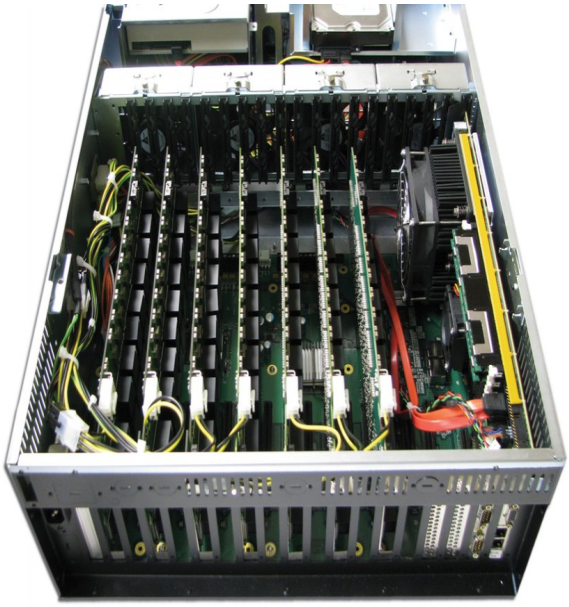
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Bruteforce password attack on FPGAs



Projet de semestre présenté par

Abivarman KANDIAH

Informatique et systèmes de communication avec orientation
Systèmes informatiques embarqués

mars, 2024

Professeur-e HES responsable

Andres UPEGUI POSADA, Stéphane
Gaétan KÜNG

Mandant

ELCA Security

Légende et source de l’illustration de couverture :

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

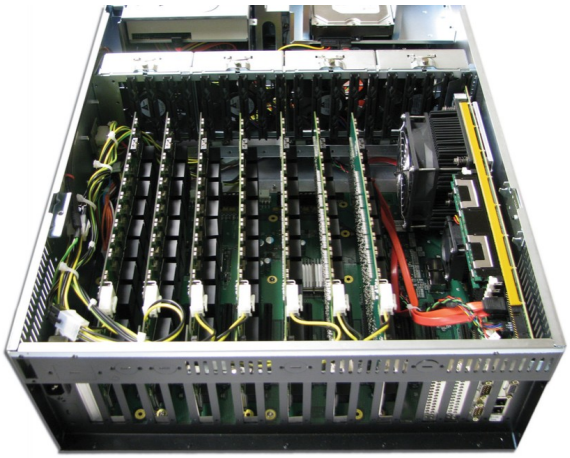
TABLE DES MATIÈRES

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

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. La génération des mots de passe pour le bruteforce est directement réalisée sur le FPGA à l'aide d'un système de compteur pour distribuer les tâches aux cœurs de calcul. Après validation du fonctionnement, des mesures ont été effectuées pour évaluer les performances et les ressources utilisées sur le FPGA. Pour explorer la possibilité de déléguer la génération de mots de passe à un ordinateur, une interface Peripheral Component Interconnect Express (PCIe) a été établie entre la carte FPGA et un ordinateur. Bien que le projet soit fonctionnel, il reste à mettre en place une interface complète entre l'ordinateur et la carte FPGA pour qu'elle soit utilisable lors d'une véritable attaque.



Candidat-e :

ABIVARMAN KANDIAH

Filière d'études : ISC

Professeur-e(s) responsable(s) :

ANDRES UPEGUI POSADA, STÉPHANE GAÉTAN KÜNG

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. 13, 14, 18, 19, 24, 25

FPGA Field-Programmable Gate Array. iv, 1, 2, 3, 7, 8, 10, 21, 23, 25, 26, 28, 29

GPIO General Purpose Input/Output. 21, 22

GPU Graphics Processing Unit. iv, 25, 28

I/O Input / Output. 13

IC Integrated Circuit. 3

IP Intellectual Property. 21, 22

PC Personal Computer. 7, 8, 26, 28

PCIe Peripheral Component Interconnect Express. iv, 2, 21, 22, 26, 27, 28, 29

SBOX Substitution boxes. 8, 16

VHDL Very High Speed Integrated Circuit Hardware Description Language. 3, 10, 28

LISTE DES ILLUSTRATIONS

Références des URL

— URL01 https://www.xilinx.com/products/boards-and-kits/ek-u1-kcu116-g.html

— URL02 https://digilent.com/reference/programmable-logic/nexys-video/start

LISTE DES ANNEXES

INTRODUCTION

Ce travail s'inscrit dans le cadre de mon travail de semestre, réalisé en réponse à une demande d'ELCA Security, une entreprise spécialisée dans la cyber-sécurité. 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. L'idée serait de préparer le terrain, pour que par la suite, je puisse reprendre le travail pour mon travail de bachelor. Ce projet a pour objectif final de leur fournir une solution extensible et performante qui puisse être utilisable lors de leurs attaques. L'intérêt technique et scientifique de ce projet réside dans la recherche de solutions moins énergivores pour l'attaque de mots de passe, en tirant parti des capacités de traitement parallèle offert par les FPGA.

Après une recherche approfondie des implémentations existantes du bcrypt sur FPGA, le projet a débuté par une analyse du papier [1] concernant une implémentation déjà existante. En parallèle, une page Wikipédia détaillant le fonctionnement de bcrypt [2] a été utilisée comme ressource principale pour comprendre les spécificités de cet algorithme. De plus, un papier sur une attaque MD5 [3] 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.

Dans le cadre de ce projet, j'ai entrepris plusieurs actions significatives. Tout d'abord, j'ai repris le code de l'implémentation existante en VHDL d'un programme d'attaque par bruteforce 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.

1. 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).

2. *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).

3. 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).

En parallèle, j'ai pu brièvement étudier le fonctionnement du PCIe afin d'y mettre en place une simple interface entre une carte FPGA et un ordinateur.

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 recherches qui ont été faites afin de trouver une implémentation existante. Puis, je vais détailler la méthodologie de travail, en exposant les différentes étapes de mise en œuvre du projet, les simulations 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 semestre. 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) dans lequel on peut programmer et interconnecter des circuits logiques. 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 Very High Speed Integrated Circuit Hardware Description Language (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 a 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 compromises, il est extrêmement difficile de retrouver les mots de passe originaux à partir de leurs hachages.

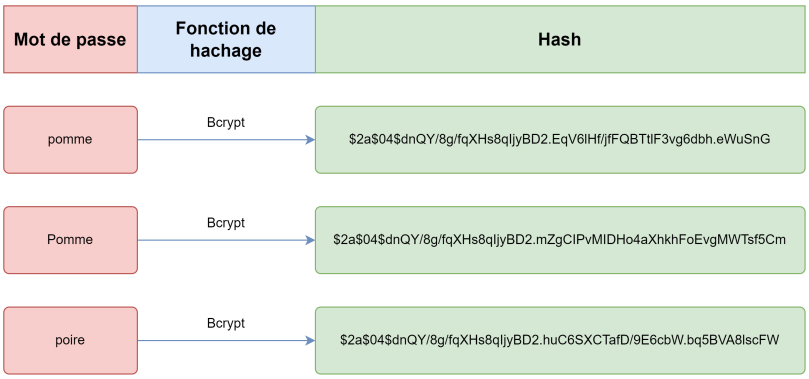


ILLUSTRATION 1.1 – Fonction de hachage. Source : réalisé par Kandiah Abivarman

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.

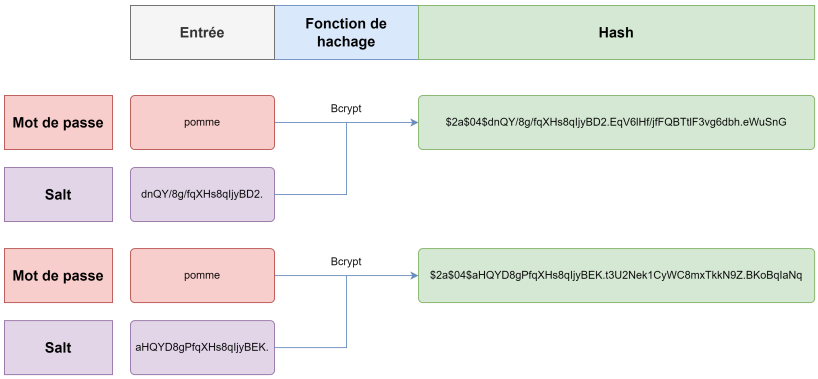


ILLUSTRATION 1.2 – Salt. Source : réalisé par Kandiah Abivarman

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

Ce chapitre a pour but d'expliquer de manière détaillée l'objectif de ce projet de semestre. Je vais aussi faire part des différentes idées qui sont ressorties lors de nos discussions avec mes professeurs. Par la suite, je vais expliquer en quoi consiste la fonction de hachage Bcrypt, son fonctionnement et ses spécificités. Pour finir, je vais rapporter les différentes implémentations sur FPGA que j'ai pu retrouver et celui que j'ai fini par reprendre durant le projet de semestre.

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 lors d'une attaque par bruteforce. 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 le hash qu'il souhaite casser.

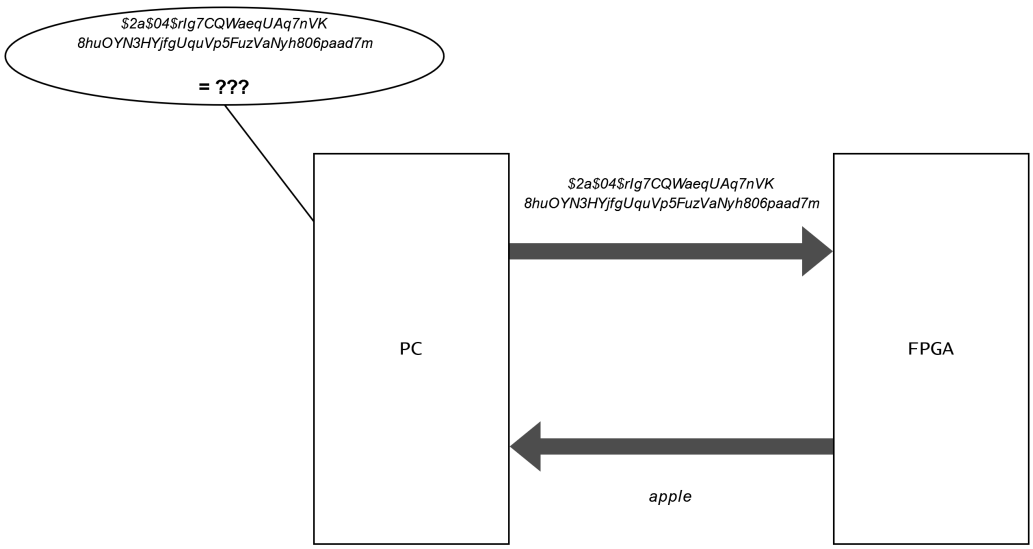


ILLUSTRATION 2.1 – Diagramme général. Source : réalisé par Kandiah Abivarman

Une des première question qu'on s'était posée avec mes professeurs responsables était la manière pour générer les mots de passe lors d'une attaque. Dans notre cas, nous avions deux possibilités, soit nous générons les mots de passe directement depuis le PC et devons les transmettre à la carte FPGA afin de procéder au hachage, soit la génération se fait directement dans le FPGA. La différence est que dans le premier cas, on peut être potentiellement limité au niveau

logiciel par le PC ou au niveau de la communication avec le FPGA. Toutefois, au niveau de la génération des mots de passe, on aura plus de flexibilité permettant d'autres types d'attaques comme par exemple une attaque par dictionnaire [4].

Pour le projet de semestre, j'ai décidé de partir plutôt vers la génération sur FPGA, afin d'avoir une première solution entièrement fonctionnelle sur FPGA sans dépendance avec un PC. La génération sur PC est toutefois envisageable par la suite pour le projet de bachelor.

2.2. 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 Blowfish [5] 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.

4. *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).

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

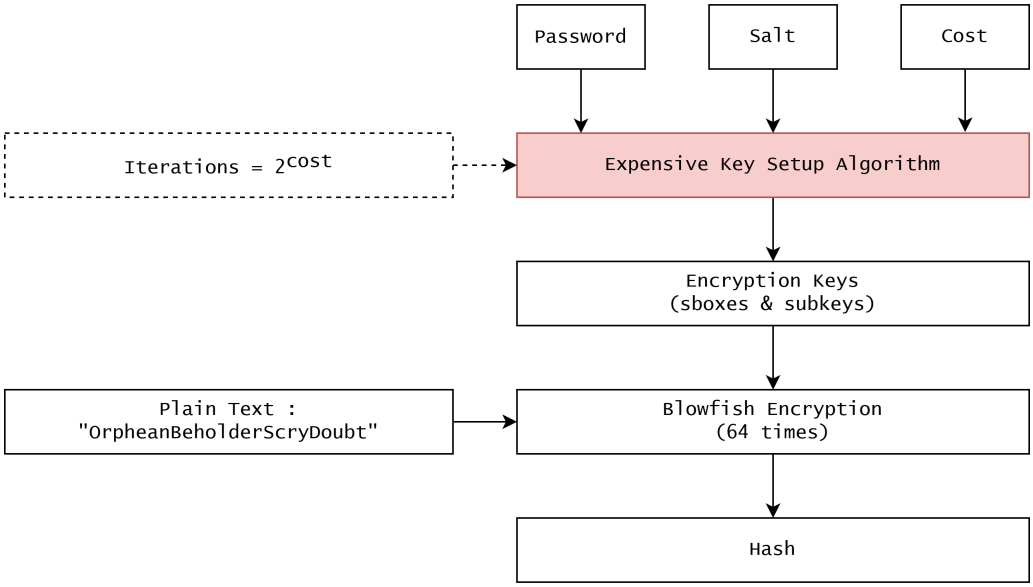


ILLUSTRATION 2.2 – Algorithme Bcrypt. Source : réalisé par Kandiah Abivarman

b. Format du Hash

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

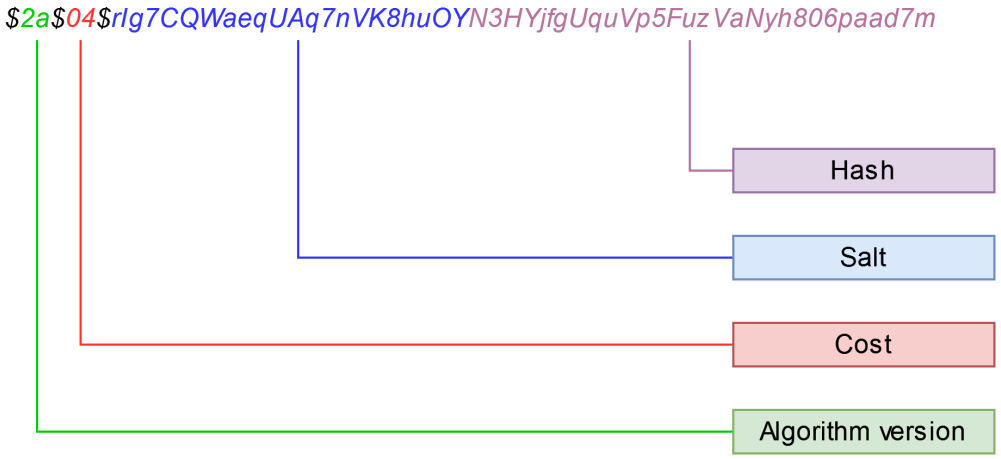


ILLUSTRATION 2.3 – Format du hash Bcrypt. Source : réalisé par Kandiah Abivarman

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 4648 [6] qui est couramment utilisé.

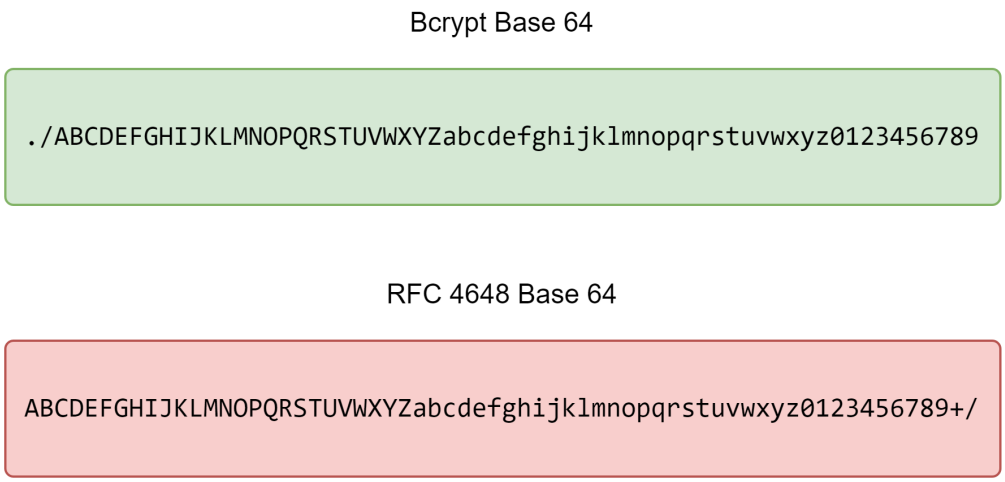


ILLUSTRATION 2.4 – Différence Base 64. Source : réalisé par Kandiah Abivarman

2.3. 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 [7] 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 [8], accompagné d'un papier [9] 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.

6. 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).

7. *openwall/john*. original-date: 2011-12-16T19:43:47Z. Mars 2024. URL : https://github.com/openwall/john (visité le 21/03/2024).

8. *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).

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

Le papier venant avec le code source 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 partie du code contenait pas mal d'erreur, des fichiers de tests étaient incomplets et des fichiers source semble avoir été retravaillé en aval. Au final la plupart des fichiers de tests qui ont été fournis n'était plus utilisables.

CHAPITRE 2 : CONCEPTION

Je vais maintenant expliquer en détail les travaux qui ont été faits autour du code source que j'ai repris.

3.1. BCRYPT SUR FPGA

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

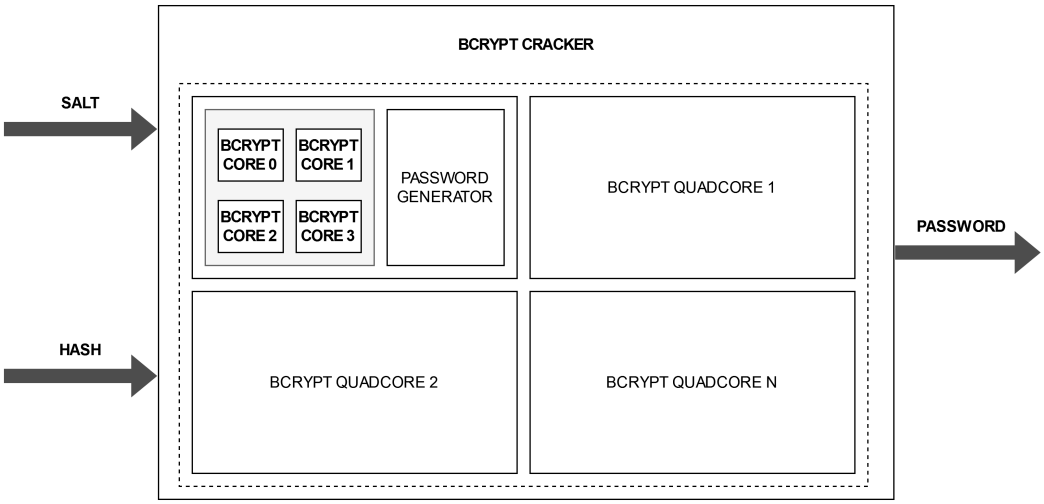


ILLUSTRATION 3.1 – Architecture Bcrypt sur FPGA. Source : réalisé par Kandiah Abivarman

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 ou j'ai analysé le code afin de comprendre l'interface du bcrypt core.

J'ai pu notamment identifier les Input / Output (I/O) qui permettant 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 cost de la fonction de hachage est lui fixé par une constante situé dans un fichier à part regroupant d'autres constantes du système.

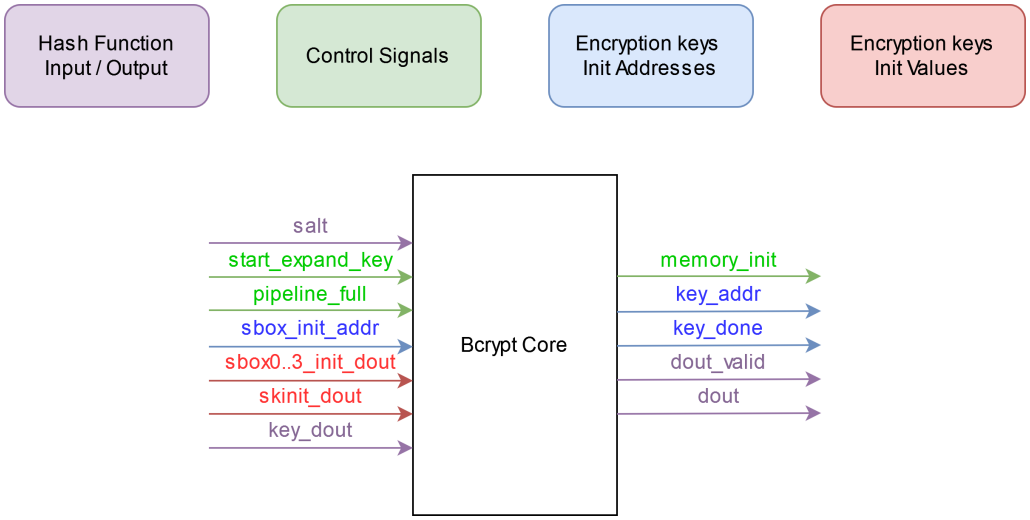


ILLUSTRATION 3.2 – Interface du Bcrypt core. Source : réalisé par Kandiah Abivarman

Après une identification des I/O, j'ai examiné les différents processus et instanciations 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.

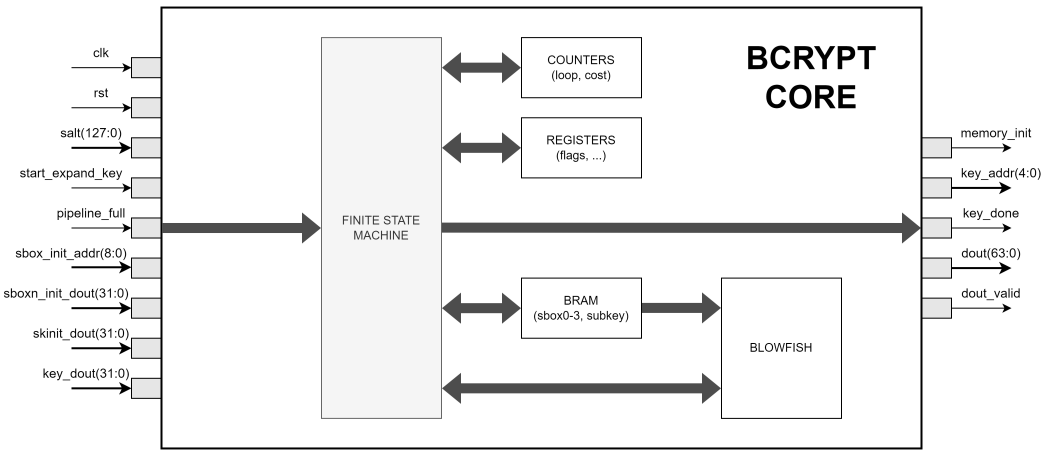


ILLUSTRATION 3.3 – Schéma du Bcrypt core simplifié. Source : réalisé par Kandiah Abivarman

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.

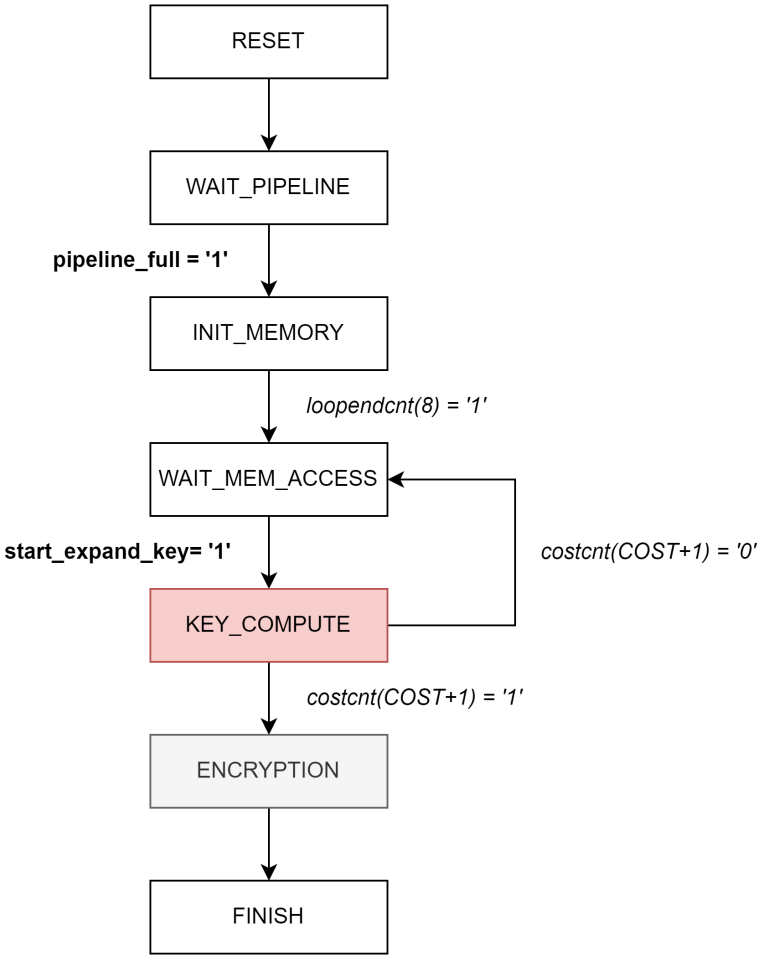


ILLUSTRATION 3.4 – Machine d'état du Bcrypt core simplifié. Source : réalisé par Kandiah Abivarman

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.

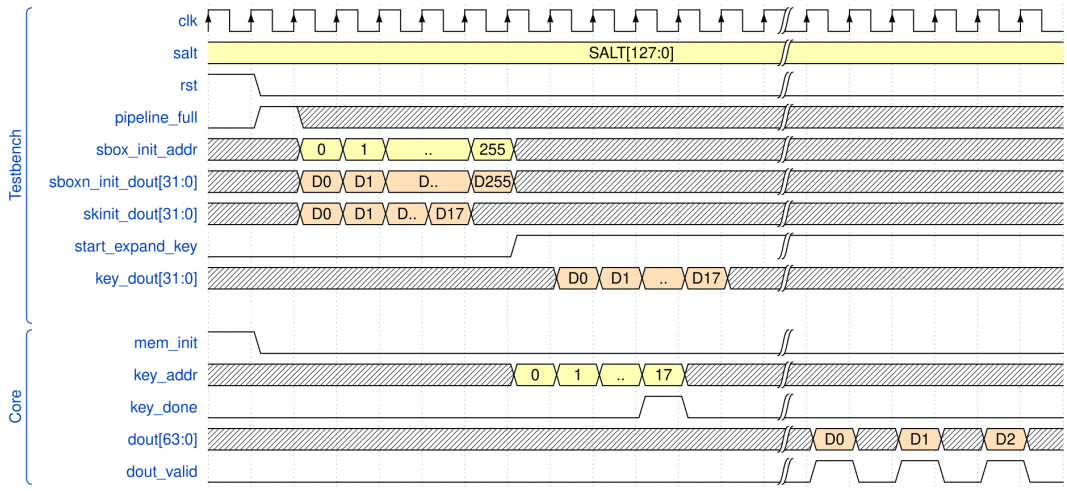


ILLUSTRATION 3.5 – Timing du bcrypt. Source : réalisé par Kandiah Abivarman

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 *sbox_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 précèdent 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 généré, 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.

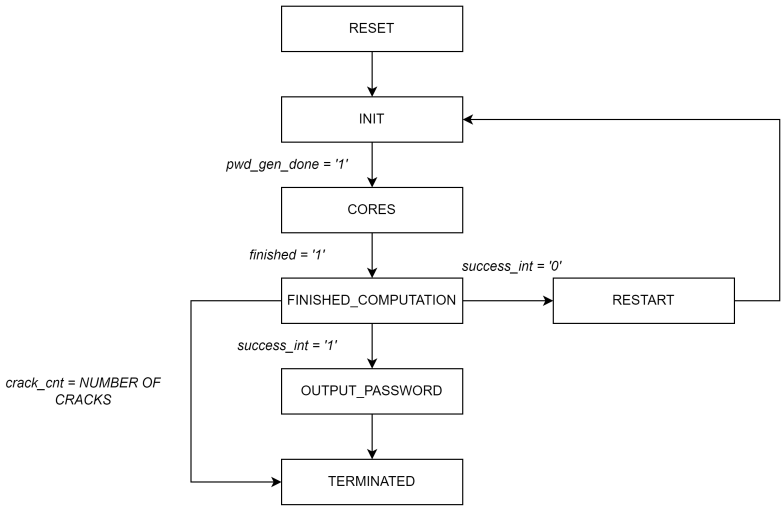


ILLUSTRATION 3.7 – Machine d'état du Bcrypt quadcore simplifié. Source : réalisé par Kandiah Abivarman

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.

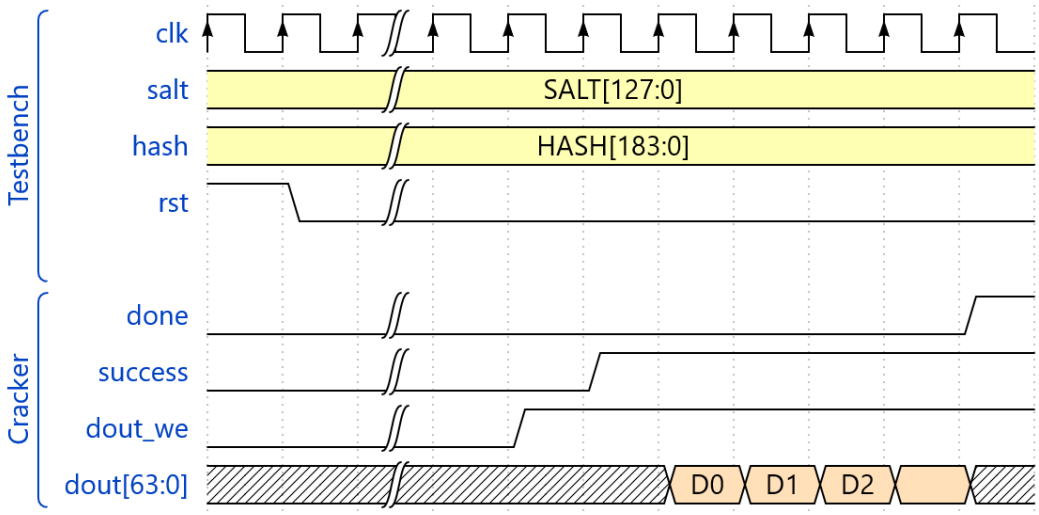


ILLUSTRATION 3.8 – Timing du bcrypt cracker. Source : réalisé par Kandiah Abivarman

Initialement, lors de mes premiers tests, le système semblait fonctionnel. Pour tester le système, je mettais des hash de mot de passe assez simple a 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.

3.2. INTERFACE PCIE SUR FPGA

Cette partie est une déscription de l'implémentation d'une interface PCIe sur FPGA. 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.

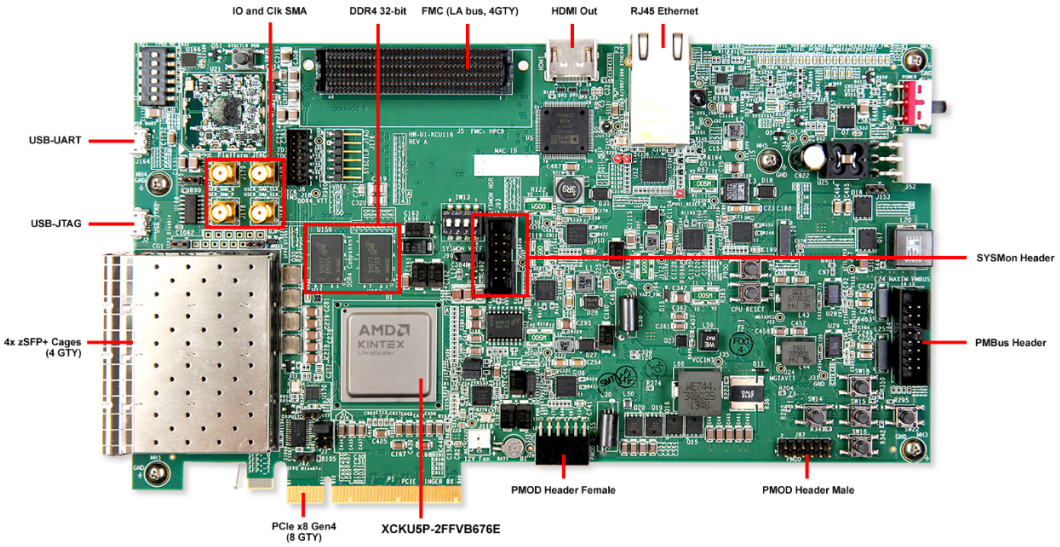


ILLUSTRATION 3.9 – Carte de développement KCU116. Source : xilinx.com ref. URL01

Pour la mise en place de l'interface, j'ai utilisé l'outil design block de Vivado qui permet d'utiliser des Intellectual Property (IP) block fournis et de les interconnecter, pour avoir un rendu sous forme de schéma plutot que du code.

Pour la partie PCIe, j'ai utilisé l'IP *DMA/Bridge Subsystem for PCI Express (PCIe)*, sur lequel j'ai interconnecter des General Purpose Input/Output (GPIO) de la carte (leds et interrupteurs) et des constantes contenant des valeurs que j'ai fixé. L'objectif étant de pouvoir lire l'état des interrupteurs, allumer les leds et lire les constantes.

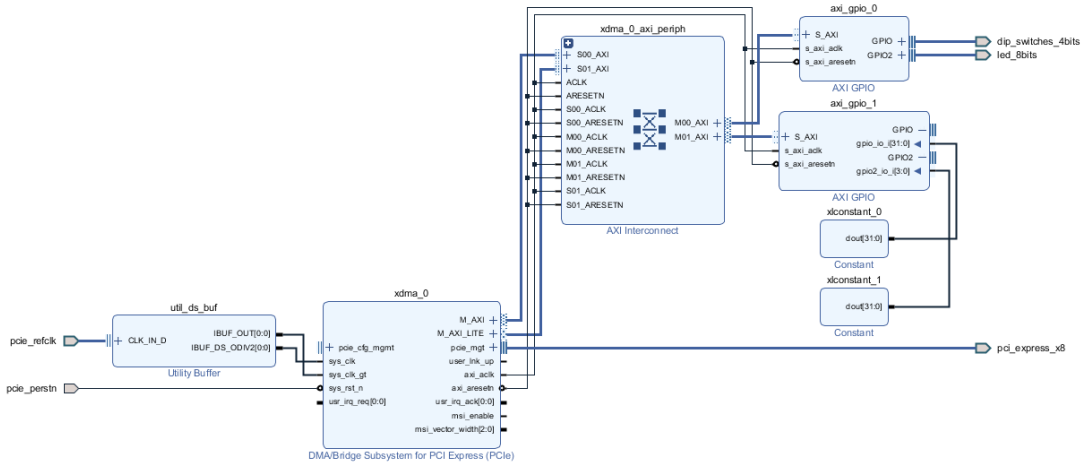


ILLUSTRATION 3.10 – Schéma du Design Block pour le PCIE. Source : réalisé par Kandiah Abivarman

J'ai par la suite configuré l'espace d'adressage du PCIe en fixant des adresses pour les GPIO et les constantes.

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
xdma_0					
M_AXI (64 address bits : 16E)					
axi_gpio_0	S_AXI	Reg	0x0000_0000_4000_0000	4K	0x0000_0000_4000_0FFF
axi_gpio_1	S_AXI	Reg	0x0000_0000_4001_0000	4K	0x0000_0000_4001_0FFF
M_AXI_LITE (32 address bits : 4G)					
axi_gpio_0	S_AXI	Reg	0x4000_0000	4K	0x4000_0FFF
axi_gpio_1	S_AXI	Reg	0x4001_0000	4K	0x4001_0FFF

ILLUSTRATION 3.11 – Espace adressable du PCIe. Source : réalisé par Kandiah Abivarman

Enfin, j'ai configuré l'IP block pour le PCIe, j'ai fixé le nombre de lignes du PCIe au maximum sur la carte qui est 8 et la vitesse de transmission maximale qui est de 8 GT/s. Pour ce qui est de l'identification, j'ai laissé le Vendor ID et le Device ID par défaut (*0x10EE* et *0x9038*). J'ai aussi fixé une addresse de translation de *0x4000_0000*, en faisant cela, lors de l'addressage, l'adresse *0* correspondera à l'adresse que j'ai fixé qui est aussi l'adresse du premier GPIO. Pour finir, j'ai fixé la taille de notre espace d'adressage à 128k.

CHAPITRE 3 : RÉSULTATS

Dans ce chapitre, je vais montrer les résultats que j'ai obtenus des différentes implémentations qui ont été faites au chapitre précèdent.

4.1. BCRYPT CRACKER

Jusqu'à présent, le système d'attaque a été testé seulement à l'aide de l'outil de simulation. Après validation des différents modules à l'aide des testbenchs, j'ai par la suite implémenté le système sur une carte FPGA.

Pour les tests, j'ai utilisé une Nexys Video qui est la carte FPGA que l'on utilise durant nos cours.

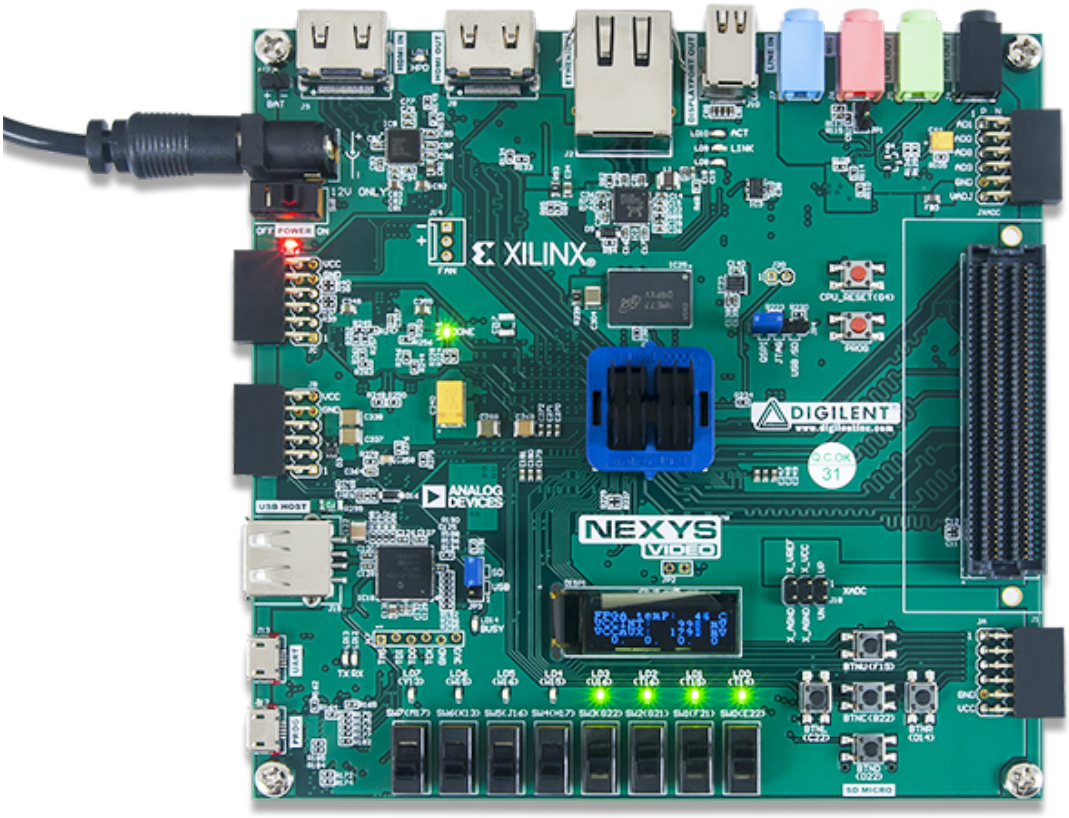


ILLUSTRATION 4.1 – Carte de développement Nexys Video. Source : digilent.com ref. URL02

a. Validation

Afin de le tester sur une carte, j'ai tout d'abord fixé dans le code le salt et le hash du mot de passe que l'on souhaite retrouver. J'ai ensuite connecté les ports *done* et *success* du module sur des LEDS, comme ça lorsque le mot de passe est trouvé, les deux LEDS s'allumeront. J'ai aussi fait en sortie d'allumer une LED, lorsque le système va démarrer.

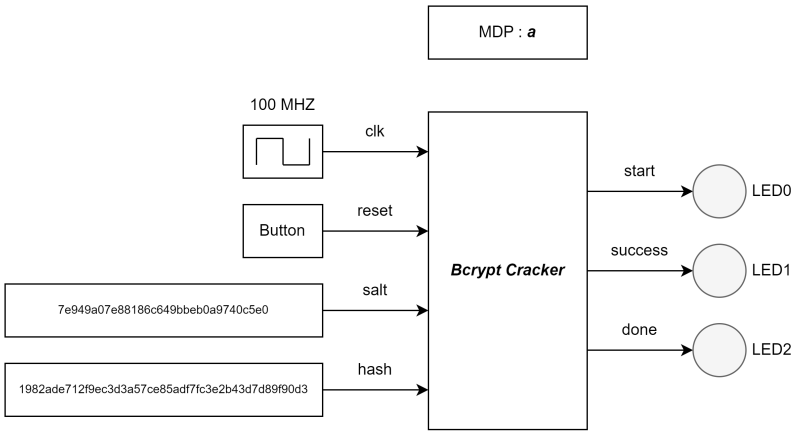


ILLUSTRATION 4.2 – Schéma du test sur carte. Source : réalisé par Kandiah Abivarman

Au premier essai, le système ne fonctionnait pas. En effet, les LEDS *start* et *done* se sont bien allumé, mais la LED *success* ne s'est pas allumé. Cela veut dire que le système s'est arrêté sans trouver, après avoir atteint le nombre maximal d'essai.

J'ai passé un certain temps à debugger, mais je n'arrivais pas à trouver le problème. J'avais néanmoins une piste, un warning me prévenant que certains de mes BRAM étaient retirer lors de la synthèse car inutile. Les BRAM qui ont été retiré sont ceux qui sont utilisés pour stocker les valeurs initiales des clés de chiffrement. Ces BRAM sont initialisé à l'aide de fichier dans laquelle sont stockés les décimales de PI, cette méthode était initialement utilisé pour éviter de polluer visuellement le code. Après le conseil de mon professeur, j'ai enlevé l'initialisation par fichier externe et j'ai tout simplement mis les valeurs directement dans le code.

J'ai ensuite pu retenter et cette fois-ci le programme a bien fonctionné, les trois LEDS se sont bien allumés.

b. Mesures

À l'aide d'un compteur que j'ai mis en place dans mes testbenchs, j'ai pu observer qu'il faut 649'225 coups d'horloge pour hacher un mot de passe avec un cost de 5. Le système tourne à 100 MHz, de ce fait le hachage d'un mot de passe prend environ 6.49 ms, on arrive donc avec seulement un bcrypt core à un taux de hash par seconde de 154.

Dans Vivado, il est possible de récupérer les ressources utilisées par notre programme dans le FPGA :

Resource	Utilization	Available	Utilization %
LUT	4424	134600	3.29
FF	3158	269200	1.17
BRAM	13	365	3.56
IO	5	285	1.75

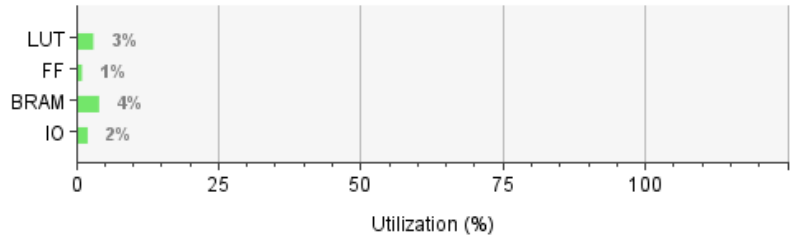


ILLUSTRATION 4.3 – Ressources utilisés par le bcrypt cracker sur la Nexys Video. Source : réalisé par Kandiah Abivarman

Dans ce programme, j'ai instancié seulement un quadcore dans mon système. On peut voir que la ressource la plus utilisée est la BRAM à hauteur de 3.56%. Avec les ressources disponibles, il serait donc potentiellement possible d'instancier 35 quadcore dans ce système.

Donc avec 35 quadcore, c'est-à-dire 140 bcrypt core, on arrive à 21'564 hash par seconde. A titre de comparaison, un GPU tel que le Nvida RTX-2080Ti qui est un GPU haut de gamme a environ 28'000 hash par seconde [10]. Notre résultat est donc assez proche des performances sur GPU.

10. 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).

4.2. INTERFACE PCIE

Cette partie consiste à montrer comment j'ai pu confirmer le bon fonctionnement de l'interface PCIe entre une carte FPGA et un PC.

a. Validation

Pour ce faire, après programmation de la carte, j'ai branché la carte au PC puis j'ai lancé la commande linux *lspci* qui est une commande permettant d'afficher des informations concernant les périphériques PCIe qui sont connectés.

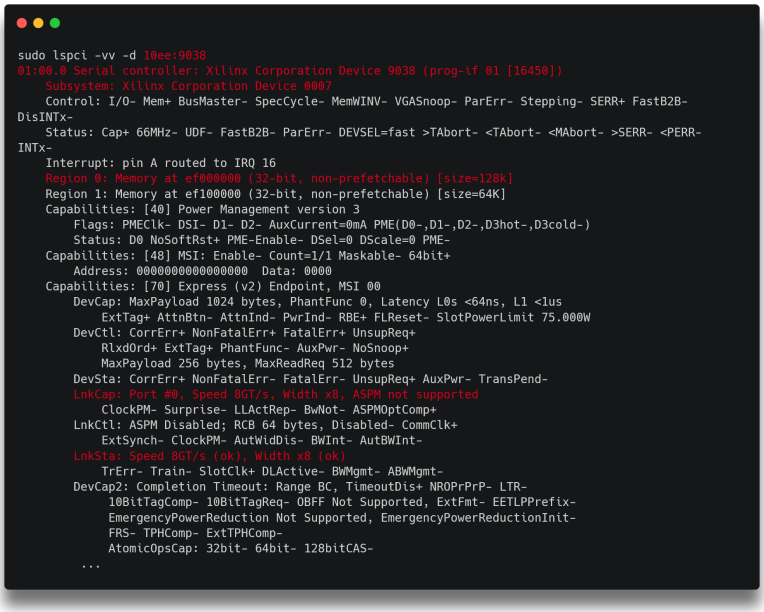


ILLUSTRATION 4.4 – lspci pour observer notre carte fpga. Source : réalisé par Kandiah Abivarman

On peut apercevoir en rouge les différents paramètres que j'ai pu régler dans le chapitre précèdent.

Afin de pouvoir interagir avec mon périphérique, je suis passé par la méthode la plus simple qui consiste à passer par le système de fichier sysfs. Sysfs est un système de fichier linux qui permet à un utilisateur d'interfacer directement avec les différents périphériques connectés au PC. Une autre méthode serait de mettre en place un driver linux, toutefois j'ai décidé de laisser cette méthode si nécessaire pour le projet de bachelor.

Pour tester le bon fonctionnement de mon interface, j'ai fait un petit programme en C qui

va utiliser le fichier *resource0* [11] qui est un des fichier exposé par sysfs. Avec ce fichier, il est possible de directement lire ou d'écrire à l'adresse souhaitée dans le périphérique PCIe.

```c
int main()
{
    uint32_t* bar0;
    int fd;

    fd = open("/sys/bus/pci/devices/0000:01:00.0/resource0", O_RDWR | O_SYNC);

    if (fd < 0)
    {
        perror("test");
        fprintf(stderr, "Failed to open bar0 file\n");
        return -1;
    }

    bar0 = mmap(NULL, 131072, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    close(fd);

    if (bar0 == MAP_FAILED)
    {
        fprintf(stderr, "Failed map bar0\n");
        return -1;
    }

    printf("Etat interrupteurs : 0x%x\n", bar0[0]);

    munmap(bar0, 131072);
    return 0;
}
```

ILLUSTRATION 4.5 – Programme C pour observer l'état des interrupteurs sur la carte FPGA. Source : réalisé par Kandiah Abivarman

11. *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).

CONCLUSION

L'objectif de ce projet de semestre 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 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 partie du code source afin de le faire fonctionner comme souhaité.

Après validation avec la simulation, j'ai entamé une phase de test sur une carte FPGA directement. J'ai pu constater que l'implémentation ne fonctionnait pas sur le matériel et je suis donc passé par une phase de debug. Après l'identification du problème, j'ai pu la régler et valider le bon fonctionnement de l'implémentation sur le matériel.

En parallèle, j'ai utilisé les ressources fournies par Vivado afin de mettre en place une interface PCIe sur une carte FPGA. Par la suite, j'ai pu valider le bon fonctionnement de la communication entre le PC et la carte FPGA.

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'intérêt de celle-ci. J'ai aussi pu me confronter à certaines difficultés, notamment lorsque j'ai repris un code source qui n'est pas forcément bien documenté et qui n'est pas de ma propre conception. Toutefois, ce travail m'a permis d'apprendre à lire et à comprendre du code source VHDL, à le modifier et à le faire fonctionner comme souhaité. De ce fait, ma compréhension du fonctionnement d'un FPGA et de la programmation de celui-ci a été grandement amélioré. La partie de debug sur la carte FPGA a aussi été intéressante, car j'ai pu faire face à la différence entre la simulation et le matériel. Une partie qui m'a beaucoup plu est la mise en place de l'interface PCIe, car c'est une technologie qui m'a toujours intéressé et je n'avais jamais eu l'occasion de m'y frotter.

Ce projet a été une première étape, mais il y a encore beaucoup d'améliorations envisageables notamment pour le projet de bachelor. Une première voie possible serait simplement d'étudier la possibilité d'optimiser l'implémentation existante. 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. Il serait intéressant de creuser l'utilisation de l'interface PCIe pour le transfert de mots de passe à hacher et de résultats. Une solution que je trouve intéressante serait de mettre en place un système séparé en deux parties. Une première partie qui va s'occuper d'attaquer avec des mots de passe généré directement sur la carte et l'autre avec des mots de passe transmise par l'interface PCIe, permettant plus de flexibilité sur la génération de mots de passe. Enfin, étudier la possibilité d'ajouter d'autres fonction de hachage au système actuel afin de créer une solution beaucoup plus polyvalente.

ANNEXES

ANNEXE 1 - REPO GITLAB

Lien du répértoire 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).

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. 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. 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 ReConFig'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).