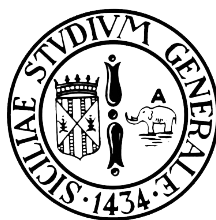
UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA ED INFORMATICA

Corso di Laurea Magistrale in Informatica
Curriculum: Reti e Sicurezza



VERIFIABLE DELAY FUNCTIONS

RELATORE:
Ch.mo Prof. **Dario CATALANO**

CORRELATORE:
Ch.mo Prof. **Mario DI RAIMONDO**

CANDIDATO:
Cristian DANIELE

Anno Accademico 2019/2020

Un ringraziamento particolare va ai Prof. Dario Catalano e Mario Di Raimondo per essere stati presenti nonostante il periodo di emergenza che abbiamo dovuto affrontare. Sempre tempestivi nelle risposte e disponibili, mi hanno permesso di preparare l'elaborato in estrema tranquillità.

Un abbraccio va al mio collega ed amico Giuseppe, per avermi affiancato durante questi cinque anni di studi.

Un grande bacio va sicuramente a mamma, papà, fretellino e nonna, per avermi supportato sempre e per avermi sopportato durante i periodi più impegnativi di questo percorso; e ai nonni che purtroppo non possono assistere a questo mio secondo, piccolo traguardo.

Vi voglio bene.

Abstract

A verifiable delay function (VDF) is a function used to adding delay in decentralized applications. VDF is a function that takes a prescribed time to compute, even on a parallel computer. However once computed, the output can be quickly verified by anyone. In this thesis we will surveys and compares two Verifiable Delay Functions (VDFs) implementation, one due to Pietrzak and the other due to Wesolowski.
We also will talk about a new VDF costruction, Continuous Verifiable Delay Function (cVDF), a function g which is iteratively sequential, where every intermediate state is verifiable. In the end we will compare Pietrzak and Wesolowski protocol benchmark C implementation.

Indice

1 Introduzione **6**

2 Conoscenze di base **7**
2.1 Crittografia moderna 7
2.1.1 Algoritmi simmetrici 8
2.1.2 Funzioni hash crittografiche 8
2.1.3 Algoritmi asimmetrici 9
2.2 Cryptocurrency 9
2.2.1 Blockchain 10
2.2.2 Marcature temporali 10
2.2.3 Mining 10
2.3 Simboli utilizzati 11
2.4 Concetti matematici 11
2.4.1 Numeri primi 11
2.4.2 Pseudoprimi 12
2.4.3 Pseudoprimi forti 12
2.4.4 Distribuzione dei numeri primi 12
2.4.5 Test di primalità 13
2.4.6 Gruppi 13
2.4.7 Gruppi ciclici 14
2.4.8 Gruppi sicuri 14
2.4.9 Funzione di Eulero 15
2.4.10 Fattorizzazione 15
2.5 Multi Precison Integer (MPI) 15

3 Stato dell'arte **17**
3.1 Pricing function con shortcut 17
3.1.1 Definizione pricing function 18
3.1.2 Shortcut 18

3.1.3 Utilità di queste funzioni 19
3.1.4 Extracting square root 19
3.1.5 Fiat-Shamir 20
3.2 Time-lock puzzle 21
3.2.1 Time-lock puzzle basati su ricerca esaustiva . . . 21
3.2.2 Time-lock puzzle basati su elevazioni di potenza . 21

4 Verifable delay function **23**
4.1 Possibili applicazioni 23
4.2 Definizione 26
4.3 Sicurezza delle VDF 27
4.4 Due Verifiable Delay Functions 28
4.4.1 Wesolowski protocol 29
4.4.2 Pietrzak's protocol 30
4.4.3 Confronto fra i due protocolli 31
4.4.4 Costruzione gruppi di ordine ignoto 32

5 Continuous Verifiable Delay Funcitons **33**
5.1 Possibili costruzioni di cVDF 33
5.2 Costruzione di una Continuous VDF 35

6 Implementazione protocolli **38**
6.1 Librerie utilizzate 38
6.1.1 Multiple Precision Arithmetic Library 38
6.1.2 Libreria Nettle 39
6.1.3 Ulteriori librerie 40
6.2 Gestione delle VDF 40
6.2.1 Strutture utilizzate 40
6.2.2 API presenti 41
6.2.3 Funzioni interne alla libreria 43
6.3 Risultati ottenuti 45
6.4 Possibili sviluppi del software 45

Capitolo 1

Introduzione

La tesi ha come obiettivo la presentazione e la spiegazione di uno strumento crittografico recente, formalizzato solo nel 2018, le Verifable delay function. Le VDF sono funzioni particolari che impongono un tempo T per essere computate, ma che permettono di verificare la corretta computazione della stessa in tempi molto brevi.
Queste funzioni hanno suscitato parecchio interesse in quanto utili a risolvere diversi problemi.
Il secondo capitolo di questa tesi sarà interamente dedicato alla raccolta e alla spiegazione di alcuni concetti di matematica, crittografia e informatica in genere, utili alla comprensione del testo.
Nel terzo capitolo si farà un breve riassunto delle soluzioni proposte negli anni per implementare funzioni con le caratteristiche delle VDF.
Nel quarto capitolo verranno presentate formalmente le VDF, saranno proposte due idee per la loro realizzazione, verranno evidenziate differenze e difficoltà implementative.
Il quinto capitolo presenterà delle VDF particolari sequenziali, in cui ogni passo intermedio può essere verificato.
Il sesto ed ultimo capitolo illustrerà come è stato realizzato il programma che implementa le due VDF e confronterà i risultati ottenuti.

Capitolo 2

Conoscenze di base

2.1 Crittografia moderna

La crittografia moderna nasce negli anni 70' con la scoperta dei primi algoritmi asimmetrici. Le prerogative fondamentali che la crittografia moderna garantisce sono:

- Riservatezza: l'informazione deve essere intelligibile solo da chi ne è autorizzato.
- Integrità: deve essere possibile rilevare se l'informazione è stata alterata
- Autenticazione: si deve essere in grado di confermare l'identità di un soggetto all'interno di una comunicazione

Le principali primitive crittografiche su cui si fonda tutta la crittografia moderna sono:

- Gli algoritmi simmetrici
- Le funzioni hash
- Gli algoritmi asimmetrici

Questi costituiscono parte importante nel processo di garanzia delle prerogative suddette, anche se da sole non bastano a garantirle nella loro pienezza.

2.1.1 Algoritmi simmetrici

Sono basati su trasformazioni a blocchi. Tale processo garantisce un'elevata velocità di computazione che li rende particolarmente utili nell'assicurare lo scambio di informazioni riservate. La riservatezza viene assicurata dal fatto che gli interlocutori condividono un'unica chiave segreta con la quale cifrano e decifrano le informazioni.La flessibilità di tali algoritmi è data dalla possibilità di applicarli a messaggi di lunghezza arbitraria, così da permettere lo scambio sicuro di grosse quantità di informazione. Uno degli aspetti negativi di tali algoritmi è la necessità di un sistema sicuro per distribuire le chiavi. Alcuni fra gli standard più diffusi e implementati in svariate modalità sono l'AES che utilizza chiavi a 128 bit, ed il suo antenato, il DES, che utilizza chiavi a 56 bit.

2.1.2 Funzioni hash crittografiche

In informatica una funzione crittografica di hash è una classe speciale delle funzioni di hash che dispone di alcune proprietà che la rendono adatta per l'uso nella crittografia.
Si tratta di un algoritmo matematico che mappa dei dati di lunghezza arbitraria (messaggio) in una stringa binaria di dimensione fissa chiamata valore di hash (o digest). Tale funzione di hash è progettata per essere unidirezionale (one-way), ovvero difficile da invertire. La funzione crittografica di hash ideale deve avere alcune proprietà fondamentali:

- deve identificare univocamente il messaggio, non è possibile che due messaggi differenti, pur essendo simili, abbiano lo stesso valore di hash
- deve essere deterministica, in modo che lo stesso messaggio si traduca sempre nello stesso hash
- deve essere semplice e veloce calcolare un valore hash da un qualunque tipo di dato
- deve essere difficilmente invertibile

Tali caratteristiche permettono alle funzioni crittografiche di hash di trovare ampio utilizzo negli ambiti della sicurezza informatica, quali firme digitali, codici di autenticazione dei messaggi (MAC) e altre forme di autenticazione.[12]

2.1.3 Algoritmi asimmetrici

Sono basati sull'assunzione che ogni utente possieda una coppia di chiavi, una pubblica, nota a tutti, e l'altra privata che è conosciuta solo dal possessore. Tali algoritmi permettono la risoluzione di svariati problemi non risolvibili altrimenti. Gli impieghi più diffusi degli algoritmi asimmetrici sono l' accordo su chiavi ed il non ripudio. Con accordo su chiavi intendiamo il processo in base al quale i due interlocutori si scambiano la chiave simmetrica (segreta) per la cifratura delle informazioni. Tale chiave, essendo un dato ipersensibile, ha bisogno essa stessa di essere cifrata mediante caratteristiche che sono proprie dei due interlocutori e che sono ritenute sicure da entrambi. Tali caratteristiche sono, appunto, la coppia di chiavi (una pubblica e l'altra privata) che identifica in maniera univoca un interlocutore. A questo punto nasce la necessità di avere una terza parte fidata che garantisca il legame fra la coppia di chiavi appartenente ad un dato interlocutore. Tale compito sarà espletato dalla CA (autorità di certificazione). L'altro impiego degli algoritmi asimmetrici è il non-ripudio. Esso garantisce l'autenticazione, nel senso che, dato un messaggio inviato da un utente, tutti gli utenti che lo ricevono e che sono a conoscenza della chiave pubblica dell'utente che lo ha inviato, non possono ripudiare l'autenticità del mittente.

2.2 Cryptocurrency

Il termine cryptocurrency (criptovaluta o criptomoneta in italiano) si riferisce ad una rappresentazione digitale di valore basata sulla crittografia. Le criptovalute utilizzano tecnologie di tipo peer-to-peer su reti i cui nodi risultano costituiti da computer di utenti, situati potenzialmente su tutto il globo. Su questi computer vengono eseguiti appositi programmi che svolgono funzioni di portamonete. Non c'è attualmente nessuna autorità centrale che le controlla; le transazioni ed il rilascio avvengono collettivamente in rete, evitando quindi un approccio di tipo centralizzato.
Il controllo decentralizzato di ciascuna criptovaluta funziona attraverso una tecnologia di contabilità generalizzata (DLT), ad esempio una *blockchain* che funge da database di transazione finanziare pubbliche.[11]

2.2.1 Blockchain

La validità delle monete di ciascuna criptovaluta è fornita da una *blockchain*; un elenco di record in continua crescita (chiamati blocchi) collegati e protetti per mezzo della crittografia.
Ogni blocco contiene in genere un puntatore hash come collegamento al blocco precedente, un *timestamp*, e dei metadati sulle transazioni. Per loro costruzione le *blockchain* sono intrinsecamente resistenti alle modifiche dei dati.
Si possono intendere le *blockchain* come un registro aperto e distribuito che può registrare transazioni tra due parti in modo efficiente, verificabile e permanente. Una volta registrati i dati in un blocco, questi non possono essere infatti esser emodificati retroattivamente senza la modifica di tutti i blocchi successivi. Questo richiede la collusione della maggioranza della rete.
Una *blockchain* è quindi sicura per costruzione, ed implementa un sistema di calcolo distribuito con elevata tolleranza agli errori. Risolve il problema del *double spending*, nonostante la sua natura decentralizzata.

2.2.2 Marcature temporali

La marcatura temporale (*time stamping*) garantisce data e ora certi per un documento informatico al momento della sua apposizione.
Un'*Autorità di Certificazione* firma digitalmente il documento cui è associata l'informazione relativa ad una data e ad una certa ora.
Le criptovalute utilizzano vari schemi di marcature temporali per evitare la necessità di una terza parte attendibile per la registrazione di timestamp aggiuntive alla blockchain. Gli schemi di timestamp utilizzati sono:

- *Proof of work*:
- *Proof of stake*:

2.2.3 Mining

Il *mining* si presenta come uno strumento per la convalida delle transazioni nella rete di una criptovaluta. Il *miner* è un individuo o società che

esegue calcoli complessi in cambio di ricompense da parte della stessa criptovaluta.

2.3 Simboli utilizzati

All'interno della tesi verranno utilizzati, per facilitare la comprensione al lettore, i seguenti simboli:

- $x \leftarrow^{R} G$: Si sceglie un valore x dall'insieme finito G, con distribuzione uniforme

2.4 Concetti matematici

Di seguito verranno ripresi alcuni concetti di algebra che saranno utili alla comprensione dei capitoli successivi.

2.4.1 Numeri primi

In matematica, un numero primo è un numero intero positivo che abbia esattamente due divisori distinti. In modo equivalente si può definire come un numero naturale maggiore di 1 che sia divisibile solamente per 1 e per sé stesso; al contrario, un numero maggiore di 1 che abbia più di due divisori è detto composto. Ad esempio 2, 3 e 5 sono primi mentre 4 e 6 non lo sono perché sono divisibili rispettivamente anche per 2 e per 2 e 3. L'unico numero primo pari è 2, in quanto tutti gli altri numeri pari sono divisibili per 2. Quello di numero primo è uno dei concetti basilari della teoria dei numeri, la parte della matematica che studia i numeri interi: l'importanza sta nella possibilità di costruire con essi, attraverso la moltiplicazione, tutti gli altri numeri interi,nonché l'unicità di tale fattorizzazione . I primi sono inoltre infiniti e la loro distribuzione è tuttora oggetto di molte ricerche. Essi sono rilevanti in molti altri ambiti della matematica pura, come ad esempio l'algebra o la geometria; recentemente hanno assunto un'importanza cruciale anche nella matematica applicata, e in particolare nella crittografia.[15]

2.4.2 Pseudoprimi

Un numero pseudoprimo è un numero che, pur non essendo primo, soddisfa alcune proprietà forti che devono essere necessariamente soddisfatte dai primi, ovvero rispetto a una serie di test si comporta analogamente ad un numero primo. La definizione di numero pseudoprimo dipende quindi dal contesto, e da cosa si intende per *comportarsi come un numero primo*.

2.4.3 Pseudoprimi forti

Sia b un intero, e sia n un intero dispari positivo, non primo, e tali che $b < n$ e $M.C.D.(b, n) = 1$.
Scriviamo $n = 2^s \cdot t + 1$, con t dispari. Il numero n si dice pseudoprimo forte in base b se vale una delle seguenti condizioni:

- $b * t \equiv 1(mod\ n)$
- esiste un $r \in N$, con $r < s$, tale che $b^{2^r \cdot t} \equiv -1\ (mod\ n)$.

In altre parole, n è uno pesudoprimo forte se è uno pseudoprimo per il test di Miller-Rabin.

2.4.4 Distribuzione dei numeri primi

Una volta dimostrato che i numeri primi sono infiniti, sorge spontaneo chiedersi come si distribuiscono all'interno della sequenza dei numeri naturali, cioè quanto sono frequenti e quando ci si può aspettare di trovare l'n-esimo numero primo.
Per ogni numero reale positivo x, si definisca la funzione:

$$\pi(x) := numero\ di\ primi\ minori\ o\ uguali\ a\ x$$

Il teorema dei numeri primi afferma che:

$$\pi(x) \sim \frac{x}{ln(x)}$$

dove $ln(x)$ è il logaritmo naturale di x.

2.4.5 Test di primalità

Un test di primalità è un algoritmo che, applicato ad un numero intero, ha lo scopo di determinare se esso è primo. Non va confuso con un algoritmo di fattorizzazione, che invece ha lo scopo di determinare i fattori primi di un numero: quest'ultima operazione è infatti generalmente più lunga e complessa. I test di primalità più efficienti oggi utilizzati sono probabilistici, nel senso che danno una risposta certa solo quando rispondono NO (ossia quando dicono che il numero è composto) mentre nel caso di risposta SÌ assicurano soltanto un limite inferiore alla probabilità che il numero sia primo. L'errore dei test può essere però reso piccolo a piacere. Il test di probabilità di Miller-Rabin è un test di primalità probabilistico.
Il test è strutturato nel seguente modo.
Fissato un intero dispari > 1, possamo sciriverlo come $n = 2^s \cdot t + 1$ con t dispari. Successivamente procediamo come segue:

- scegliamo a caso un intero b_1, con $1 < b_1 < n$ e calcoliamo $M.C.D.(b_1, n)$
- se $M.C.D.(b_1, n) > 1$, allora n non è primo, ed abbiamo finito
- se $M.C.D.(b_1, n) = 1$, calcoliamo b_1^t $(mod\ n)$. Se $b_1^t \equiv +1\ mod(\ n)$ oppure $b_1^t \equiv -1\ mod(\ n)$, n è primo oppure è pseudoprimo forte in base b_1.
- Se non vale che $b_1^t \equiv +1\ mod(\ n)$ oppure $b_1^t \equiv -1\ mod(\ n)$ calcoliamo b_1^{2t} $(mod\ n)$. Se $b_1^{2t} \equiv -1$ $(mod\ n)$ allora n è pseudoprimo forte in base b_1
- Se non vale che $b_1^{2t} \equiv -1$ $(mod\ n)$, passiamo a tutte le altre potenze di 2, moltiplicate per t.Se tutti i $b_1^{2^r \cdot t}$, per $r = 1, \ldots, s-1$ non sono mai congrui a -1 $mod\ n$, allora n non è un primo. Altrimenti n è uno pseudoprimo forte in base b_1

2.4.6 Gruppi

In matematica un gruppo è una struttura algebrica formata dall'abbinamento di un insieme non vuoto con un'operazione binaria interna (come ad esempio la somma o il prodotto), che soddisfa le proprietà di associatività, di esistenza dell'elemento neutro e di esistenza dell'inverso di

ogni elemento. Tali assiomi sono soddisfatti da numerose strutture algebriche, come ad esempio i numeri interi con l'operazione di addizione, ma essi sono molto più generali e prescindono dalla natura particolare del gruppo considerato. In questo modo diviene possibile lavorare in maniera flessibile con oggetti matematici di natura e origine molto diverse tra loro, riconoscendone alcuni importanti aspetti strutturali comuni. Il ruolo chiave dei gruppi in numerose aree interne ed esterne alla matematica ne fa uno dei concetti fondamentali della matematica moderna. Il concetto di gruppo nacque dagli studi sulle equazioni polinomiali, iniziati da Évariste Galois negli anni trenta del XIX secolo. In seguito a contributi provenienti da altri settori della matematica come la teoria dei numeri e la geometria, la nozione di gruppo fu generalizzata e definita stabilmente attorno al 1870. La moderna teoria dei gruppi, si occupa dello studio astratto dei gruppi. [14]

2.4.7 Gruppi ciclici

In matematica, più precisamente nella teoria dei gruppi, un gruppo ciclico è un gruppo che può essere generato da un unico elemento. Un gruppo G è ciclico se esiste un elemento g del gruppo (detto generatore) tale che G è l'insieme delle potenze di g ad esponente intero, in simboli, usando una notazione moltiplicativa:
$G = \{g^n : n \in Z.\}$
Quindi se $G = \{e, g^1, g^2, g^3, g^4, g^5\}$ allora G è ciclico.

2.4.8 Gruppi sicuri

Di importanza fondamentale nella creazione di un sistema crittografico è la scelta della dimensione del gruppo finito affinché questo sia sicuro contro attacchi.
L'idea utilizzata da Ronald Rivest, Adi Shamir per l'implementazione di RSA è stata quella di ottenere N (dimensione del gruppo) dal prodotto di due numeri p ed n primi e *grandi*. Questo rende molto difficile la fattorizzazione di N.
Si può pensare di usare dei *safe primes*,numeri primi nella forma $2p+1$, come valori di p e q

2.4.9 Funzione di Eulero

In matematica, la funzione ϕ di Eulero o semplicemente funzione di Eulero, è una funzione definita, per ogni intero positivo n, come il numero degli interi compresi tra 1 e n che sono coprimi con n n.
Ad esempio $\phi(8) = 4$ poichè i numeri coprimi con 8 sono 4: 1,3,5,7. La funzione di Eulero riveste un ruolo molto importante nella teoria dei numeri, principalmente perchè è la cardinalità del gruppo moltiplicativo degli interi modulo N.[13]
Un'espressione per la funzione è la seguente:

$$\phi(n) = n \cdot \left[\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_r}\right) \right] = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

2.4.10 Fattorizzazione

In matematica la fattorizzazione consiste, dato un numero intero positivo n, nel trovare un insieme di numeri interi positivi $\{a_0, a_1, a_2 \dots\}$ tali che il loro prodotto sia il numero originario.
Per numeri n molto grandi il problema della fattorizzazione rimane un problema complesso, per questo motivo è sfruttato per diversi modelli crittografici.

2.5 Multi Precison Integer (MPI)

Nell'implementazione pratica di protocolli crittografici, si ha l'esigenza di lavorare con numeri molto grandi (dell'ordine dei 4000 bit). I tipi standard non sono in grado di rappresentare numeri così grandi.

Tipo	**Byte**	**Valori rappresentati**
int	4	Da -2.147.483.648 a 2.147.483.647
unsigned int	4	Da 0 a 4.294.967.295
int64	8	Da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
unsigned int64	8	Da 0 a 18.446.744.073.709.551.615
float	4	3.4E +/- 38 (7 cifre)
double	8	1.7E +/- 308 (15 cifre)

Alla luce di questo, è necessario l'uso di una libreria in grado di gestire numeri potenzialmente grandi quanto tutta la memoria. La libreria che vedremo sarà GMP.

Capitolo 3

Stato dell'arte

Già dal 1992 si è sviluppato l'interesse nel creare funzioni difficili da computare ma veloci da verificare. Quelle che più si sono avvicinate sono state:

- Pricing function con shortcut[5]
- Time-lock puzzles[9]

Vediamo adesso nello specifico cosa prevedono queste due idee, e il motivo per il quale non sono state definitive nella costruzione di VDF.

3.1 Pricing function con shortcut

Questo approccio è nato dell'esigenza di combattere le email di spam e di controllare l'accesso a risorse condivise. L'idea alla base è quella di richiedere all'utente di computare una funzione inerentemente complicata, in modo tale da garantirsi l'accesso alla risorsa. A questo scopo furono proposti tre metodi: estrazione di radice modulo un numero primo, lo schema di firma Fiat-Shamir, e lo schema di firma Ong-Schnorr-Shamir (che è stato però rotto e che quindi non vedremo). Tutti questi metodi sono nati dall'esigenza di imporre un "costo" aggiuntivo nelle transazioni, costo che fungerebbe da deterrente per quanto riguarda l'invio di mail di spam, ma che non interferirebbe con l'uso lecito del sistema. L'idea alla base è di far computare all'utente che vuole accedere alla risorsa una funzione moderatamente difficile (pricing function), ma non intrattabile, del messaggio unito ad altre informazioni (ad esempio un id composto dall'identificativo dell'utente assieme alla data e all'orario della richiesta).

3.1.1 Definizione pricing function

Una funzione f viene definita pricing function[5] quando è:

- moderatamente facile da calcolare
- non parallelizzabile
- dato x e y è facile determinare se y=$f(x)$

Questa funzione deve quindi prevedere una backdoor, in modo tale da risultare meno dispendiosa da calcolare qualora si fornissero informazioni aggiuntive.
La backdoor serve all'eventuale gestore della risorsa per accedere in maniera "economica" alla stessa; bypassando quindi il meccanismo di controllo. La funzione utilizzata per implementare lo schema ha dei parametri che hanno un ruolo analogo ai parametri di sicurezza nei crittosistemi. Si cerca una funzione che richieda, diciamo 0.01 secondi di tempo di CPU per essere valutata sfruttando la backdoor, e diciamo 10 secondi per essere valutata senza sfruttare la sua debolezza. Il termine funzione non è stato fino ad ora ben definito. Per il nostro scopo, spesso indichiamo con il termine funzione una relazione.

- F=$\{f_s\}$ è una famiglia di funzioni indicizzate da s $\in$ S $\subseteq$ $\{0,1\}^*$
- F'=$\{f_s\}$ è una collezione di famiglia di funzioni indicizzate da un diverso parametro k.

E' importante non scegliere funzioni che dopo qualche utilizzo potrebbero essere sempre computate in maniera molto efficiente. L'indice s è formato da un set di l numeri $a_1, a_2, ..., a_l$ con $1 \leq a_1 \leq 2^l$, essendo 2^l moderatamente grande.
Su un certo input x, $f(x)$ è un subset di $a_1, a_2, ..., a_l$ la cui somma fa x.
Computare la funzione deve richiedere tempo proporzionale a 2^l.

3.1.2 Shortcut

La trapdoor (o shortcuts) della funzione è utilizzata per bypassare il meccanismo di controllo di accesso. Diciamo che una collezione di famiglie ha una trapdoor se:

- Esiste un algoritmo che in tempo polinomiale genera una coppia s, c
- f_s è una funzione in F'
- sia c uno shortcut, f_s è computabile più facilmente dato c

Cosa importante è la difficoltà di calcolare c dato s.

3.1.3 Utilità di queste funzioni

La motivazione principale del lavoro svolto da Cynthia Dwork e Moni Naor è stato quello di combattere le email di spam. Il sistema da loro ideato prevedeva una singola funzione f_s, uno shortcut c ed una funzione hash h. Mentre funzione e hash possono essere pubblici, lo shortcut deve essere gestito dall'autorità principale, e ceduto solo ad agenti fidati. Lo scopo è quello di mandare un messaggio m al tempo t ad un destinatario d. Per far questo il mittente deve calcolare $y = f_s(h(m \cdot t \cdot d))$ e inviare y, m, t, d a d. L'autorità principale che si occupa della gestione di questo sistema, avrà il compito di verificare che $y = f_s(h(m \cdot t \cdot d))$. Se la verifica fallisce il messaggio verrà scartato; nel caso invece la verifica abbia successo e il messaggio sia in tempo, verrà solo allora recapitato.

3.1.4 Extracting square root

La prima idea si basa sulla difficolta (ma non sull'impossibilità) di estrarre una radice modulo un numero primo. Non si conoscono però shortcut per questa funzione.

- Indice: Un numero primo p di lunghezza dipendente dal parametro in questione. Una lunghezza ragionevole potrebbe essere 1024 bit
- Definizione di f_p: Il dominio di f_p è Z_p
- Verifica: Dati x, y controllare che $y^2 \equiv$ x $mod\ p$.

Lo step di verifica richiede solo una moltiplicazione. Non sono noti però metodi per estrarre facilmente radici mod p che facciano meno di log p moltiplicazioni.

3.1.5 Fiat-Shamir

Questa implementazione si basa sullo schema di firma di Fiat e Shamir.[1]

- Indice: Sia N il prodotto di due numeri p e q primi e sufficientemente grandi da rendere la fattorizzazione di N non trattabile. Sia $y_1 = x_1^2, ..., y_k = x_k^2$ un k-quadrato modulo N. Sia inoltre h una funzione hash con dominio in $Z_{\mathrm{n}}{}^* \times Z_{\mathrm{n}}{}^*$ il cui codominio è $\{0,1\}^k$. L'indice s è la (k+2)esima tupla $(N, y_1...y_{\mathrm{k}}, h)$

- Shortcut: La radice quadrata $x_1...x_{\mathrm{k}}$

- Definizione di f_s: Il dominio di f_s è Z_N^*. Ecco un algoritmo moderatamente semplice per calcolare z e r^2 in modo tale che soddisfino le seguenti condizioni.
  Sia $h(x, r^2) = b_1...b_k$ con b_i un singolo bit. Vogliamo che z e r^2 soddisfino la seguente condizione:

$$z^2 = r^2 x^2 \prod_{i=1}^{k} y_i b_i mod N$$

  $f_s(x) = (z, r^2)$

- Verifica: Dati x, z, r^2, si computa $b_1...b_k = h(x, r^2)$ e so verifica che

$$z^2 = r^2 x^2 \prod_{i=1}^{k} y_i b_i mod N$$

- Valutazione con lo shortcut: Si sceglie un r random e si computa $h(x, r^2) = b_1...b_k$. Si imposti poi $z = rx \prod x_i b_i fs(x) = (z, r^2)$ $f_s(x) = (z, r^2)$ può essere computata come segue:
  - Si indovini $b_1...b_k \in \{0,1\}^k$
  - Si computi $B = \prod y_i b_i mod N$
  - Ripetere i senguenti passi:
    * Scegliere z random $\in Z_N^*$
    * Definire $r^2 = (z^2/Bx^2) mod N$
  - finchè $h(x, r^2) = b_i...b_k$

3.2 Time-lock puzzle

Introdotti da Rivest, Shamir e Wagner [9] forniscono una cifratura che può essere aperta solo dopo un certo periodo di tempo. Nessuno, neanche il mittente, può rompere la cifratura prima dello scadere del tempo. Lo scopo è quello di *inviare informazioni nel futuro*. L'approccio che più ci interessa è quello di creare un problema computazionalmente difficile, che non può essere risolto senza un'elaborazione continua da parte di una macchina per un certo arco di tempo.

3.2.1 Time-lock puzzle basati su ricerca esaustiva

Sia M il testo in chiaro da voler cifrare per un certo periodo di tempo. Sia S la velocità della workstation misurata in decifrature al secondo. Scegliamo un criptosistema convenzionale, utilizzando una chiave di lunghezza circa $lg(2ST)$ bits. Utilizzando una ricerca esaustiva sullo spazio delle chiavi, una workstation impiegherebbe circa T secondi per trovare la chiave. Ci sono due problemi con questo approccio:

- Un attacco di tipo brute force è facilmente parallelizzabile, N computers potrebbero impiegare meno tempo ad estrarre la chiave
- Il tempo computazionale di T secondi fornisce solo una stima, il tempo potrebbe cambiare significativamente, in relazione all'ordine in cui si analizzano le chiavi.

3.2.2 Time-lock puzzle basati su elevazioni di potenza

Sia M il messaggio che Alice vuole cifrare con un time-lock puzzle di periodo T secondi, vediamo i passi per creare e per risolvere il puzzle.

- Creazione del puzzle:
  - Alice genera un modulo composto del tipo $n = pq$ e computa $\phi(n) = (p-1)(q-1)$
  - Alice computa t=TS, con S numero di elevazioni a potenza modulo n al secondo che colui che deve rispolvere il puzzle può computare.

- Alice genera una chiave random K, grande abbastanza da rendere la ricerca esaustiva inapplicabile.
- Alice cifra M usando la chiave K e ottiene il criptotesto $C_M = Enc_k(M)$
- Alice prende un elemento random a modulo n e cifra K nel seguente modo
  * $C_K = K + a^{2^t} (mod\ n)$

  Per effettuare questo in maniera efficiente, Alice computa i seguenti valori
  * $e = 2^t (mod\ \phi(x))$
  * $b = a^e (mod\ n)$
- Alice produce in output il seguente time-lock puzzle

$$(n, a, t, c_K, C_M)$$

  e distrugge qualsiasi altra variabile (come ad esempio p e q, create durante la computazione.

• Risoluzione del puzzle: Essendo infattibile estrapolare K direttamente, l'approccio più rapido per risolvere il puzzle è determinare

$$b = a^{2^t}$$

La conoscenza di $\phi(n)$ permetterebbe di ridurre drasticamente i tempi di computazione, motivo per cui i fattori p e q devono essere distrutti. L'unico modo di risolvere il puzzle e computare b è effettuare t elevazioni di potenza.
Il numero di elevazioni richieste per risolvere il puzzle può essere perfettamente controllato. In questo modo è possibile creare puzzle con diversi livelli di difficoltà. Cosa importante è il fatto che le elevazioni di potenza sono *intrinsecamente sequenziali*, questo ci permette di non preoccuparci dell'hardware sul quale si risolve il puzzle.

Capitolo 4

Verifable delay function

Le Vefifiable Delay Funcion (VDF) sono state formalizate per la prima volta nel 2018 da Boneh, Bonneau, Bunz e Fis [2] come funzioni nella forma:

$$f : X \to Y$$

che richiedono un tempo prestabilito per essere calcolate, indipendentemente dalla tecnologia impiegata, ma il cui riultato può essere verificato in maniera efficiente. In poche parole, è richisto un tempo T di step sequenziali per calcolare la funzione, ma dato un input x e l'output y chiunque è in grado di verificare in maniera efficiente che y=*f(x)*.
Il problema è quindi creare questa funzione che impieghi T passi per essere valutata, anche con uso massiccio di parallelismo, ma che possa essere verificata con un tempo trascurabile.

4.1 Possibili applicazioni

Prima di dare una precisa definizione e una descrizione di queste funzioni, vediamo alcune loro possibili applicazioni:

- Randomness beacon : Nelle proof-of-work blockchains (ad esempio Bitcon) i partecipanti alla rete inviano soluzioni di puzzle computazionali per ottenere ricompense monetarie. Alla base della sicurezza delle proof-of-work blockchain c'è la convinzione che le soluzioni trovate abbiamo un'elevata entropia computazionale. I minatori più forti potrebbero manipolare questo risultato, rifiutando di inviare blocchi che producono un'emissione sfavorevole di beacon. Questo

attacco sarebbe possibile solo se il beacon fosse calcolato rapidamente. Se si utilizzasse una VDF con un ritardo sufficientemente lungo per calcolare il beacon, si vanificherebbe l'attacco.

- Blockchain efficienti in termini di risorse: Tra le crescenti preoccupazioni per la sostenibilità a lungo termine delle blockchain a *proof of work* come Bitcoin, ci sono stati sforzi per sviluppare blockchain effecienti in termini di risorse in cui i minatori investono in anticipo le stesse, per poi sfruttarle durante l'estrazione. L'estrazione efficiente delle risorse, tuttavia, soffre di attacchi di simulazione senza costi. Un metodo per contrastare questo problema sarebbe utilizzare un randomness beacon per selezionare nuovi leader a intervalli regolari. La probabilità di diventare leader sarebbe influenzata dalla qualità delle prove presentate dai minatori (in termine, ad esempio, di puntata o spazio). Un'idea che non richiede una maggioranza di nodi onesti e non collusi è quella proposta da Cohen.[4] L'idea è quella di combinare *prove di risorse* con delle *VDF*, e sfruttare il prodotto delle due tecniche come misura della qualità della blockchain. Questo garantirebbe che le risorse siano impegnate nella blockchain e non possano essere utilizzate per altri scopi. Questo processo imita il ritardo casuale nel trovare un blocco Bitcoin (ponderato dalla quantità di risorse ontrollate da ciascun minatore), ma senza che ciascun minatore esegua un grande calcolo parallelo.

- Proof of replication: Un'altra interessante applicazione delle VDF è la *proof of replication*. Si tratta di un tipo speciale di prova di archiviazione dei dati che richiede al prover di dedicare una memoria unica anche se i dati sono disponibili da un'altra fonte. Questo potrebbe essere usato per dimostrare che sono state memorizzate numerose repliche dello stesso file.
  Le *proof of replication* classiche sono in genere definite da un tipo di architettura client-server a chiave privata in cui il server dimostra al client che è in grado di recuperare dati (privati) del client. Il client può facilmente verificare usando la sua chiave privata. In questo caso l'obiettivo è verificare che un determinato server stia memorizzando una replica univoca di alcuni dati disponibili pubblicamente. Lerner ha proposto nel suo paper *prova di archiviazione di blockchain unica*[6] cifrature asimmetriche temporali per applicare ai file

una transformazione lenta utilizzando un unico identificatore come chiave.
Durante una sessione il verifier sfida periodicamente il server su blocchi del file campionati casualmente. Il server non dovrebbe essere in grado di rispondere rapidamente, la verifica che i blocchi ricevuti dal server siano corretti è invece rapida grazie all'asimmetria temporale della codifica.
Le VDF potrebbero essere implementate per migliorare l'approccio ideato da Lerner. Una Verifiable Delay Function potrebbe infatti essere usata come funzione di cifratura assimetrica nel tempo. Vediamo un po' più nello specifico la costruzione basata sulle VDF.
Al *replicator* viene fornito come input:

- un file
- un *id* univoco del *replicator*
- parametri pubblici $pp \xleftarrow{R} Setup(\lambda, t)$

in modo da essere in grado di computare una cifratura dell'intero file (usando la VDF) che richiederà T passi sequenziali. In più dettaglio, la codifica viene calcolata suddividendo il file in blocchi $B_1..B_N$ e calcolando $y_1..y_n$ dove $y_i = Eval(pp, B_i \bigoplus H(id||i)$ con H funzione hash resistente alle collisioni.
Per accertarsi che il *replicator* abbia archiviato questa copia, un *verifier* potrebbe eseguire una query su un blocco cifrato y_i. Se il gruppo univoco che codifica y_i non era stato in precedenza memorizzato, non potrà essere ricalcolato abbastanza rapidamente da ingannare il *verifier*.

- Computational timestamping: Tutti i sistemi *proof-of-stake* sono vulnerabili alle *long-range forks* a causa dei possibili comportamenti scorretti delle parti interessate. Nei protocolli *proof-of-stake* si assume una maggioranza onesta, poichè tutti sono interessati a mantenere un sistema corretto. Dopo che le parti interessate hanno però ceduto, non hanno più questo incentivo; la maggiorana potrebbe quindi colludere al fine di creare una *storia alternativa* del sistema fino a quel momento.
I meccanismi attuali presuppongono che ciò sia impedito attraverso un meccanismo di timestamp esterno che dimostra agli utenti quale

sia la corretta history del sistema. VDF incrementali potrebbero fornire prove computazionali che una data versione del sistema è più vecchia di un'altra. Ciò consentirebbe di rilevare *long-range forks* senza fare affidamento a meccanismi di timestamp esterni.

4.2 Definizione

Una VDF può essere definita mediante una tripla *(Setup,Eval,Verify)*:

- $Setup(\lambda, t) \rightarrow pp = (ek, vk)$ è un algoritmo randomizzato che prende in input un parametro di sicurezza λ e un tempo limite t, e ritorna in output il parametro publico *pp* composto da una chiave (ek) di valutazione e una chiave (vk) di verifica.
- $Eval(ek, x) \rightarrow (y, \pi)$ è un algoritmo che prende in input $x \in X$ e produce in output $y \in Y$ e una $proof$ π. *Eval* potrebbe richiedere dei bit random per generare la $proof$ π ma non per computare y. Per ogni *pp* generato da $Setup(\lambda, t)$ e per ogni $x \in X$ *Eval* deve computare in tempo t con $poly(log(t), \lambda)$ processori che lavorano in parallelo.
- $Verify(vk, x, y, \pi) \rightarrow$*{Yes,No}* è un algoritmo deterministico che prende in input vk, x, y, π e restituisce in output *Yes o No*. *Verify* deve computare in tempo polinomiale a *log(t)* e λ. E' quindi molto più veloce di *Eval*

Per essere definita tale, una VDF, deve soddisfare tre importanti proprietà:

- *ϵ-valutation time*: L'algoritmo *Eval* deve computare in tempo al massimo uguale a $(1+\epsilon)T$ per ogni $x \in X$ e ogni *pp* restituito da $Setup(\lambda, T)$.
- *sequentiality*: un algoritmo A parallelo che usa la massimo *poly*(λ) processori, e computa tempo minore di T non è in grado di computare correttamente la funzione. In particolare, preso x da X casualmente e sia *pp* l'output della funzione *Setup(*λ, T), se $(y, pi) \leftarrow$ *Eval(pp,x)* allora $Pr[A(pp, x) = y]$ è trascurabile.

- *uniqueness*: preso $x \in X$, una sola $y \in Y$ sarà accettata da *Verify*. In particolare sia A un algoritmo efficiente che preso in input pp restituisce (x, y, π) in modo tale che *Verify*$(pp, x, y, \pi) =$*Yes* allora $Pr[A(pp, x) = x]$ è trascurabile.

4.3 Sicurezza delle VDF

La proprietà di sicurezza richiesta dalle VDF è la $\sigma-$*sequentiality*. Si richiede che nessun avversario sia in grado di computare un output valido in tempo $\sigma(t) < t$ anche utilizzando parallelismo e precomputazione. Da notare è il fatto che un avversario con $|Y|$ processori può sempre computare outputs in tempo $o(t)$ provando tutti i possibili output in Y. Questo suggerisce che una VDF deve avere $|Y| > poly(t)$. Definiamo la seguente sfida applicata ad un avversario $A := (A_0, A_1)$:

- choose a random pp
  $pp \leftarrow_R Setup(\lambda, t)$
- adversary preprocess pp
  $L \leftarrow_R A_0(\lambda, pp, t)$
- choose a random input x
  $x \leftarrow_R X$
- adversary computes an output y_A
  $y_A \leftarrow_R A_1(L, pp, x)$

  (A_0, A_1) vincono il gioco se $y_A = y$ con $(y, \pi) := Eval(pp, x)$

Diciamo che una VDF gode della proprietà di $\sigma-$*sequentiality* se se non esiste un algoritmo A_0 che impega tempo totale $O(poly(t,\lambda))$ che vince la sfida illustrata prima con probrabilità maggiore di $negl(\lambda)$ contro un algoritmo A_1 che impiega tempo parallelo $\sigma(t)$ con al più $p(t)$ processori.
La definizione illustra il fatto che anche dopo la computazione dei parametri pp da parte di A_0 per un tempo lungo, l'avversario A_1 non è in grado di computare un output a partire dall'input x in tempo $\sigma(t)$ su $p(t)$ processori che lavorano in parallelo.

4.4 Due Verifiable Delay Functions

Le VDF, come già spiegato, si basano su task che non possono essere velocizzati utilizzando parallelismo. L'esponenziazione in un gruppo di ordine sconosciuto gode di questa proprietà, tanto da essere stato usato da Rivest, Shamir e Wagner per costruire le time-lock puzzle[9]. Proponiamo adesso due VDF functions, la prima ideata da Pietrzak[8], la seconda da Wesolowski[10].
Entrambi gli approcci operano come segue:

- L'algoritmo *Setup* restituisce due oggetti:
  - Un gruppo ciclico abeliano G di ordine sconosciuto (nel capitolo 4.4.4 vedremo come sarà scelto)
  - Una funzione hash crittografica efficiente $H : X \to G$ che sarà utilizzata come un random oracle.

  Si settano i parametri publici $pp := (G, H, T)$
- L'algoritmo di valutazione $Eval(pp, x)$ è definito nel seguente modo:
  - computa $y \leftarrow H(x)^{(2^t)} \in G$ computando T elevazioni di potenza in G partendo da $H(x)$
  - Computa la *proof* π come descritto in seguito (capitoli 4.4.1 e 4.4.2)
  - restituisce in output (y, π)

Misureremo il *running time* come il numero di operazioni nel gruppo G necessarie a computare la funzione. Si da per certo che computare y richieda T esponenziazioni sequenziali in G anche su computer paralleli con $poly(\lambda)$ processori. Come vedremo, computare la $proof \pi$ incrementerà il *running time* a $(1 + \epsilon)T$.
Serve stabilire come un verificatore pubblico possa verificare la correttezza del calcolo della funzione in maniera efficace. In questo punto i due approcci divergono, proponendo due differenti *public-coin* per provare la correttezza dell'output y. Per mantenere il problema il più astratto possibile, useremo la seguente notazione:

- con $g := H(x) \in G$ indichiamo l'elemento base dato come input alla funzione VDF

- con $h := y \in G$ indichiamo l'output della funzione VDF, diciamo $h = g^{(2^T)}$
- $T > 0$ è una quantità pubblica e nota.

L'entità con il compito di valutare la correttezza della VDF dovrà produrre una prova che, data la tupla (G, g, h, T) soddisfi $h = g^{(2^T)}$ *in* G

4.4.1 Wesolowski protocol

Data una tupla (G, g, h, T) in input, il *prover* e il *verifier* si scambiano i seguenti messaggi per provare che $h = g^{2^t}$ in G. Sia *Primes(*λ) un set contenente i primi 2^k numeri primi, quindi 2,3,5,7, etc.

- Il *verifier* controlla che $g, h \in G$
- Il *verifier* manda al *prover* un numero primo random l, estratto con distribuzione uniforme da $Primes(\lambda)$
- Il *prover* computa $q, r \in Z$ in modo tale che $2^t = ql + r$ con $0 \leq r < l$ e manda $\pi \leftarrow g^q$ al *verifier*
- Il *verifier* computa $r \leftarrow 2^t \ mod \ l$ e restituisce *Yes* se $\pi \in G$ e $h = \pi^l g^r \in G$

Il protocollo funzionerebbe allo stesso modo se il termine 2^t fosse un intero arbitrario e, e non necessariamente una potenza del due. Il *verifier* deve essere semplicemente in grado di computare velocemente $r := e \ mod \ l$.
Il *verifier* deve computare $r \leftarrow 2^t \ mod \ l$, effettuando $log \ T$ moltiplicazioni in Z/l. Deve poi effettuare due esponenziazioni piccole in G.
Il *prover* deve invece computare $\pi = g^q \in G$ con $q = \lfloor 2^t/l. \rfloor$. Essendo t grande serve computare π effettuando $2t$ operazioni nel gruppo usando l'algoritmo per le divisioni lunghe, in cui il quoziente è computato nell'esponenziazione base g.

- $\pi \leftarrow 1 \in G, r \leftarrow 1 \in G$
- ripetere t volte:
  - $b \leftarrow \lfloor 2r/l \rfloor \in \{0, 1\}$

- $r \leftarrow (2r \ mod \ l) \in \{0, ..., l-1\}$
- $\pi \leftarrow \pi^2 g^b \ \in \ G$

- restituisce π // che è uguale a g^q

4.4.2 Pietrzak's protocol

Pietrzak presenta una differente implementazione della creazione della *public coin proofs*. Data una tupla (G, g, h, T) come input, *prover* e *verifier* ingaggiano il seguente protocollo ricorsivo per provare che $h = g^{2^t} \ \in \ G$

- il *verifier* controlla che $g, h \ \in \ G$
- Se $t = 1$ il *verifier* controlla che $h = g^2 \ \in \ G$, restituisce *Yes,No* e si ferma.
- Se $t > 1$:
  - Il *prover* computa $v \leftarrow g^{2^{t/2}} \ \in \ G$ ed invia v al *verifier.*
  - Il *verifier* controlla che $v \ in \ G$
  - Il *prover* deve convincere il *verifier* che $h = v^{2^{t/2}}$ e che $v = g^{2^{t/2}}$. Questo proverebbe che $h = g^{2^t}$. Essendo lo stesso esponente usato in entrambe le equazioni, le due eguaglianze possono essere verificate simultaneamente controllando una combinazione lineare casuale del tipo:

    $$v^r h = (g^r v)^{2^{t/2}} \ , \ r \leftarrow^r \in \{1, ..., 2^\lambda\}$$

    nel seguente modo:
  - Il *verifier* manda al *prover* $\leftarrow^r \{1, ..., 2^\lambda\}$
  - *prover* e *receiver* computano $g_1 \leftarrow g^r v \ , \ h_1 \leftarrow v^r h \ \in \ G$
  - *prover* e *verifier* ingaggiano una prova ricorsiva che dimostra $h_1 = g_1^{(2^{t/2})}$

Il *verifier* ad ogni livello di ricorsione esegue due piccole esponenziazioni in G per computare g_1 , h_1, valori che servivanno al livello successivo di ricorsione. Verificare la prova richiede quindi circa $2log_2 T$ piccole

esponenziazioni in G.
Il *prover* deve computare v per ogni livello di ricorsione. Supponiamo v_1 , r_1 siano i valori in cima alla ricorsione, v_2 , r_2 quelli subito prima e così via.
Svolgendo la ricorsione otteniamo questi risultati:

$$v_1 = g^{(2^{t/2})}$$

$$v_2 = g^{(2^{t/4})} = (g^{r_1} v_1)^{(2^{t/4})} = (g^{(2^{t/4})})^{r_1} g^{(2^{3t/4})}$$

$$v_3 = g^{(2^{t/8})} = (g_1^{r_2} v_2)^{(2^{t/8})} = (g^{r_1 r_2} v_1^{r_2} v_2)^{2^{t/8}} = (g^{(2^{t/8})})^{r_1 r_2} (g^{(2^{3t/8})})^{r_1} (g^{(2^{5t/8})})^{r_2} g^{2^{7t/8}}$$

$$v_4 = g^{(2^{t/16})} = \mathit{una\ potenza\ di\ otto\ elementi} \ = \ g^{(2^{t/16})}\ ,\ g^{(2^{3t/16})}\ ,\ g^{(2^{5t/16})}\ ,\ ...\ g^{(2^{15t/16})}$$

Il pattern che emerge suggerisce un possibile modo per la costruzione della *proof* π. Quando viene computato per la prima volta l'output della funzione ($h = g^{2^t}$), vengono salvati 2^d elementi del gruppo $g^{(2^{(i \cdot t/2^d)})}$ per $i \in \{0 \dots \ 2^d - 1\}$ che vengono computati comunque durante il calcolo. Successivamente, questi valori memorizzati permetteranno di calcolare gli elementi del gruppo $v_1 \dots v_d$ necessari per la *proof* effettuando 2^d piccole esponenziazioni in G. Il *prover* computa i rimanenti elementi $v_{d+1}, v_{d+2}, \dots, v_{logT}$ elevando $g_{d+1}, g_{d+2}, \dots, g_{logT}$ all'esponente adeguato. Questo passo costa di esattamente $T/2^d$ moltiplicazioni in G. Il tempo totale impegato per calcolare la *proof* è $2^d + t/2^d$. Questo ci suggerisce che $d = 1/2log_2t$ è una quantità ideale.
L'output della VDF e la *proof* possono essere computate in circa $(1 + 2/\sqrt{t})t$

4.4.3 Confronto fra i due protocolli

Entrambi gli approcci proposti sono validi, e nessuno domina l'altro.
La *proof system* di Wesolowski produce una *proof* più piccola (un elemento del gruppo, contro i log_2t elementi di Pietrzak) e la verifica è più rapida (due esponenziazioni, contro $2log_2t$). L'approccio di Pietrzak[8] ha però due vantaggi:

- Prover efficiency: per le VDF, l'approccio di Pietrzak risulta essere più efficiente. Impiega $O(\sqrt{t})$ operazioni interne al gruppo per costruire la prova, contro le $O(T)$ operazioni impegate da Wesolowski.

- Confronto delle ipotesi: Se il protocollo ideato da Wesolowski è sicuro, allora lo è anche quello di Pietrzak. Tuttavia il contrario non è detto sia vero.

4.4.4 Costruzione gruppi di ordine ignoto

Serve trovare il modo di creare un gruppo G il cui ordine è ignoto e difficile da calcolare. Gli approcci più interessanti, trattati da Boneh, Bünz e Fish [3] sono:

- Gruppo RSA: Sia $GGen$ un algoritmo che restitusice in output un intero N di fattorizzazione ignota. Computare l'ordine del gruppo moltiplicativo $G := (Z/N)^*$ è tanto difficile quanto fattorizzare N. Il gruppo G può quindi essere utilizzato come gruppo di ordine ignoto. La difficoltà sta nel creare un algoritmo $GGen$ che non lasci trapelare informazioni circa la fattorizzazione di N. Si potrebbe pensare di usare una randomness pubblica per scegliere un N sufficientemente grande da rendere la sua fattorizzazione praticamente impossibile.

- Campo quadrato numerico immaginario:Per risolvere il problema di un *trusted setup* si potrebbe ricorrere all'uso di un campo quadrato numerico immaginario $Q(\sqrt{p})$ con p primo negativo $p \equiv 1 \ mod \ 4$. Questo gruppo ha ordine dispari computazionalmente difficile da calcolare se $|p|$ è grande.
  Il generatore di gruppo $GGen(\lambda)$ restituirebbe quindi un primo negativo p.

Capitolo 5

Continuous Verifiable Delay Funcitons

Una cVDF, è intuitivamente una funzione sequenziale in cui ogni passo intermedio può essere verificato.[7] Sia x_0 il punto di partenza, $x_t = Eval^{(t)}(x_0)$ il t-esimo passo della cVDFe e sia inoltre B un upper bound del numero totale di step nella computazione; per ogni passo $t < B$, vogliamo le seguenti proprietà:

- Completeness: x_t può essere verificato come il t-esimo stato in tempo $polylog(t)$
- Adaptive Soundness: Qualsiasi valore $x'_t \neq x_t$ computato da un avversario non è in grado di verificare il t-esimo stato. Ogni stato è (computazionalmente) unico.
- Iteratively Sequential: Dato un generico x_0, un avversario non è in grado di computare x_t in tempo $(1 - \epsilon) \cdot t \cdot l$, con l tempo che impiega un agente onesto per eseguire un passo di computazione.

5.1 Possibili costruzioni di cVDF

- Utilizzando una VDF: un primo approccio banale è quello di usare una VDF per realizzaer una cVDF. Basterebbe iterare la VDF come una catena di computazioni. Per ogni singolo step di computazione, da x_0 a x_T ad esempio (con T difficoltà di base), si deve creare una *proof* $\pi_{0 \to t}$ che dimostri la correttezza da $0 \to T$. Dovremmo poi creare una nuova instanza di VDF, con relativa *proof* $\pi_{t \to 2T}$ che attesti la correttezza da $t \to 2T$. A questo punto risulta facile

verificare la correttezza di x_{2T}, verificando entrambe le *proof*.
Questa soluzione ha la proprietà che dopo t steps, un altro agente può, prelevando il valore $x_{t \cdot T}$, continuare la catena VDF. Questo approccio ha però un difetto, la *proof* finale $\pi_{(t-1) \cdot T \to t \cdot T}$ non garantisce nulla circa le computazioni dal passo 0 al passo $(t - 1 \cdot T)$. Per verificare la correttezza di tutta la catena serve verificare tutte le *proof*.
Un'idea possibile per risolvere il problema di cui sopra sarebbe quella di usare una generica *proof*, ottenuta dall'unione di tutte le $proof$ della catena.

- Utilizzando i logaritmi: Avendo la possibilità di iterare la strategia sopra menzionata per qualsiasi T, si può pensare di eseguire $log\ B$ catene di VDF indipendenti in parallelo con $T = 1, 2, 4, \ldots, 2^{logB}$ x_0. Problema di questo approccio è la ovvia mancanza della natura distribuita di una VDF continua. Ogni passo che il *prover* esegue per calcolare x_t non è più un'istanza indipendente di una computazione di VDF. Il *prover*, durante il calcolo di x_t, mantiene alcuni stati interni che serviranno per computare valori non ancora completi al passo t. Il *verifier* non ha modo di andare a prelevare questi valori. La soluzione in questo caso sarebbe quella di utilizzare un TTP che mantiene gli stati interni della VDF per un lungo periodo di tempo. L'utilizzo del protocollo proposto da Pietrzak permette al *verifier* di accedere, e quindi verificare, lo stato interno del *prover*.

- Utilizzando il protocollo Pietrzak: ricordiamo che il *prover* computa innanzitutto $u = x^{2^{t/2}}$, computa poi $u^{2^{t/2}} = y$.Provare ricorsivamente entrambe le espressioni risulta molto costoso in termini di tempo. Pietrzak propone di usare una *random challenge* r in modo da poter combinare entrambe le equazioni in una singola prova, in particolar modo si computa $u^r y = (x^r u)^{2^{t/2}}$ vera se e solo se le prime due equazioni sono vere.
Sottolineiamo il fatto che provare che $u^r y = (x^r u)^{2^{t/2}}$ è equivalente a provare l'equazione originale, ma con difficoltà $t/2$
Guardando il protocollo, è chiaro come gli unici stati interni che il *prover* deve mantenere sono $u =^{2^{t/2}}$ e l'output $y = x^{2^t}$. Volessimo provare la correttezza della computazione in qualsiasi momento,

dovremmo verificare la correttezza di u e quella di y.
Si noti che provare la correttezza di u richiede una VDF, indipendente dalla prima, con complessità $t/2$.
Questo è vero in una struttura ad albero ricorsiva in cui ogni computazione di t steps consiste nel provare un'istanza dell'albero di dimensione $t/2$. In modo particolare si dovrà computare :

- $u = x^{2^{t/2}}$
- $y = u^{2^{t/2}}$
- $u^r y = (x^r u)^{2^{t/2}}$

Una volta provate queste tre istanze, si ha la prova diretta che il *genitore* dell'istanza x^{2^t} è y.
Questo suggerisce un'idea per costruire VDF continue; partendo dalla radice, nella quale vogliamo computare x^{2^t}, si computa e prova ricorsivamente ogni sotto-istanza dell'albero.
In modo particolare, ogni step della cVDF è un passo dell'attraversamento dell'albero. In ogni punto, se tutte e tre le sotto-istanze del nodo sono state provate, basta *unire* le *proof* in una *proof* unica e dimenticare i calcoli effettuati nelle sotto-istanze.
In questo modo otteniamo due proprietà desiderate nelle cVDF:

- in ogni punto un nuovo membro può verificare lo stato prima di continuare la computazione
- grazie alla struttura della *proof*, la dimensione della stessa è limitata dall'altezza dell'albero.

Proof merging: questo approccio si basa fortemente sulla tecnica di *merge* delle prove. Le prove delle sotto-istanze devono quindi essere efficacemente unite in una prova di quel nodo padre. Questo può essere ottenuto grazie alla struttura delle prove nel protocollo di Pietrzak.

5.2 Costruzione di una Continuous VDF

L'idea che vogliamo approfondire è quella basata su una struttura ad albero in cui ogni stato intermedio del calcolo può essere verificato e le prove del calcolo possono essere efficacemente unite.

Nello specifico, i passi di computazione corrispondono ad un attraversamento del (k+1)ario albero di altezza $h = log_k B$. Ad ogni nodo dell'albero è associato uno *statement (x,y,t,π)* riguardante la VDF sottostante.
Si ricorda che $y = x^{2^t}$ e π è la rispettiva *proof of correctness*, sia x il nodo input, y il nodo output, π la *proof* e t la difficoltà.
La difficoltà è determinata dall'altezza dell'albero. Ad esempio, un nodo a distanza l dalla radice ha difficoltà $t = k^{h-l}$.
L'albero è definito come segue. Partendo dalla radice, con input x_0 e difficoltà $t = k^h$ dividiamo t in k segmenti $x_1, \ldots, x_k$, in modo analogo alla nostra costruzione della VDF. Questo segmento va a creare l'input e l'output dei primi k figli: il figlio i-esimo avrà input x_{i-1} e output x_i e richiede una *proof* che $(x_{i-1})^t/k = x_i$.
Il suo$(k + 1)$-esimo figlio corrisponde ad un nodo in cui i k *statement* dei fratelli sono uniti in un singolo *statement*. Dividere ricorsivamente gli *statement* in questo modo permette di assegnare gli stessi *statement* ad ogni nodo dell'albero finchè non si raggiunge la foglia; nodo in cui l'elevazione di potenza può essere effettuata direttamente.
Utilizzando questra struttura si nota come solo le foglie richiedono computazione, gli *statement* dei nodi a livello più alto possono essere dedotti dagli *statement* dei figli. Vorremmo che ogni step di computazione nella nostra costruzione corrisponda alla computazione di uno *statement* di una singola foglia. Per raggiungere questo scopo dobbiamo essere in grado di computare l'input x della foglia dallo stato precedente. Si osserva come questo richiede la sola conoscenza di una piccola quantità di nodi che sono già stati computati, questi nodi saranno chiamati d'ora in avanti *frontiera*.
La frontiera di una foglia s, *frontier(s)*, contiene tutti i fratelli di sinistra dei suoi antenati, inclusi i fratelli di sinistra di s stesso. Ogni stato della computazione contiene inltre una *leaf label s*, e lo *statement* associato alla *frontier(s)*. Un singolo step della nostra cVDF, dato uno stato $v = (s,frontier(s))$, inizialmente verifica v per poi computare $v' = (s',frontier(s'))$ con s' la foglia successiva ad s. Nella figura 5.1 possiamo notare i primi sei stati della cVDF con k=3 e difficoltà $D = k^{d'}$, con d' costante. In ogni albero, i segmenti solidi che congiungono due nodi stanno ad indicare un *segment node*, le linee tratteggiate un *sketch node*. I nodi in giallo indicano la foglia corrente, quelli rosa indicano

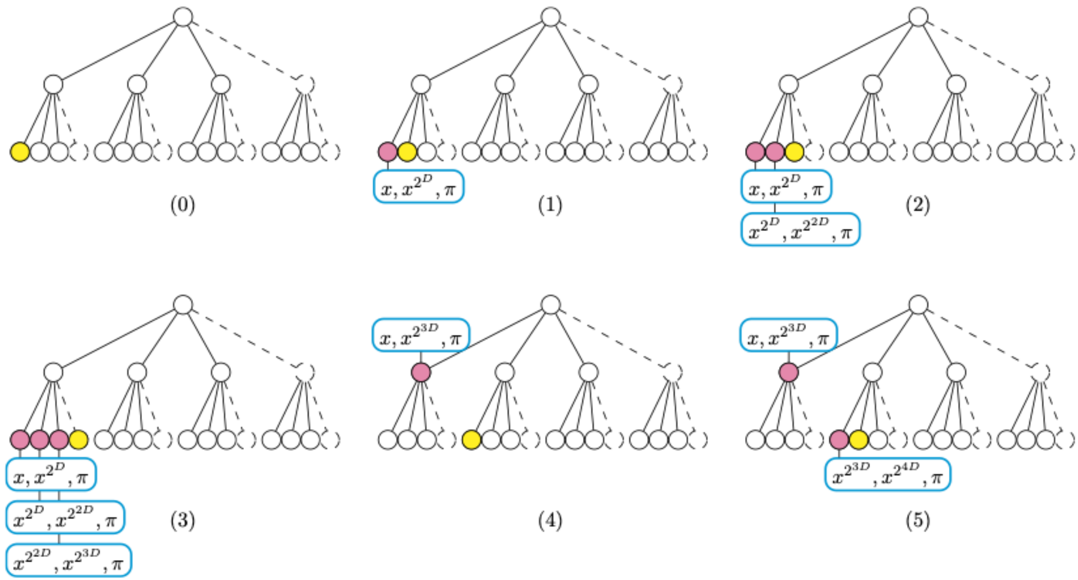


Figura 5.1: Illustrazione della modalità di computazione dello stato successivo

la frontiera. Nei riquadri blu contengono (x, y, π) del nodo corrispondente. La *proof* π di un nodo foglia con input y corrisponde alla VDF sottostante in cui $x^{2^D} = y$ e la prova di ogni nodo più alto nell'albero consiste nel suo *segment* e nella *proof* dello *sketch node*.

Capitolo 6

Implementazione protocolli

E' stato realizzato un programma, scritto interamente in C, che implementa i due approcci visti in precedenza. Il programma da la possibilità di eseguire VDF utilizzando l'idea di Wesolowski e quella di Pietrzak e di comparare le prestazioni delle due implementazioni.

6.1 Librerie utilizzate

6.1.1 Multiple Precision Arithmetic Library

GMP è una libreria che da la possibilità di gestire numeri con precisione arbitraria. L'unico limite è dettato dalla disponibilità di memoria.
La libreria possiede un set ampio di istruzioni che permettono di fare operazioni aritmetiche su MPI. L'interfaccia base è per C ma esistono diversi wrapper per diversi linguaggi di programmazione (Ada,C++, C#,Julia,PHP,Python ad esempio).
La maggior parte di applicazioni che usano questa libreria sono applicazioni di tipo crittografiche o di internet security. La prima release fu rilasciata nel 1991, e da quel momento la libreria è costantemente sviluppata e mantenuta. Il tipo utilizzato è *mpz_t* che supporta interi a precisione arbitraria. Di seguito un elenco delle funzioni più utilizzate all'interno del codice prodotto.

- mpz_inits(mpz_t x, ..., NULL): inizializza un numero arbitrario di mpz_t
- mpz_set(mpz_t rop, mpz_t op) : assegna op ad rop
- mpz_set_ui(mpz_t rop, unsigned long int op) : assegna op ad rop

- mpz_mul(mpz_t rop, mpz_t op1, mpz_t op2) : esegue il prodotto fra op1 e op2, salvando il risultato in rop
- mpz_mul_ui(mpz_t rop, mpz_t op1, unsigned long int op2) : esegue il prodotto fra op1 e op2, salvando il risultato in rop
- mpz_cdiv_qr(mpz_t q, mpz_t r, mpz_t n, mpz_t d): esegue il rapporto fra n e d, salvando il quoziente su q e l'eventuale resto su r
- mpz_powm(mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod): esegue un'esponenziazione modulare e salva il risultato su rop
- mpz_cmp(mpz_t op1, mpz_t op2): confronta op1 con op2

6.1.2 Libreria Nettle

Libreria GNU che implementa una serie di strumenti crittografici. Anche in questo caso la libreria è scritta in C e assembly ma esistono vari binding (C++, Python, Perl ad esempio): Nettle è una libreria libera e ad oggi ancora sviluppata e mantenuta.
Gli strumenti che mette a disposizione sono:

- funzioni hash (SHA-1, famiglie SHA-2 e SHA-3,..)
- cifrari a blocchi (AES, Blowfish, 3-DES) e a stream (RC4,..)
- funzioni MAC (HMAC, UMAC,POLY1305,..)
- schemi crittografici asimmetrici (RSA, DSA,ECDSA)

Di seguito un elenco delle funzioni più utilizzate all'interno del codice prodotto.

- sha1_init(struct sha1_ctx *ctx): inizializza il contesto
- sha1_update(struct sha1_ctx *ctx, size_t length, uint8_t *data): aggiorna il contesto
- sha1_digest(struct sha1_ctx *ctx, size_t length, uint8_t *digest): crea il digest

6.1.3 Ulteriori librerie

Durante la stesura del codice si sono rivelate molto utili alcune delle librerie messe a disposizione dal Professore Mario Di Raimondo dell'Università di Catania. Le librerie utilizzate sono state:

- lib-misc : libreria di supporto con varie funzioni:
  - estrazione seed random dal pool di sistema per sistemi Linux, Mac OS X e Windows
  - selezione della dimensione di un gruppo finito affinchè questo sia sicuro contro attachi al logaritmo discreto utilizzando algoritmi non-generici
- lib-timing: libreria per il campionamento preciso del timing attraverso varie chiamate di sistema (POSIX e non) e attraverso l'uso del contatore Intel TSC
- lib-mesg: libreria di supporto per la gestione del messagging a livelli di priorità

6.2 Gestione delle VDF

E' stata creata una libreria di supporto per la gestione e l'utilizzo di VDF. La libreria mette a disposizione una serie di API che permettono con facilità di creare una VDF, quindi utilizzarla e creare *proof* ricalcando l'idea di Wesolowski o quella di Pietrzak.

6.2.1 Strutture utilizzate

La scrittura del programma ha reso necessaria la creazione di diverse strutture, atte a contenere e racchiudere valori di tipo diverso. Le *struct* utilizzate sono state le seguenti:

- *pp_struct*: questa struttura conterrà i parametri pubblici della simulazione. In particolar modo avrà i seguenti campi:
  - *mpz_t N*: numero sicuro che sarà il modulo sul quale lavoreremo

- *unsigned long T*: numero di operazioni da effettuare all'interno del gruppo

- *output_eval_struct*: questa struttura andrà a contenere tutti i valori restituiti dalla computazione della VDF. In particolar modo conterrà:
  - *mpz_t g*: che corrisponde all'interpretazione decimale della funzione hash applicata sull'input della funzione
  - *mpz_t h*: che rappresenta l'output della funzione
  - *mpz_t proof*: che corrisponde la *proof* prodotta
- *proof_metadata_struct_W*: contiene tutti i metadati prodotti dalla computazione della *proof* durante il protocollo ideato da Wesolowski. Contiene i seguenti campi:
  - *mpz_t l*: che conterrà il numero primo previsto dal protocollo
  - *mpz_t q*: che rappresenta l'output della funzione
  - *mpz_t r_prover*
  - *mpz_t r_verifier*
  - *mpz_t _tmp*: che contiene una variabile di comodo 2^t
- *proof_metadata_struct_P*: contiene tutti i metadati prodotti dalla computazione della *proof* durante il protocollo ideato da Pietrzak. Contiene i seguenti campi:
  - *mpz_t v*: che conterrà $g^{2^{t/2}}$
  - *mpz_t g1_prover*
  - *mpz_t h1_prover*
  - *mpz_t g1_verifier*
  - *mpz_t h1_verifier*

6.2.2 API presenti

La libreria offre una serie di funzioni richiamabili direttamente dal main che permettono di creare, verificare e stampare il risultato di una VDF. Le funzioni presenti sono le seguenti:

- *Setup*: questa funzione (uguale per entrambi gli schemi) si occupa di inizializzare l'ambiente per la creazione della VDF prende in input:
  - la struttura che ospiterà tutti i parametri pubblici *pp_struct*
  - la grandezza di N, in numero di bits
  - il generatore di numeri casuali
  - T, numero di operazioni da dover effettuare all'interno del gruppo G

  in modo particolare si occupa di:
  - generare il gruppo finito G
  - settare l'hash da utilizzare (di default si utilizza SHA-1)
  - impostare il numero T di operazioni da dover effettuare all'interno del gruppo
- *Eval*: questa funzione computa la VDF prende in input:
  - un valore x, input della VDF
  - i parametri pubblici
  - la struttura che conterrà i risultati di *Eval*, *output_eval*
- *Start_W_Protocol*: la funzione provvede a creare la proof secondo il protocollo di Wesolowski; prende in input:
  - i parametri pubblici
  - *output_eval*
  - la struttura che conterrà tutti i metadati prodotti, *proof_metadata*
- *Start_P_Protocol*: la funzione provvede a creare la proof secondo il protocollo di Pietrzak; prende in input:
  - i parametri pubblici
  - *output_eval*
  - la struttura che conterrà tutti i metadati prodotti, *proof_metadata*
- *verify_proof_W*:la funzione esegue la verifica della VDF; prende in input:

- i parametri pubblici
- *proof_metadata*
- *output_eval*

e restituisce come ouptut:

- *true* se la verifica è andata a buon fine
- *false* altrimenti

• *verify_interactive_proof_P*:la funzione esegue la verifica interattiva della *proof* prevista da Pietrzak; prende in input:

- i parametri pubblici
- *proof_metadata*
- *output_eval*

e restituisce come ouptut:

- *true* se la verifica è andata a buon fine
- *false* altrimenti

6.2.3 Funzioni interne alla libreria

Alcune funzioni sono state rese inaccessibili dal *main*, ma sono servite nell'implementazione dei protocolli. Le funzioni sono le seguenti:

• *getSecureN*: la funzione ha lo scopo di generare un numero N sicuro, prende in input:

- *pp*, struttura che contiene tutti i parametri pubblici
- la grandezza di *N*, in numero di bits
- un generatore di numeri casuali

e va a salvare nel campo N di *pp* un numero della forma $q \cdot p$.

• *getDigestSHA1*: la funzione prende in input:

- un intero x
- un *mpz_t sum*

e va a salvare in *sum* l'interpretazione decimale dell'hash (SHA1) applicato sull'input x;

- *getRandom_ mplz*: la funzione prende in input:

  – *proof metadata*

  e si occupa di generare un numero primo casuale estraendo un *seed* casuale dall'interfaccia offerta dal sistema operativo

- *find_ q_ r*: richiamata all'interno della funzione *Start_ W_ Protocol*, ha il compito di calcolare q r come indicato dal protocollo Wesolowski

- *crafting_proof_ W*: anch'essa richiamata all'interno della funzione esposta *Start_ W_ Protocol* ha il compito di computare la *proof* come indicato dal protocollo Wesolowski

- *compute_ v*:richiamata all'interno della funzione *Start_ P_ Protocol* ha il compito di computare la v come indicato dal protocollo Pietrzak

- *prover_ computations* e *verifier_ computations*, richiamate anch'esse all'interno della funzione *Start_ P_ Protocol* hanno il compito di calcolare g_1 , h_1 come indicato dal protocollo Pietrzak.

```
[INFO]Starting eval..
   [EVAL]H(x): 'sum' (162 bit): 3225231641.....1781708355
   [EVAL]F(H(x)): 'output_eval->h' (4093 bit): 6974489796.....443357967
[BENCH] Evalutation VDF time: 417.091142000 s
[INFO]End eval..
```

Figura 6.1: Output funzione Eval

6.3 Risultati ottenuti

Sul codice prodotto sono stati fatti diversi esperimenti. Verranno in seguito riportati i risultati ottenuti.
Nella figura 6.1 possiamo notare quanto tempo richieda la computazione della VDF. Per l'esperimento è stato fissato un tempo $T = 100000$. Nelle figure 6.2 e 6.3 possiamo vedere il tempo impiegato dalle funzioni per costruire la proof, rispettivamente nel protocollo di Pietrzak e in quello di Wesolowski.
Infine nelle figure 6.5 e 6.4 si nota quanto rapida sia la verifica delle due funzioni. Sono soddisfatto del risultato ottenuto in quanto i tempi ottenuti sono in linea con quelli previsti.

6.4 Possibili sviluppi del software

Possibili sviluppi del software prodotto sono:

- l'implementazione delle cVDF: queste permetterebbero di andare a verificare la funzione *in punti intermedi* e non solo al completamento della computazione. Questa caratteristica potrebbe essere utile in diverse applicazioni.

- l'implementazione su rete dei protocolli mediante socket: si può pensare di ampliare il codice prodotto in modo tale da permettere a *Prover* e *Verifier* di contribuire, da remoto, all'esecuzione del protocollo. In questo modo sarebbe possibile computare e verificare VDF anche fra utenti che non risiedono nella stessa macchina.

```
Starting Pietrzak's procolol..
   [P-PROTOCOL]prover computes v: 'proof_metadata_pietr->v' (4096 bit): 8029112419.....631978807
   [P-PROTOCOL]prover computes h1: 'proof_metadata_pietr->h_1_prover' (4095 bit): 3648689341.....0096785269
   [P-PROTOCOL]prover computes g1: 'proof_metadata_pietr->g_1_prover' (4096 bit): 5788702326.....149877216
   [P-PROTOCOL]verifier computes h1: 'proof_metadata_pietr->h_1_verifier' (4095 bit): 3648689341.....0096785269
   [P-PROTOCOL]verifier computes g1 'proof_metadata_pietr->g_1_verifier' (4096 bit): 5788702326.....149877216
[BENCH]Crafting proof Pietrzak's protocol time:412.843610000 s
```

Figura 6.2: Output funzione Crafting_proof_P

```
Starting Wesolowski's procolol..
   Random Prime Number: 'pp->prime' (10 bit): 1021
  [W-PROTOCOL]verifier->prover: 'proof_metadata->l' (10 bit): 1021
  [W-PROTOCOL]prover computes: 'proof_metadata->q' (99991 bit): 9784545475.....756986395
  [W-PROTOCOL]prover computes: 'proof_metadata->r_prover' (7 bit): 81
  [W-PROTOCOL]prover->verifier: proof(g*q) 'output_eval->proof' (4094 bit): 2602003157.....8488156156
[BENCH]Crafting proof Wesolowski's protocol time: 444.895093000 s
```

Figura 6.3: Output funzione Crafting_proof_W

```
[INFO]Starting verify P: h1_prover?=g1_verifier^(2^(T/2))
   [P-PROTOCOL-VERIFY]h1_prover 'proof_metadata_pietr->h_1_prover' (4095 bit): 3648689341.....0096785269
   [P-PROTOCOL-VERIFY]g1_verifier^(2^(T/2)) 'proof_metadata_pietr->_g_1' (4095 bit): 3648689341.....0096785269
*-*-*-*-*-*-*-*-*-*-SUCCESS*-*-*-*-*-*-*-*-*-*-
[BENCH]Verify Pietrzak's protocol time: 0.025072000 s
[INFO]End verify P..
```

Figura 6.4: Output funzione Verify_P

```
[INFO]Starting verify W
  [W-PROTOCOL-VERIFY]: 'output_eval->h' (4093 bit): 6974489796.....443357967
  [W-PROTOCOL-VERIFY]: 'h_verifier_in_G' (4093 bit): 6974489796.....443357967
*-*-*-*-*-*-*-*-*-*-SUCCESS*-*-*-*-*-*-*-*-*-*-
[BENCH]Verify Wesolowski's protocol time: 0.285881000 s
[INFO]End verify W..
```

Figura 6.5: Output funzione Verify_W

Bibliografia

[1] A.Shamir A. Fiat. How to prove yourself. *Proc. of Crypto*, 1988.

[2] Bonneau J. Bu nz B. Fisch B Boneh, D. Verifiable delay functions. *Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018.*, page 757–788, 2018. Springer International Publishing, Cham.

[3] Peter Caxton. The title of the workfafafaaa. How it was published, The address of the publisher, 7 1993. An optional note.

[4] B. Cohen. Proofs of space and time. *Blockchain Protocol Analysis and Security Engineering*, 2017.

[5] Naor M.B Dwork, C. Pricing via processing or combatting junk mail. *Brickell, E.F. (ed.) Advances in Cryptology — CRYPTO' 92.*, page 139–147, 1992. Berlin Heidelberg, Berlin, Heidelberg (1993).

[6] S. D. Lerner. Proof of unique blockchain storage. 2014.

[7] I. Komargodski R.Pass N. Ephraim, C.Freitag. Continuous verifiable delay funcitons. *Cryptology ePrint Archive, report 2019/619*, 2019.

[8] K. Pietrzak. Simple verifiable delay functon. *Cryptology ePrint Archive, report 2018/627*, 2018.

[9] Shamir A. Wagner D.A. Rivest, R.L. Time-lock puzzles and timed-release crypto. *Tech. rep., Cambridge, MA, USA*, 1996.

[10] B. Wesolowski. Efficient verifiable delay funcions. *Cryptology ePrint Archive, report 2018/623*, 2018.

[11] Wikipedia. Criptovaluta. *https://it.wikipedia.org/wiki/Criptovaluta.*

[12] Wikipedia. Funzione crittografica di hash. *https://it.wikipedia.org/wiki/Funzione*$_c$*rittografica*$_d$*i*$_h$*ash*.

[13] Wikipedia. Funzione di eulero. *https://it.wikipedia.org/wiki/Funzione*$_{di_Eulero}$.

[14] Wikipedia. Gruppi algebrici. *https://it.wikipedia.org/wiki/Gruppo*$_($*matematica*).

[15] Wikipedia. Numero primo. *https://it.wikipedia.org/wiki/Numero*$_p$*rimo*.