre que le quadcore soit bien configurer avant de démarrer l'attaque.

Initialement, j'avais prévu que chaque quadcore soit indépendant et que lorsque l'on envoie un paquet pour régler un quadcore, celui-ci lance l'attaque. Au final, je n'ai pas pu faire cela, car tous les cœurs bcrypt ont une phase d'initialisation de mémoire dans laquelle il faut leur remplir leur mémoire avec des valeurs spécifiques. La Logique qui s'occupe de ça est dans le bcrypt cracker et il a été pensé pour que toutes les mémoires soient initialisées en même temps. Par soucis de temps, j'ai préféré garder le fonctionnement de celui-ci tel quel, mais une amélioration possible serait de modifier cela.

e. Vérification Hardware

Afin de valider le système d'attaque avec **UART**, j'ai mis en place un script python me permettant d'envoyer différents types de paquets au **FPGA**. J'ai pu grâce à cela tester le système de réception du **FPGA** puis plus tard le système au complet.

e.1. Interfacage avec PC

La génération et l'envoi des paquets depuis le **PC** ont été fait à l'aide d'un script python. J'ai tout d'abord implémenté une fonction me permettant de générer assez aisément les paquets respectant l'encodage **COBS** et en y mettant un **CRC**. Puis, j'ai mis en place différents paquets de tests, un paquet pour un mot de passe simple à trouver, un paquet pour un mot de passe un peu plus long à trouver et des paquets contenant des erreurs afin de tester la bonne détection d'erreur du **FPGA**.

e.2. Test réception sur **FPGA**

Lorsque j'ai implémenté ce système, j'ai commencé par faire la partie réception du système de paquet. Après validation de celui-ci à l'aide de simulations, j'ai pu tester le bon fonctionnement sur le **FPGA**. Pour ce faire, j'ai tout d'abord tester le contenu des paquets reçus en utilisant celui-ci pour adresser les LEDs sur la carte **FPGA**. J'ai ensuite testé la robustesse de la communication en y adressant les LEDs avec les codes d'erreurs provenant des modules de réception,

en envoyant des paquets à répétition. Tous ces tests m'ont permis de valider le fonctionnement du système de réception de paquets.

e.3. Test système au complet

J'ai commencé le test, en observant le retour des paquets. En effet, j'ai d'abord voulu vérifier que lorsque un paquet est défaillant, je suis bien informé. J'ai donc envoyé les paquets contenant différents types d'erreur afin de vérifier que les retours soient ceux que j'attends.

Le problème étant que je ne recevais pas les paquets de retours. Je recevais bien les paquets de statuts, mais pas les paquets de retours. Il se trouve qu'initialement, je n'avais pas mis de délai d'une seconde entre chaque paquet de statuts. Je soupçonnait donc que mon script python n'avait jamais le temps de lire le retour des paquets, car le paquet se faisait écraser par les paquets de statuts. Après l'ajout du délai d'une seconde, le problème a été résolu et je recevais bien les bons paquets de retour.

Par la suite, j'ai découvert d'autres problèmes tels que le système bcrypt qui lorsque il a trouvé le mot de passe, m'envoyait le mauvais. Ce problème a toujours été présent, mais je n'y avais jamais fait attention, car celui-ci apparaît lorsque un quadcore autre que le premier trouve le mot de passe. Lors de mes tests durant le projet de semestre, tous les quadcores faisait exactement les même attaques, donc lorsque le mot de passe était trouvé, il était trouvé par tous les quadcores, dont le premier. Le problème venait du module **VHDL** tree reduction qui s'occupait de récupérer le mot de passe provenant du quadcore ayant trouvé le mot de passe.

f. Améliorations

Une amélioration possible concerne le déclenchement des quadcores, actuellement synchronisé pour que tous démarrent en même temps pour l'initialisation de la mémoire. Cette synchronisation peut être optimisée en permettant un déclenchement asynchrone, où chaque quadcore commence l'attaque dès que sa configuration est complète. Cela impliquerait de rendre l'initialisation de la mémoire indépendante pour chaque quadcore, ce qui permettrait d'améliorer l'efficacité et de réduire les délais d'attente inutiles.

L'ajout d'une interface utilisateur au script de test pourrait rendre les tests plus intuitifs et accessibles, tout en permettant un suivi en temps réel des résultats.

Lors de l'instanciation d'un grand nombre de quadcores, il a été observé que les contraintes

de timing n'étaient pas atteintes de très peu, malgré le fonctionnement de celui-ci. Bien que le système continue de fonctionner correctement, cela pourrait devenir problématique si des fonctionnalités supplémentaires étaient ajoutées ou si les performances du FPGA étaient poussées à leurs limites. Une optimisation des chemins critiques et une meilleure gestion des ressources pourraient être envisagées pour garantir que les timings restent robustes.

4.3. DESCRIPTION SOLUTION PCIe

Cette deuxième solution a pour objectif de retirer la génération de mots de passe sur **FPGA** et la mettre sur le **PC**. Ainsi, à chaque cycle d'attaque, il faudrait que l'ordinateur fournisse les mots de passe afin que les différents quadcores puissent calculer le hash. Pour ce faire, j'ai besoin d'un moyen de communication très rapide et fiable, c'est pour cela que le **PCIe** est le moyen le plus intéressant pour cette solution.

Afin de mettre en place une communication de ce type, il m'a donc été donné la KCU116 qui est une carte de développement avec un **FPGA** Kyntex Ultrascale+ et une interface **PCIe**.

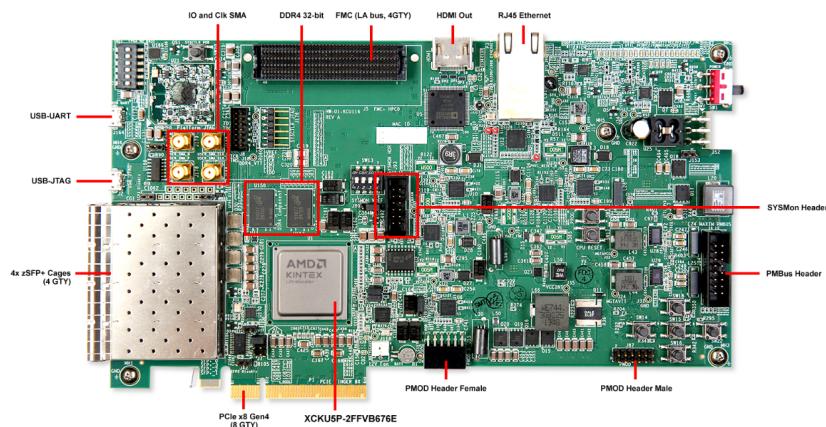


ILLUSTRATION 4.10 – Carte de développement KCU116. Source : xilinx.com ref. URL04

4.4. IMPLÉMENTATION SOLUTION PCIe

a. Architecture Logique

Afin d'interfacer avec le **PCIe** dans le **FPGA**, Vivado nous met à disposition un **IP block** dénommé le **XDMA** qui me permet d'interfacer avec celui-ci à l'aide d'un bus **AXI4**.

Comme pour la première solution, il me fallait mettre en place un module intermédiaire faisant le pont entre le **bcrypt cracker** et le **XDMA**. Ce module **AXI4 Attack Ctrl** va contenir une **BRAM** qui va s'occuper de stocker tous les mots de passe reçus de la part du **PC**, ainsi que des registres qui vont permettre d'interagir avec le **bcrypt cracker**.

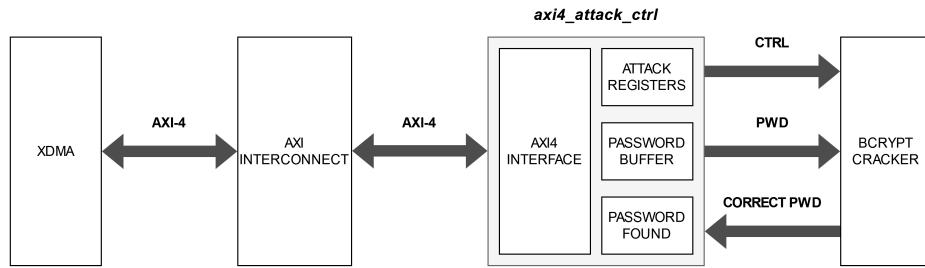


ILLUSTRATION 4.11 – Schéma Système PCIe - FPGA. Source : réalisé par Kandiah Abivarman

b. Implémentation sur FPGA

J'ai très vite rencontré un problème avec l'implémentation, en effet une **BRAM** ne peut avoir que deux ports d'accès mémoire. Le soucis est que j'ai besoin des deux ports afin de transférer le plus rapidement possible tout le mot de passe à la **BRAM** et j'en ai aussi besoin pour transférer les mots de passe au brypt cracker. J'ai donc mis en place un module dénommé **Cracker Regs** qui doit s'occuper de multiplexer l'accès mémoire entre le brypt cracker et le **XDMA** et qui s'occupe du système de registre pour le contrôle du brypt cracker. J'ai aussi utilisé un **IP block** me permettant d'interfacer la **BRAM** avec de l'**AXI4** ce qui va servir pour l'accès provenant de l'**XDMA**.

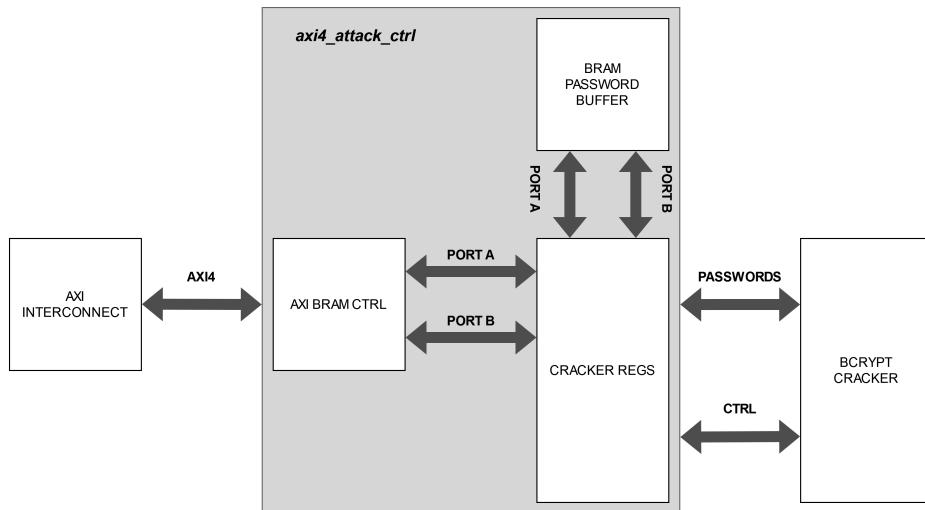


ILLUSTRATION 4.12 – Schéma logique AXI4 Attack Ctrl - FPGA. Source : réalisé par Kandiah Abivarman

b.1. Implémentation Cracker Regs

Le module Cracker Regs doit s'occuper de donner l'accès mémoire soit au PC, soit au bcrypt cracker en fonction de l'état actuel du système.

Initialement le PC aura la main sur la BRAM, alors l'utilisateur pourra envoyer le hash et le salt que l'on souhaite casser et tous les mots de passe nécessaires au bcrypt quadcore présents. Puis lorsque l'on aura écrit sur le registre start, le bcrypt cracker prendra la main sur la BRAM le temps de distribuer tous les mots de passe aux différents quadcores. L'ordinateur n'aura pas le droit d'écrire dans la mémoire et lors d'une lecture, il recevra la valeur actuelle dans le registre start. Lorsque le bcrypt cracker aura fini de récupérer tous les mots de passe, le registre start repassera à zéro et le module Cracker Regs redonnera la main à l'ordinateur pour qu'il puisse envoyer les prochains mots de passe.

Les mots de passe, le hash, le salt et le registre d'attaque sont tous stockés à des adresses spécifiques, les adresses vont dépendre du nombre de quadcores.

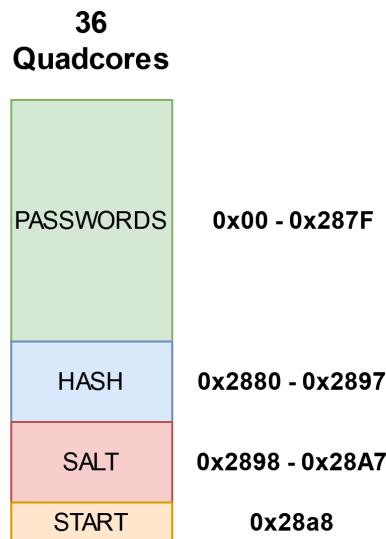


ILLUSTRATION 4.13 – BRAM Adresses - FPGA. Source : réalisé par Kandiah Abivarman

b.2. Modifications Bcrypt Cracker

J'ai tout d'abord modifié le bcrypt quadcore en enlevant le générateur de mots de passe. Puis comme pour la première solution, j'ai ajouté deux états supplémentaires d'attente, une première

qui va attendre de recevoir les mots de passe à tester et une deuxième qui va attendre que les autres quadcores soient prêt pour l'initialisation de la mémoire.

Puis j'ai modifié le bcrypt cracker pour qu'il puisse récupérer les mots de passe et les distribuer au quadcores. J'ai aussi fait en sorte d'avertir le Cracker Regs lorsque tous les mots de passe ont été récupérés.

c. Vérification Hardware

Afin de gagner du temps pour la vérification, j'ai décidé de remplacer le PCIe par un microblaze qui est un IP bloc fourni par Vivado qui fait office de microcontrôleur. Ainsi, le microblaze pourra prendre la place de l'XDMA et je pourrai mettre en place un code C afin de tester aisément le fonctionnement de mon système. Si le système est fonctionnel de cette manière, alors normalement mon système devrait marcher aussi avec le PCIe. Afin de vérifier si le mots de passe a bien été trouvé, j'ai mis les interfaces du bcrypt cracker sur des IP General Purpose Input/Output (GPIO)s qui sont adressable par AXI4 afin d'éviter de trop compliquer le module Cracker Regs.

Malgré de nombreuses heures passées à faire du debug, je n'ai malheureusement pas réussi à faire marcher le système. Par manque de temps, j'ai décidé de tester directement sans passer par la simulation, en utilisant les outils de debug qui sont mis à disposition par Vivado. J'ai pu identifier et réglé quelques problèmes, mais le système n'est toujours pas fonctionnel.

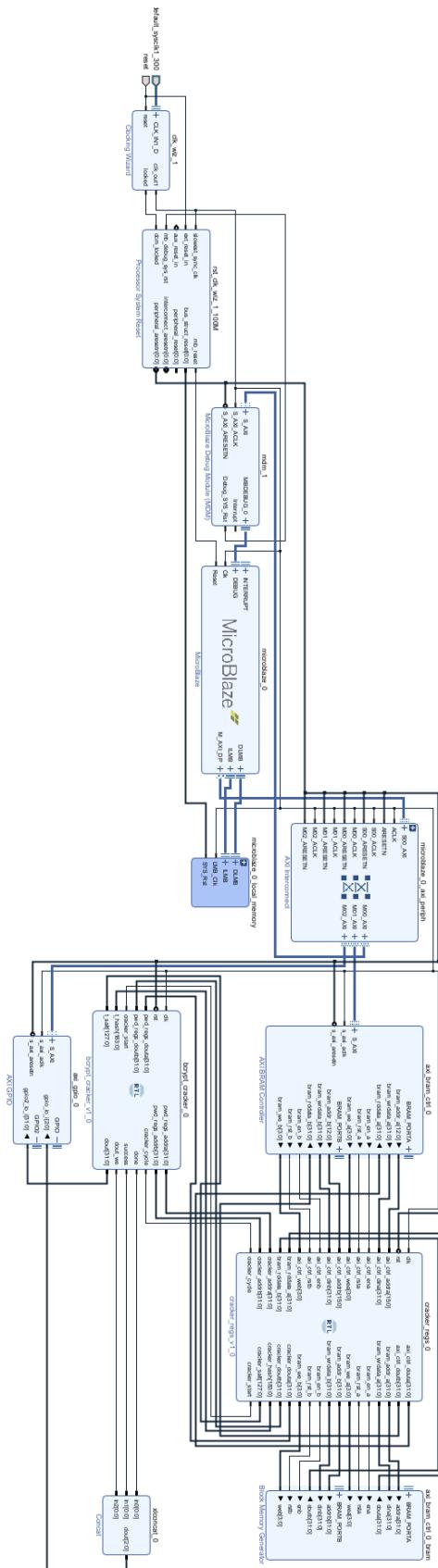


ILLUSTRATION 4.14 – Block Design Test PCIe - FPGA. Source : réalisé par Kandiah Abivarman

CHAPITRE 4 : MESURES ET PERFORMANCES

Toutes les mesures de vitesse sont faites sur l'algorithme Bcrypt avec un cost de 5.

5.1. MESURES CPU

Les performances de l'attaque Bcrypt sur Central Processing Unit (CPU) ont été évaluées en utilisant un programme C multi-threadé. Les tests ont été effectués en augmentant progressivement le nombre de threads (1, 2, 4, etc.) pour observer l'évolution de la vitesse de calcul.

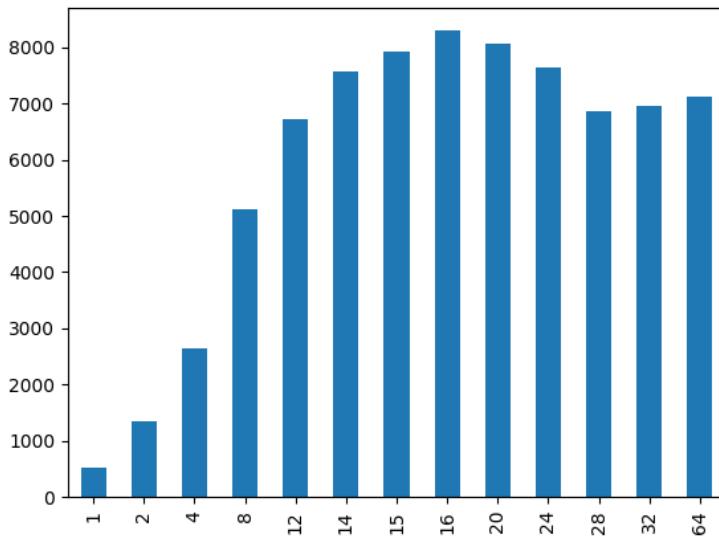


ILLUSTRATION 5.1 – Mesures Bcrypt CPU. Source : réalisé par Kandiah Abivarman

Cette mesure a été faite sur un AMD Ryzen 7 4800U avec 8 Cores et 2 Threads par core. On peut observer que le hashrate le plus élevé est à environ 8200 H/s. Les résultats montrent une augmentation linéaire du hashrate jusqu'à un certain point, après lequel l'amélioration s'arrête en raison de la surcharge de gestion des threads.

5.2. MESURES GPU

Pour évaluer les performances de l'attaque Bcrypt sur GPU, un benchmark a été réalisé à l'aide de Hashcat, un outil largement utilisé pour les attaques de mots de passe. Le test a permis de mesurer le hashrate du GPU, qui s'est avéré être significativement supérieur à celui du CPU, grâce à la capacité du GPU à traiter un grand nombre de calculs en parallèle.

Les performances observées sont directement corrélées avec la puissance de calcul du GPU, ainsi que la mémoire dédiée. Le tableau suivant résume les résultats obtenus avec différentes configurations de GPU.

```
.\hashcat.exe -m 3200 --benchmark
hashcat (v6.2.6) starting in benchmark mode

Benchmarking uses hand-optimized kernel code by default.
You can use it in your cracking session by setting the -O option.
Note: Using optimized kernel code limits the maximum supported password length.
To disable the optimized kernel code in benchmark mode, use the -w option.

CUDA API (CUDA 12.4)
=====
* Device #1: NVIDIA GeForce GTX 1660 SUPER, 5134/6143 MB, 22MCU

OpenCL API (OpenCL 3.0 CUDA 12.4.131) - Platform #1 [NVIDIA Corporation]
=====
* Device #2: NVIDIA GeForce GTX 1660 SUPER, skipped

Benchmark relevant options:
=====
* --optimized-kernel-enable

-----
* Hash-Mode 3200 (bcrypt $2*$, Blowfish (Unix)) [Iterations: 32]
-----

Speed.#1.....: 19201 H/s (66.78ms) @ Accel:4 Loops:32 Thr:16 Vec:1

Started: Fri May 31 18:11:39 2024
Stopped: Fri May 31 18:11:47 2024
```

ILLUSTRATION 5.2 – Mesures Bcrypt GPU. Source : réalisé par Kandiah Abivarman

La mesure a été faite sur un NVIDIA GTX 1660 Super, qui a une puissance maximale de 125 W. Comparé au CPU, le GPU démontre une efficacité accrue pour cette tâche, avec un hashrate à 19201 H/s.

5.3. MESURES FPGA

Les performances de l'attaque Bcrypt sur FPGA ont été mesurées en termes de hashrate et de consommation électrique. D'après mes mesures, un bcrypt core prend 649'225 coups d'horloge pour un hash.

a. Solution UART

J'arrive à instancier jusqu'à 22 quadcores sur cette carte FPGA. Toutefois, la fréquence maximale que j'arrive à mettre est de 100 MHz. De ce fait, pour 22 quadcores, c'est-à-dire 88 Bcrypt cores, on a un hashrate de 13'554 Hash/s.

Toutefois, on pourrait encore ajouter des quadcores, mais nous sommes pas limitées par les

ressources mais par les contraintes de timings.

Resource	Utilization	Available	Utilization %
LUT	101180	134600	75.17
LUTRAM	23	46200	0.05
FF	79027	269200	29.36
BRAM	286	365	78.36
IO	12	285	4.21

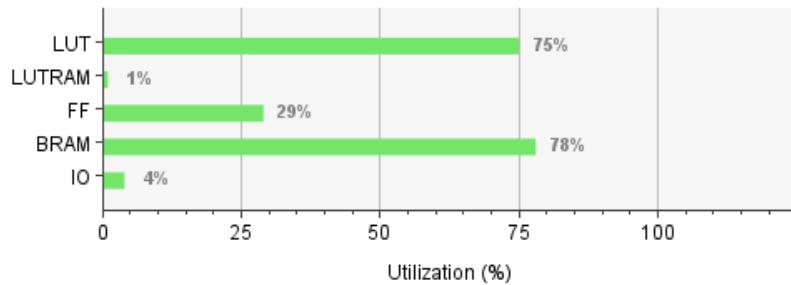


ILLUSTRATION 5.3 – Ressources utilisés Nexys Video. Source : réalisé par Kandiah Abivarman

En termes de puissance dissipés, on retrouve des résultats plutôt bon.

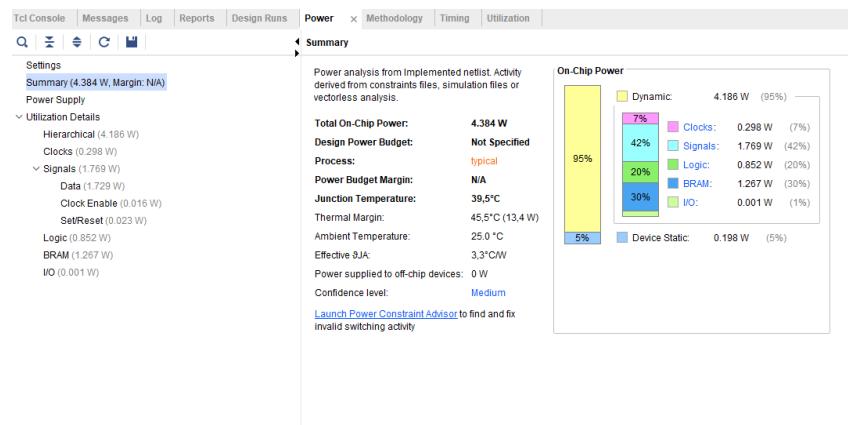


ILLUSTRATION 5.4 – Mesures Puissance Max Nexys Video. Source : réalisé par Kandiah Abivarman

b. Solution PCIe

L'interface PCIe, en revanche, permet d'exploiter pleinement la puissance de l'**FPGA**, avec un hashrate significativement plus élevé par rapport à l'interface UART. En effet, contrairement à la Nexys Video, sur la KCU116, j'ai réussi à monter la fréquence d'horloge.

J'ai donc fait un tableau de mesures faites à différentes fréquences.

Freq (MHz)	Quadcores Max.	Utilisations (%)	WNS	Hashrate (cost : 5)
100	36	BRAM : 97.50, LUT : 68	0.988	22'180 H/s
200	36	BRAM : 97.50, LUT : 68	0.388	44'369 H/s
250	36	BRAM : 97.50, LUT : 68	0.115	55'450 H/s
275	< 30	-	-	< 50'820 H/s
300	5	BRAM : 13.54, LUT : 7.87	0.007	9241 H/s
325	0	0	X	-

TABLEAU 5.1 – Mesures de hashrate sur KCU116

On peut observer qu'à 250 MHz, on atteint un hashrate de 55'450 H/s avec 36 Quadcores.

Cependant, cette solution requiert une consommation électrique plus importante que la solution **UART**.

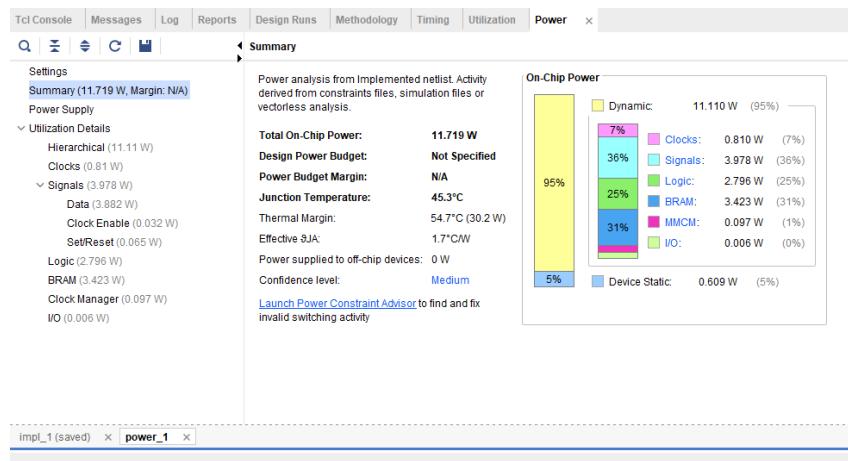


ILLUSTRATION 5.5 – Mesures Puissance Max KCU116. Source : réalisé par Kandiah Abivarman

En effet, on peut voir une puissance Max. de 11.7 W.

c. Comparaisons

Dans cette section, je vais comparer les performances des différentes architectures (**CPU**, **GPU**, **FPGA**) en termes de hashrate et de consommation électrique. Ces comparaisons per-

mettent de mettre en évidence les points forts et les faiblesses de chaque solution pour l'attaque Bcrypt avec un cost de 5.

c.1. Comparaison du Hashrate

Le tableau présente une comparaison des hashrates obtenus pour chaque solution :

Architecture	Hashrate (H/s)
CPU (AMD Ryzen 7 4800U, 16 threads)	8'200 H/s
GPU (NVIDIA GTX 1660 Super)	19'201 H/s
FPGA (Nexys Video, 22 Quadcores, 100 MHz)	13'554 H/s
FPGA (KCU116, 36 Quadcores, 250 MHz)	55'450 H/s

TABLEAU 5.2 – Comparaison des hashrates pour différentes architectures.

Les résultats montrent que le **FPGA** avec interface PCIe à 250 MHz est la solution la plus performante en termes de hashrate, atteignant 55'450 H/s, suivi par le **GPU** NVIDIA GTX 1660 Super avec un hashrate de 19'201 H/s. Le **CPU** et le **FPGA** avec interface UART ont des performances moindres, avec respectivement 8'200 H/s et 13'554 H/s.

c.2. Comparaison de la Consommation Électrique

Le tableau présente une comparaison des consommations électriques maximales mesurées pour chaque solution :

Architecture	Consommation (W)
CPU (AMD Ryzen 7 4800U, 16 threads)	Environ 25 W
GPU (NVIDIA GTX 1660 Super)	125 W
FPGA (Nexys Video, 22 Quadcores, 100 MHz)	4.38 W
FPGA (KCU116, 36 Quadcores, 250 MHz)	11.7 W

TABLEAU 5.3 – Comparaison des consommations électriques pour différentes architectures.

La comparaison montre que le **FPGA**, particulièrement la solution Nexys Video avec interface UART, est de loin la solution la plus économique en énergie, avec une consommation de seulement 4.38 W. Le **GPU** GTX 1660 Super, bien que performant en termes de hashrate, consomme 125 W, ce qui est significativement plus élevé. Le **FPGA** avec interface PCIe, bien qu'il consomme plus que la solution UART, reste plus économique que le **GPU** avec une consommation de 11.7 W.

c.3. Efficacité Énergétique

L'efficacité énergétique, mesurée en termes de Hashes par joule (H/J), est un critère important pour évaluer la performance globale des différentes solutions. Le tableau résume ces données :

Architecture	Efficacité Énergétique (H/J)
CPU (AMD Ryzen 7 4800U, 16 threads)	328 H/J
GPU (NVIDIA GTX 1660 Super)	154 H/J
FPGA (Nexys Video, 22 Quadcores, 100 MHz)	3'094 H/J
FPGA (KCU116, 36 Quadcores, 250 MHz)	4'739 H/J

TABLEAU 5.4 – Comparaison de l'efficacité énergétique (Hashes par joule).

Les résultats montrent que les solutions **FPGA**, particulièrement avec l'interface PCIe, offrent la meilleure efficacité énergétique, atteignant jusqu'à 4'739 H/J. Bien que le **GPU** offre un bon hashrate, son efficacité énergétique est inférieure, à seulement 154 H/J, ce qui le rend moins intéressant pour des applications où la consommation d'énergie est critique. Le **CPU** a une efficacité énergétique légèrement meilleure que le **GPU**, mais reste loin derrière les **FPGA**.

c.4. Consommation Énergétique sur 24 Heures

Le graphique ci-dessous montre l'évolution de la consommation énergétique en wattheures (Wh) pour chaque architecture sur une période de 24 heures.

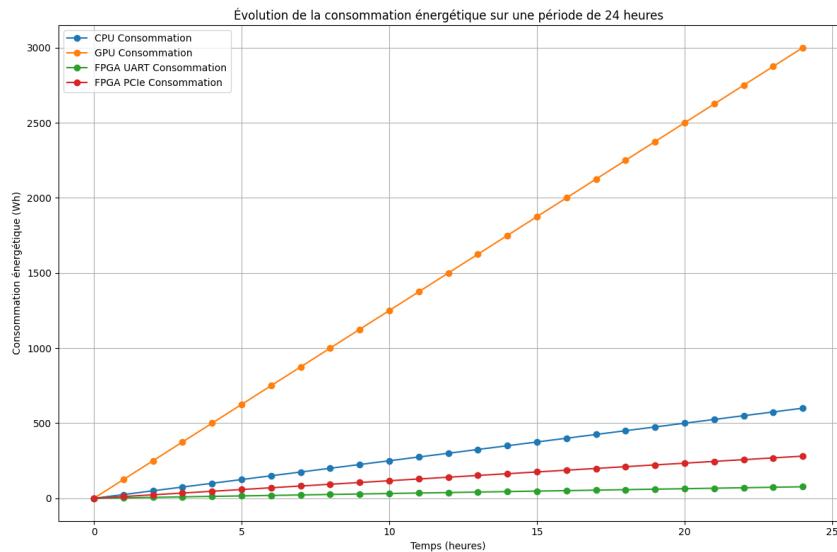


ILLUSTRATION 5.6 – Évolution de la consommation énergétique en Wattheures (Wh) pour le CPU, le GPU, et les solutions FPGA (UART et PCIe) sur une période de 24 heures. Source : réalisé par Kandiah Abivarman

Le graphique montre comment la consommation énergétique s'accumule au fil du temps pour chaque architecture. On peut donc observer l'évolution dans le temps, qui est un facteur important étant donné qu'une attaque par bruteforce peut prendre beaucoup de temps. L'observation la plus flagrante est la différence du GPU par rapport aux autres.

c.5. Conclusion des Comparaisons

En résumé, chaque architecture présente des avantages spécifiques. Le FPGA avec interface PCIe est la solution la plus performante en termes de compromis entre hashrate et efficacité énergétique. Le GPU, malgré son hashrate relativement élevé, présente une efficacité énergétique inférieure aux autres. Le CPU, bien qu'il soit la solution la moins performante, est toujours utile pour des applications généralistes. Le FPGA avec interface UART offre une solution extrêmement économique en énergie, idéale pour un environnement à faible consommation.

CONCLUSION

L'objectif de ce travail de bachelor est d'explorer la possibilité de mise en place d'une attaque de mot de passe bcrypt par bruteforce sur un **FPGA**. Le but étant de chercher une solution plus efficient que les solutions actuelles sur **GPU**.

Pour ce faire, j'ai entamé le projet durant le travail de semestre par une recherche sur le fonctionnement de la fonction de hachage bcrypt et les différentes implémentations existantes. J'ai choisi l'implémentation que j'ai trouvée la plus intéressante et je me suis lancé sur une première phase de simulation. En effet, j'ai commencé par analyser le fonctionnement des différents modules présents et tester à l'aide de la simulation le bon fonctionnement du système. J'ai aussi eu l'occasion de corriger certaines parties du code source afin de le faire fonctionner comme souhaité.

Par la suite, dans le cadre du projet de bachelor, j'ai développé une interface UART permettant l'initialisation des différents cœurs de calcul bcrypt sur la FPGA. Cette interface inclut un système de paquets utilisant un encodage **COBS** et un **CRC** pour gérer les communications entre la FPGA et l'ordinateur, ce qui a permis de recevoir des confirmations et des erreurs et de renvoyer le mot de passe une fois trouvé.

J'avais également prévu de mettre en place une solution PCIe pour améliorer les performances de communication entre la carte FPGA et l'ordinateur. Cependant, en raison des contraintes de temps, cette partie du projet n'a pas pu être finalisée. Malgré les efforts déployés, la solution PCIe n'a pas pu être terminée dans les délais impartis, et le projet a donc été concentré sur la solution UART pour atteindre les objectifs principaux.

Ce projet m'a permis de mettre en pratique les connaissances acquises durant mes cours de **FPGA** et de **VHDL**, notamment sur la partie simulation et l'importance de cette étape cruciale. En travaillant sur une implémentation de l'attaque bcrypt sur FPGA, j'ai eu l'occasion de confronter mes compétences théoriques à des défis pratiques réels.

Une des principales difficultés rencontrées a été de reprendre un code source qui n'était pas bien documenté et qui n'était pas de ma propre conception. Cela m'a contraint à développer des compétences en lecture et en compréhension de code **VHDL** existant. J'ai dû apprendre à déchiffrer, modifier et faire fonctionner ce code selon les objectifs fixés, ce qui a grandement enrichi ma compréhension du fonctionnement des FPGA et de leur programmation.

Le processus de débogage sur la carte **FPGA** a été particulièrement instructif. Il m'a permis de constater les écarts entre la simulation et le fonctionnement réel du matériel. Cette expérience a renforcé ma compréhension des défis associés à l'implémentation matérielle et m'a appris à résoudre des problèmes concrets liés au hardware.

La mise en place de l'interface **UART** a été un aspect intéressant du projet, car elle a nécessité la création d'un système de communication efficace pour l'attaque bcrypt. De plus, bien que je n'aie pas pu finaliser la solution PCIe, le travail préparatoire m'a donné un aperçu précieux sur cette technologie, qui m'a toujours intéressé.

Ce projet a constitué une étape initiale significative dans le développement d'une solution FPGA pour les attaques de mots de passe bcrypt. Toutefois, plusieurs axes d'amélioration peuvent être explorés pour enrichir et étendre cette solution. Une première voie serait simplement de finir de mettre en place la solution **PCIe**, afin d'avoir une solution permettant l'attaque par dictionnaire. Une autre possibilité serait d'améliorer la solution **UART** par exemple, réglant les petits problèmes de contrainte de timing et en améliorant le script de test en python. Il serait intéressant de tenter d'optimiser l'implémentation du bcrypt. En effet, durant mon travail, j'ai pu identifier quelques sections qui pourraient faire le sujet d'optimisation notamment la partie blowfish. Une autre possibilité serait de chercher à rendre la solution la plus extensible possible, afin de mettre en place par exemple un système de cluster de cartes **FPGA** pour augmenter la puissance de calcul. Enfin, étudier la possibilité d'ajouter d'autres fonctions de hachage au système actuel afin de créer une solution beaucoup plus polyvalente.

ANNEXES

ANNEXE 1 - REPO GITLAB

Lien du répertoire Gitlab :

https://gitedu.hesge.ch/abivarma.kandiah/fpga_bruteforce_attack.

RÉFÉRENCES DOCUMENTAIRES

5. *Accessing PCI device resources through sysfs — The Linux Kernel documentation.* URL : <https://docs.kernel.org/PCI/sysfs-pci.html> (visité le 21/03/2024).
- Attaque par dictionnaire.* fr. Page Version ID: 188231625. Nov. 2021. URL : https://fr.wikipedia.org/w/index.php?title=Attaque_par_dictionnaire&oldid=188231625 (visité le 21/03/2024).
- bcrypt.* en. Page Version ID: 1210874707. Fév. 2024. URL : <https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1210874707> (visité le 10/03/2024).
- Blowfish Algorithm with Examples.* en-US. Section: Algorithms. Oct. 2019. URL : <https://www.geeksforgeeks.org/blowfish-algorithm-with-examples/> (visité le 21/03/2024).
- BY. *All Your Passwords Are Belong To FPGA.* en-US. Mai 2020. URL : <https://hackaday.com/2020/05/15/all-your-passwords-are-belong-to-fpga/> (visité le 11/08/2024).
- Generator for CRC HDL code.* URL : <https://bues.ch/cms/hacking/crcgen> (visité le 18/08/2024).
- GILLELA, Maruthi, Vaclav PRENOSIL et Venkat Reddy GINJALA. “Parallelization of Brute-Force Attack on MD5 Hash Algorithm on FPGA”. In : *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. ISSN: 2380-6923. Jan. 2019, p. 88-93. DOI : [10.1109/VLSID.2019.00034](https://doi.org/10.1109/VLSID.2019.00034). URL : <https://ieeexplore.ieee.org/document/8710753> (visité le 10/03/2024).
- Introduction • DMA/Bridge Subsystem for PCI Express Product Guide (PG195) • Reader • AMD Technical Information Portal.* URL : <https://docs.amd.com/r/en-US/pg195-pcie-dma> (visité le 21/03/2024).
- JOSEFSSON, Simon. *The Base16, Base32, and Base64 Data Encodings.* Request for Comments RFC 4648. Num Pages: 18. Internet Engineering Task Force, oct. 2006. DOI : [10.17487/RFC4648](https://doi.org/10.17487/RFC4648). URL : <https://datatracker.ietf.org/doc/rfc4648> (visité le 21/03/2024).
- openwall/john.* original-date: 2011-12-16T19:43:47Z. Mars 2024. URL : <https://github.com/openwall/john> (visité le 21/03/2024).
- rub-hgi/high-speed_bcrypt: VHDL implementation and LaTeX source of "High-Speed Implementation of bcrypt Password Search using Special-Purpose Hardware", published at Re-*

ConFig'14. URL : <https://github.com/rub-hgi/high-speed-bcrypt> (visité le 21/03/2024).

SCATTEREDSECRETS.COM. *Bcrypt password cracking extremely slow? Not if you are using hundreds of FPGAs!* en. Sept. 2020. URL : <https://scatteredsecrets.medium.com/bcrypt-password-cracking-extremely-slow-not-if-you-are-using-hundreds-of-fpgas-7ae42e3272f6> (visité le 20/03/2024).

WIEMER, Friedrich et Ralf ZIMMERMANN. “High-speed implementation of bcrypt password search using special-purpose hardware”. In : *2014 International Conference on ReConfigurable Computing and FPGAs (ReConfig14)*. ISSN: 2325-6532. Déc. 2014, p. 1-6. DOI : 10.1109/ReConfig.2014.7032529. URL : <https://ieeexplore.ieee.org/document/7032529> (visité le 10/03/2024).